A Qualitative Study of Vulnerability-Fixing Commits

Mouafak Mkhallalati

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

December 2019

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mouafak Mkhallalati**

Entitled: **A Qualitative Study of Vulnerability-Fixing Commits**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Tse-Hsun Chen

_____ Examiner
Dr. Jinqiu Yang

_____ Examiner
Dr. Nikolaos Tsantalis

_____ Supervisor
Dr. Emad Shihab

Approved by _____
Lata Narayanan, Chair
Department of Computer Science and Software Engineering

_____ 2019

_____
Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

A Qualitative Study of Vulnerability-Fixing Commits

Mouafak Mkhallalati

Security issues are a major concern in software development since the impact of exploiting security issues can be detrimental. Much of the prior work has proposed techniques that scan for and predict security vulnerabilities. However, in-depth, qualitative studies on software vulnerabilities are limited. Such studies can help the community better understand the types of vulnerabilities that exist and their potential impact in order to avoid them in the future.

Therefore, in this thesis, we present the results of studying security issues faced by developers. Our study leverages data provided by the SAP research team, which contains security fixing commits related to open source Java projects used by SAP and manually curated and validated by their researchers. We study a statistically significant sample of those commits. In particular, we collect information from the related repositories, issue trackers, documentation and advisories with the aim to comprehend and categorize such security issues. Also, we provide the context required to understand the issue along with code examples extracted from each of the categories in our study.

Our findings show that the vulnerabilities commonly facing developers are related to deserialization of untrusted data, zip slip, xml external entity processing, validation, authorization, race conditions, and information exposure. The fixes required to fix those vulnerabilities range from providing proper configuration of the used parser in the case of XML related issues to requiring in-depth knowledge of the code and the security issue as in vulnerabilities related to thread synchronization.

Acknowledgments

To my father and mother, the two people who I owe everything in my life to. To my father in law, a great man and a true father. I dedicate this work to you.

To my supervisor, Dr. Emad Shihab. without whom this work wouldn't have been possible, thank you for all the guidance and support.

To my lab mates who I had the opportunity to work with, thank you, you made this experience more enjoyable. I wish you all the success through your journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Problem Statement

Security is an important aspect of software development. Writing code that fulfills the required functionality should not be a separate activity from securing the code. However, studies have shown that developers focus on delivering the task in hand before thinking about security, and may use code snippets from online sources that are easily accessible and quickly understandable without thinking about their implications on the application security Acar et al. (2016). The majority of these snippets do not follow secure coding practices for the related task and may contain vulnerable code Fischer et al. (2017). Security experts think that developers are not getting proper security education, which leads to them writing insecure code and introducing security issues, which leads to additional cost and time to fix the issues later in the project life and it also increases the chance that some of these issues would slip into production Thomas, Tabassum, Chu, and Lipford (2018). In a recent survey[1] 70% of the surveyed developers said they think the security education they received was not adequate for what their current positions require. and three-quarters of college-educated respondents to the survey said they were never required to complete a single course focused on security during higher education.

[1] https://www.veracode.com/sites/default/files/pdf/resources/analystreports/the-devsecops-global-skills-survey-veracode-analyst-report.pdf

Writing secure code is not a trivial task, developers find it hard to correlate the task in hand with theoretical knowledge about the issue Oliveira et al. (2014). Security scanning tools are not a magical solution to secure the code, in fact developers not only have to understand the concept behind the alerts from the tools Imtiaz, Rahman, Farhana, and Williams (2019), but they also have to deal with false positives in the scanning results, which imposes an additional usage barrier due to the increase of the cost of using these tools Johnson, Song, Murphy-Hill, and Bowdidge (2013); Nadeem, Williams, and Allen (2012). We should also note that focusing the efforts on after-introduction remediation of security issues will keep us in firefighting type of solutions, and security related issues will continue to increase even through the life time of the project Mitropoulos, Gousios, and Spinellis (2012); Tan et al. (2014); Wen (2017).

To assist developers in battling security issues we should provide them with insights and proper knowledge that helps them understand those problems and lower their introduction rate. Evidently, fix rates for security issues increases with remediation coaching from security experts[2], which means that once the developers have proper understanding of the security issues, they take active measures to fix and avoid them.

To that extent, and given the two facts that 1) when fixing a security issue developers rely on support from colleagues who faced similar issues[1]; Assal and Chiasson (2019), and 2) open source is a rich knowledge source and open source communities are innovative and they foster learning and knowledge building Hemetsberger and Reinhardt (2006); Wen (2018), **we conduct a manual study on security issues in open source projects where we inspect the security fixing commits and aggregate additional information from issues trackers, bug reports, advisories, and related documentation to help studying the commits issues. Our study aims to increase the awareness of developers and the community about security with code samples extracted from open source projects that are commonly used and relied on. Our study -like previous similar studies which inspect the studied issues thoroughly- is**

[2]https://www.veracode.com/sites/default/files/pdf/resources/reports/state-of-software-security-developer-supplement-veracode-report.pdf

an essential step to understand those issues and help practitioners get a better grasp of them, it also motivates experts to look for better ways to remedy them.

1.1 Thesis Contribution

The main contributions of our study are the following:

- Thorough manual analysis of a statistically significant number of fixing commits from open source projects and external related information.
- Categorization of the studied fixing commits, which resulted in the proposal of 7 main categories.
- Explanation of the context of the studied security issues. For each category we provide code examples from the fixing commits.

1.2 Thesis Overview

Chapter 2 provides the literature review and related work. In Chapter 3, we describe our dataset and the methodology of our study. Chapter 4 shows the results of our study, we start with a preliminary analysis of the commits, list the categories of the commits that resulted from our analysis, we then go through each category and its subcategories to provide the context and code of the fix. Finally, in Chapter 5, we reflect on the results of our study and provide additional implications and suggestions based on our analysis and list our future work directions.

Chapter 2

Literature Review

2.1 Measurements and Characteristics

In this section we look at previous work that studied the aspects of security issues like metrics and other approaches that can be used to predict or characterize vulnerable code or commits. Frank et al. Li and Paxson (2017) conducted a large scale empirical study to compare bug fixes with security fixes. They compared them with respect to their timelines and the complexity of different types of patches and their impact on code bases. they find that security fixes have less complexity than non-security fixes and longer timelines providing attackers an opportunity to strike the software. Shin, Meneely, Williams, and Osborne (2010) evaluated three types of metrics, namely complexity, code churn, and developer activity metrics, in order to predict vulnerability location in the source code. Their model was mainly applied on two systems: Mozilla Firefox web browser and the Red Hat Enterprise Linux kernel. The model could predict approximately 80% of the vulnerable files. While these studies focused on studying vulnerabilities insights, our work focuses on studying the characteristics of vulnerability fixes. Sabetta and Bezzi (2018) proposed a machine-learning approach to find the security fixes commits. The approach mainly applies NLP classification methods on the source code changes. Their approach achieved acceptable precision and recall, 80% and 43%, respectively. Similarly, Wan (2019) proposed a deep learning approach to find security fixing commits based on the commit messages only.

Their approach achieved a precision of 88%, with a recall rate of 89%. They claim that their approach has achieved significantly better precision than state-of-the-art while improving the recall rate by 16.8%.

2.2 Available Resources

In this section we look at previous work that studied the resources available to developers during their studies and on their job tasks and how that affects the security aspect of the code they produce. Thomas et al. (2018) interviewed 32 security experts in a study to examine their views on application security processes and their interactions with developers. The majority of interviewees called for more security training for developers to provide them with a greater awareness of security and the knowledge necessary to produce fewer security bugs. In their study, they also indicate that finding and fixing security vulnerabilities so late in the software lifecycle can result in costly delays and expense to applications, and an increased likelihood that applications are not adequately secure. And that separating security from development adds communication overhead and barriers. Acar et al. (2016) systematically analyzed how the use of information resources impacts code security, they held a lab study where the participants were assigned tasks to finish in a tight time limit and unfamiliar starter code. Their finding is that real-world Android developers use Stack Overflow as a major resource for solving programming problems, including security and privacy relevant problems. And that other resources, such as official Android API documentation, do not provide the same degree of quickly understandable, directly applicable assistance. They also state that Stack Overflow contains many insecure answers, and that Android developers who rely on this resource are likely to create less secure code.

Fischer et al. (2017) studied the impact of copying code snippets from Stack Overflow on code security. They analyzed 1.3 million Android applications and found that 15.4% of them contained vulnerable code snippets that were very likely copied from Stack Overflow. The authors state that code security has a complex nature, and it is very difficult to provide ready-to-use and secure solutions for every problem. Hence, integrating a security-related

code snippet from Stack Overflow into production software requires caution and expertise.

Meng, Nagy, Yao, Zhuang, and Arango-Argoty (2018) conducted an empirical study on StackOverflow posts to understand developers' concerns on Java secure coding, their programming obstacles, and insecure coding practices. They found that the top five most popular programming challenges facing developers are authentication, cryptography, Java EE security, access control, and secure communication. Among the five categories they identified security vulnerabilities in the accepted answers of three frequently discussed topics which are Cross-site request forgery (CSRF), SSL/TLS, and Password Hashing. The authors reported security vulnerabilities in the suggested code of accepted answers on the StackOverflow forum. Their findings reveal the insufficiency of secure coding assistance and documentation, as well as the huge gap between security theory and coding practices.

2.3 Security Scanning Tools

A survey by Shahriar and Zulkernine (2012) presented different security scanning tools that help to mitigate program security vulnerabilities, including program vulnerability analysis and discovery methods published between 1994 and 2010. All program analysis approaches can be categorized into three main categories: (1) Static Analysis: a given program is analyzed based on its source code, without the need to execute it. (2) Dynamic Analysis: a given program is analyzed by executing it with specific input data and monitoring its runtime behavior. (3) Hybrid Analysis: a given program is analyzed with a mixture of static analysis and dynamic analysis techniques. In addition to the aforementioned security scanning methods, there is a different class of works that utilize techniques from the fields of data science and artificial intelligence (AI) to address the problem of software vulnerability scanning and discovery. This interesting class of approaches is neglected in the review of Shahriar and Zulkernine (2012), while it has received increasing attention by the research community in the following years (from 2011 onwards). Ghaffarian and Shahriari (2017) presented a categorized review of this class of tools in the field of software security scanning that utilize data mining and machine-learning techniques. One challenge for the proper

advancement of such tools is the lack of standard benchmarking datasets. The current state of evaluations by researchers, where they report performance results on self-gathered datasets, is not a reliable indicator of the proposed approaches' success or failure. A standard benchmarking dataset that is carefully crafted by considering many different aspects of a suitable software vulnerability dataset to provide the means for proper evaluation and comparison of different approaches. Therefore, in our work we focus on qualitatively analyzing the vulnerability fixing issues to better understand the vulnerability issues and their fixes, which provides a dataset that can help researchers and practitioners in the future.

2.4 Qualitative Analysis

In this section we look at previous work that manually analyzed and categorized security issues, we note that the studies we show in this section are the only ones we found in literature. Holzinger, Triller, Bartel, and Bodden (2016) studied exploits that break the complex sandbox security mechanisms in Java. For their study, they collect exploits information from various datasets. Then, they manually run the exploits to compare and cluster their behavior in order to find out weak spots of the Java platform which are used in many attacks. They point out that final goal of the exploits fall into three goals: Information Disclosure (reveal sensitive system information), Full Bypass (often achieved through disabling the active security manager), and Denial of service. In their work they explain inner workings in Java (like how *classloading* happens and how security check happens) and how they contribute to the security of the platform. Their results however are more interesting to Java language maintainers and are targeted towards improving the security architecture of Java. While our study differs from their study by focusing on security issues from the applications side and not the languages side.

Jimenez, Papadakis, Bissyandé, and Klein (2016) studied 42 vulnerabilities impacting the Android Operating System reported between 2008 and 2014. They build a taxonomy of the issues causing vulnerabilities and characterize the fixes made to correct the vulnerabilities. Their results show 8 causes of issues which are Resource Management, Data,

Semantic, Initialization Bug, Forget to remove debug features, Flow, Unauthorized Access, and Insecure Protocol. The first 4 causes are related to Code, 1 related to Test and 3 related to Design, respectively. They give further details on the Code related problems by providing 9 reasons for issues that fall under them, which are Buffer overflow, Incorrect pointer dereference, Stack consumption, Input not verified, Serialization issue, Unprotected use of a function, Missing/Incorrect implementation of a feature, Object not rightly created, and Wrong initialization of data. Another study that is related to vulnerabilities in the Android OS is by Linares-Vásquez, Bavota, and Escobar-Velásquez (2017). The goal of their study is to investigate Android-related security vulnerabilities reported over the history of the Android OS. They mined 660 vulnerabilities from official Android security bulletins and information available on the National Vulnerability Database, and for their manual analysis part they also looked at vulnerability related documents available on issue trackers, versioning systems. They check the CVE details to identify its hierarchy (vulnerability type), and to identify the subsystem affected by the vulnerability. As a result, they found the most frequent vulnerabilities affecting Android OS are related to weaknesses that affect the memory, data handling permissions, privileges, and access control. For each of their categories, they give an example with text description from the CVE and from the fixing commit. They link the vulnerabilities to the subsystems affected by the vulnerabilities, which are *Linux Kernel*, the *Native libraries* layer, the *Applications* layer, the *Android Framework*, and the *Hardware Abstraction Layer*. The manual analysis in the study does not focus on code snippets related to the security vulnerability. It is more geared towards categorizing and identifying the affected components.

Our study is in the same direction as previous studies that analyze security related issues. However, our study is focused on security issues from open source Java applications, whereas previous studies focused on the Java language or on Android OS. The dataset we start with is a collection of security fixes from various open source Java projects. Like previous work, we extend our initial dataset with information from issues trackers, bug reports, advisories, and related documentation. Additionally, we categorize the security issues based on the collected information. Finally, we provide examples and code snippets in addition to explanation of the issue and how it was fixed. Our study, like previous qualitative studies, can help the community better understand the types of vulnerabilities that exist and their potential impact in order to avoid them in the future.

Chapter 3

Methodology

In this chapter, we describe the dataset we used in our study, how we augmented our initial dataset with the additional information required for achieving our research goal and the process to obtain that information.

3.1 Data Collection and Curation

The main goal of this work is to qualitatively study the characteristics of security issues and their fixes. To that aim, 1) we study the complexity of security-related fixes by looking at a number of metrics related to the fixing commits; 2) we study the vulnerability fixing commits along with additional information we gather related to the commit in order to better understand the security issues facing application developers, we categorize such issues into seven main categories; 3) we provide examples from each category to explain the security issues and their fixes. Therefore, we have the need for collecting a dataset that contains security fixing commits and the context of these commits.

We resort to the published dataset by SAP Security Research Ponta, Plate, Sabetta, Bezzi, and Dangremont (2019). The dataset is collected over a period of four years. Also, the dataset contains 205 distinct open-source Java projects, 1,282 fixing commits that correspond to 624 unique vulnerabilities. The dataset is manually curated and validated by SAP Security Research team. Also, it is publicly available as a CSV file. Each entry in

the file contains *vulnerability id, project repository url, and commit id*. Vulnerability ID is a unique identifier of a vulnerability. Project repository url is the link to the project that contains the vulnerability. Commit ID is an identifier of a commit that fixed the vulnerability. The dataset is available under the Apache 2.0 license and it covered projects used in SAP products. Though such projects are well-maintained and popular Java projects which are used broadly by the OSS communities.

In our study, we examine the fixing commits, and this part is manually done at the source code level along with many other pieces of information we collected while inspecting the commits, and it definitely requires a significant amount of time and effort to review and validate the examination process. Therefore, for our analysis, we selected statistically significant sample with 95% confidence level and 5% confidence interval from the 1282 fixing commits (i.e., 296 commits).

To make our dataset of quality (to focus on the recent security issues facing application developers), we decided to select the 296 commits from the recent commits, i.e., *commits from 2017 and 2018*, we filtered the dataset to commits in those two years and ended up with 546 commits. Then, we randomly select the 296 commits from those filtered in the previous step.

3.1.1 Card Sort

Card sort is a method to organize and categorize knowledge Wood and Wood (2008), it consists of writing labels that represent the studied case, then sorting them and giving names to the groups that indicate what the cases in that group have in common. We used YAML[1] for this task. YAML is a human friendly data serialization standard, it allows expressing key-value pairs, lists, and nested structures which we used during our card sort task.

We followed the card sort technique in our study. After we gathered the information related to the studied vulnerability fix, we label the fix with short phrases that represent the case, in this step we have multiple labels for each case, the labels in this step aim to

[1] https://yaml.org/

capture the information about the problem, the fixing code, and the solution. For example, a commit[2] from *Jenkins* was labeled with the following labels: 'Form Validation', 'Check Permission', 'Fix Empty and Trim'. Next, we sort the cases that are similar or close to each other to become consecutive so that we can navigate them more easily. We revisited the cases in each group multiple times to make sure that they are sorted correctly. In the next step, we iterate the groups to see if they can be grouped under a more broad one. Then we give a name to each one of these groups.

3.1.2 Validation of the Manual Check

To validate the manual analysis, the categories of the commits were validated by a PhD student. The rich data required for the categorization was collected and initially validated by the thesis author, then the cases were reviewed and revalidated by the colleague. The author provided the 296 studied cases with the added information, labels, and categorization, and the PhD student went through the cases and verified that the categorization of the commits is suitable for the issues they solve. Whenever we failed to agree on a categorization, we sit together and go over the commit information and make an agreed decision. For the majority of the cases, there was an agreement on the classification of the vulnerability issues. To measure the overall agreement, we use Cohen's Kappa coefficient Cohen (1960). The resulting value is a number between -1 and 1, where zero or negative values mean no agreement, and values above 0.8 are considered good agreement. In our categorization, the level of agreement was +0.93 (277 out of 296), which is considered to be an excellent inter-rater agreement.

3.2 Complexity Measurement of Vulnerability Fixes

We start our study with a preliminary analysis of the security fixing commits. We look at a number of metrics related to the code affected by a vulnerability. Our goal from this analysis is to inspect how large are the changes required for the fixing commits and whether

[2]https://github.com/jenkinsci/jenkins/commit/d7ea3f40efedd50541a57b943d5f7bbed046d091

they span lots of files. We used four metrics for our analysis, and we focus on the actual code changes in the fixing commits meaning that we exclude changes in non-code files, i.e., non Java files. We also exclude *test files* from this analysis since our goal is to find the amount of code change required to remedy the security issue.

- **The number of code-files changed:** We use this metric to measure how many files were part of the changes, we only consider code files changed in the commit (i.e. no non-Java files and no Test files).

- **The entropy:** The entropy is an indicator of whether the change was focused in a few changed files or spread across them. High values of the entropy mean that the changes are more spread. We calculate the entropy as defined by the Shannon Entropy formula Hassan (2009); Xia, Shihab, Kamei, Lo, and Wang (2016). Entropy is computed as: $H(P) = -\sum_{n=1}^{k}(p_k * log_2\ p_k)$ where n: number of files changed, p_k: is the probability for file k. The probability is calculated by dividing the number of changes in a file on the total number of lines. For example, the entropy for a commit[3] is 0.094, two code files were changed in the commit, one of them had few changes while the other one had major changes. The entropy for another commit[4] is 1.366, the commit changed four code files, and all of them contained major changes. Minimal values of the entropy occur when one file has most or all of the changes, and the remaining files have few changes. While maximal values of the entropy occur when changes are similarly scattered across the files.

- **The number of lines of code added by the commit:** For each code file changed in the commit, we measure the number of added code-lines that contribute to the fixing commits.

- **The number of lines of code deleted by the commit:** We follow the same steps as the previous metric. Instead of considering the number of lines of code added by the commit, we consider the number of lines of code deleted by the commit.

[3] https://github.com/facebook/buck/commit/8c5500981812564877bd122c0f8fab48d3528ddf
[4] https://github.com/apache/struts/commit/19494718865f2fb7da5ea363de3822f87fbda264

We gather those metrics per commit and we analyse them to get an overview of how much effort is required to carry on the changes made to fix a security vulnerability in our dataset.

3.3 Manual Categorization of Vulnerabilities

To understand the security issues facing developers and how they mitigated them, we need to have a full picture of the context of the issues in our studied projects dataset. In order to facilitate that goal, we collected related data from various sources in addition to the data from the commits in order to build up a detailed description of the issues and their context.

The selected fixing commits were manually analysed with the goal of assigning a free text describing (i) the vulnerability issue discussed and (ii) the solution (fix) proposed/adopted., i.e., no predetermined suggestions or information were provided. After that, we group and categorize the information obtained for the vulnerabilities and their fixes. That said, we use an open card sorting technique to conduct our manual analysis.

The manual analysis is done in multiple steps to gather as much related information as available. We start by visiting the fixing commit link on GitHub and inspecting the *commit message and the changed code* in the commit. Then, we look at *bug reports if available, documentation, advisories, and other external information sources* that help to understand the vulnerability issue and the fix.

During the analysis of the code changes in the fixing commits we focus on the changes that are essential to implement the fix, a commit may have an essential change or addition of code in one or more files and also update other files to reflect the changes like a change in call parameters. In the examples we report in each category, we focus on the essential changes of the commit, without going into details about the related updates that result from the change or that are not essentially the core fix of the problem. We show the code examples in the categories in *diff* format, meaning that we highlight the new code with green and the removed code with red, code that is untouched uncolored. In cases where

whole fixing code was newly added to the commit we highlight the background of the code with green diff color.

Checking Commit Messages. Checking the commit message is the first thing to do when visiting the fixing commit on GitHub. In some commits, the commit messages were quite helpful and rich in information, and gave a sense of the problem that the commit is trying to address. For example, Figure 3.1 shows an example of a commit message that provides a high level of understanding about the security issue being fixed. We see in the figure that the commit message explains the intent of the fix, which is fixing plaintext storage of user password.

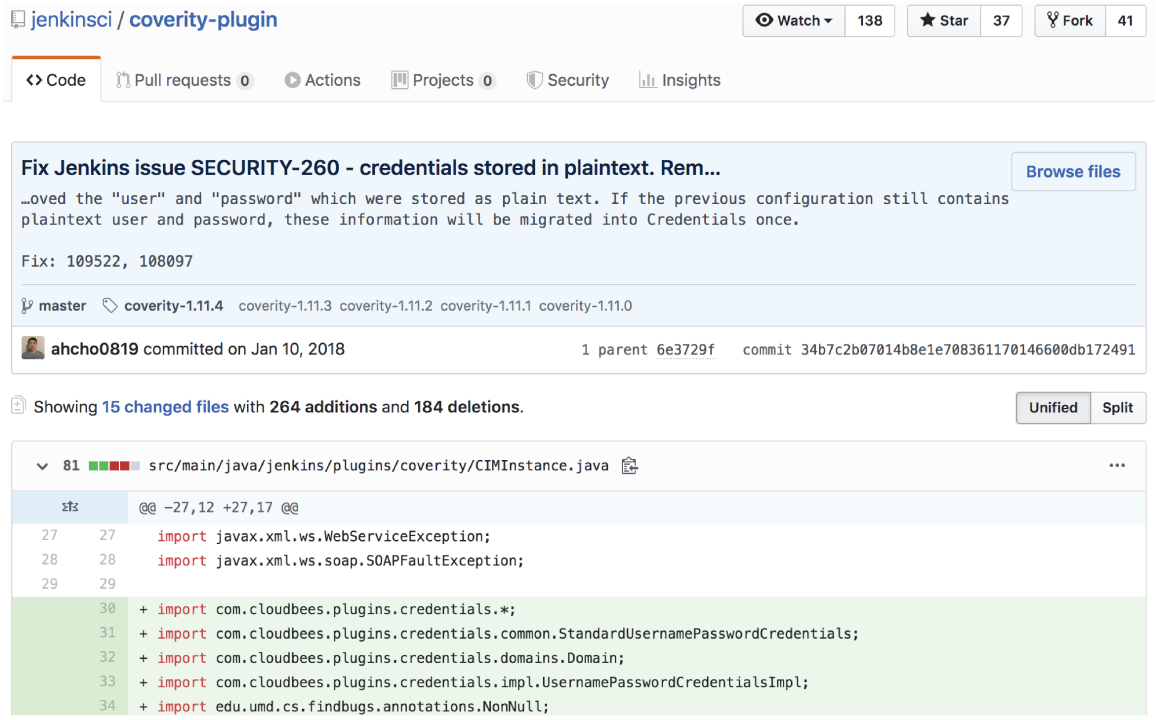


Figure 3.1: An example of a straight forward commit message.

The commit messages in some projects had a unique identifier in the form of capitalized set of characters followed by a set of numbers. We looked up the mentioned identifier whenever the commit message had it and found that in some cases the identifier referred to *an advisory id* related to the project. Looking up the identifier from the commit message we see in Figure 3.1 would lead us to the advisory related to the project, as we see in Figure 3.2

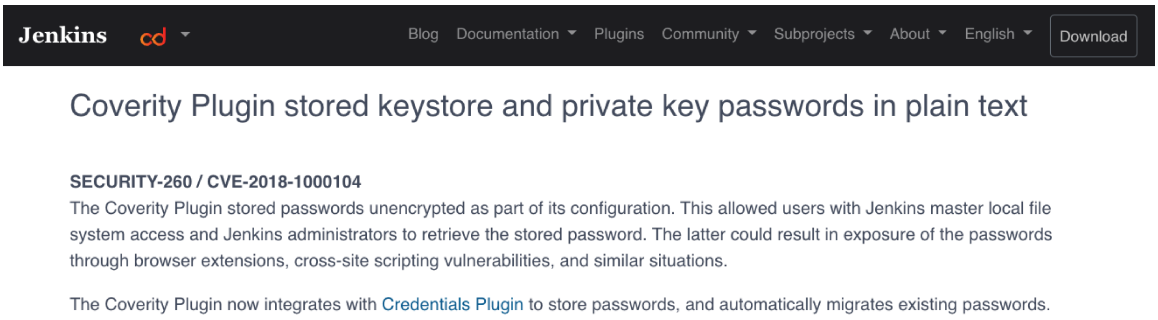


Figure 3.2: An example of project advisory.

Many projects have their own advisories where they publish latest issues and solutions for users of their software in order to avoid being exploitable by the discovered vulnerabilities.

However, the identifier in other cases turned out to be an issue id. Projects that use an external issue tracker have to either put an explicit link to the related issue in the commit message, or to have the issue id in the commit message. As we see in Figure 3.3, the commit message includes an identifier 'UNDERTOW-1190' , searching for the identifier in a search engine reveals the related issue[5], which has more explanation of the problem, as shown in Figure 3.4.

Projects that use GitHub for issue tracking[6] can have a more fluent way to link the commit with the related issue. By simply mentioning the issue number in special syntax[7], GitHub will create a hyper-link in the commit message that links to the issue page as in Figure 3.5.

In all cases of commit messages, regardless of how much information are obtained from them, we inspect the files changed by the commit and look thoroughly at the code changes.

Checking commit's changed files. After looking at the commit message, we look into the code changes. The changes carried in a commit are highlighted on GitHub when the commit link is visited. GitHub shows a *diff* of every changed file in the commit, where the left hand side of the diff shows the file prior to the changes and the right hand side shows

[5]https://issues.jboss.org/browse/UNDERTOW-1190

[6]GitHub is not the only issue tracker used by open source projects, other issue trackers are JIRA and JBOSS are frequently used

[7]https://help.github.com/en/github/writing-on-github/autolinked-references-and-urls#issues-and-pull-requests

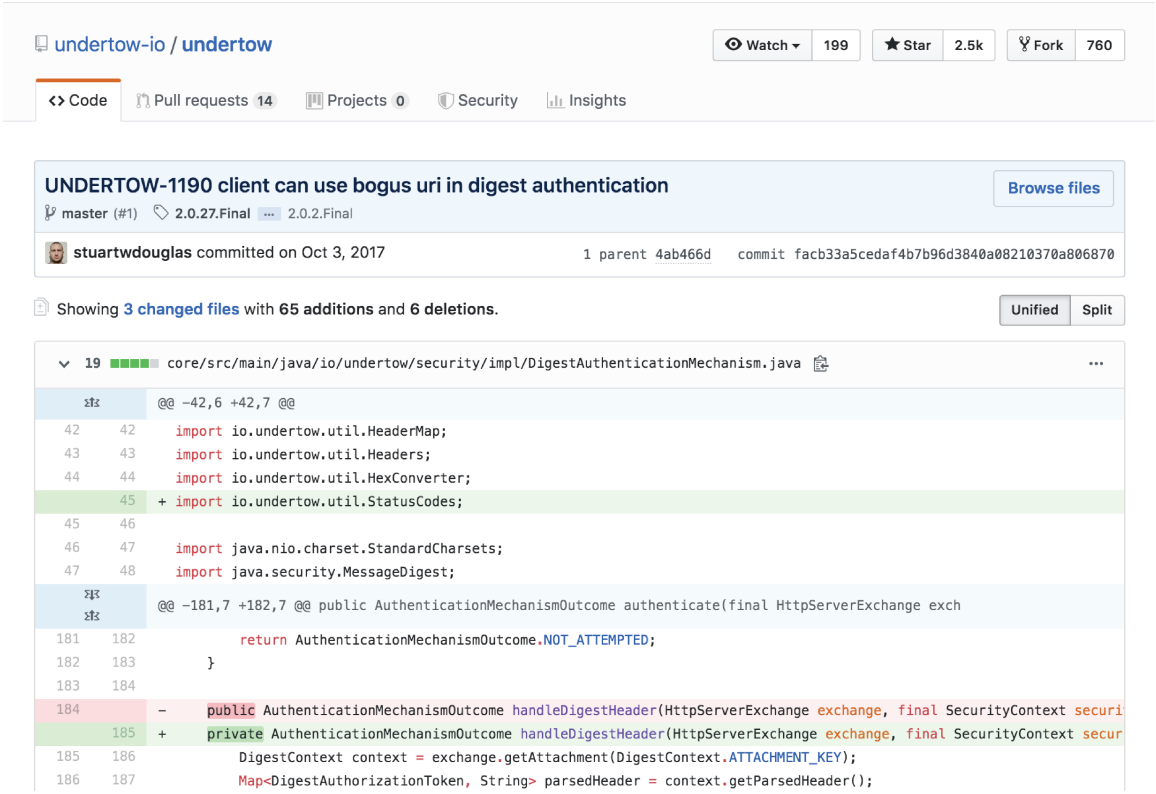


Figure 3.3: An example of identifier in the commit message.

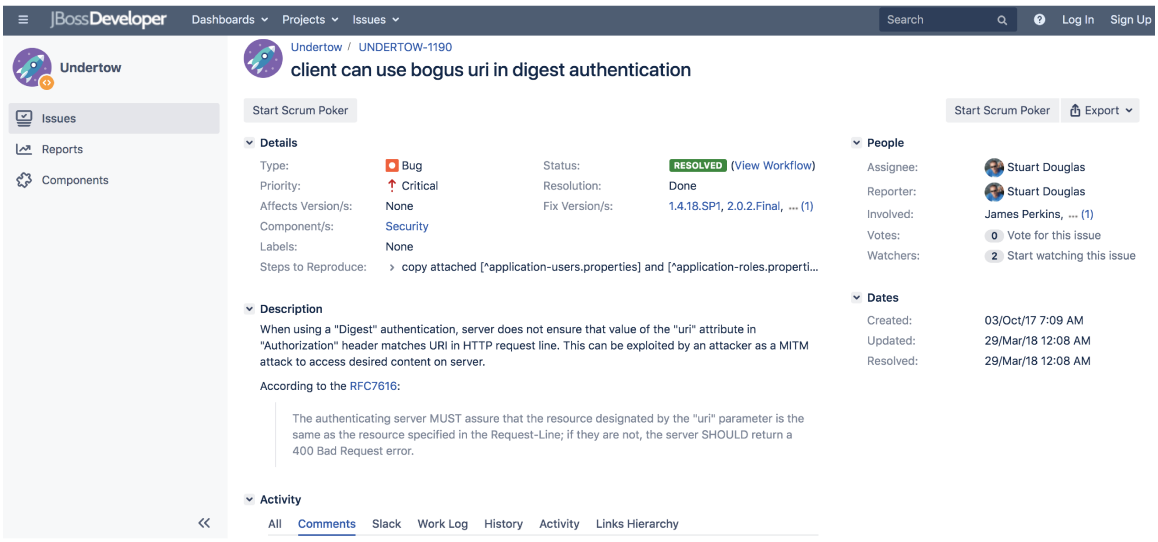


Figure 3.4: An example of related issue from external issue tracker.

the code after the changes. GitHub also shows a few lines before and after the changed code which aims to help in understanding the context of the code.

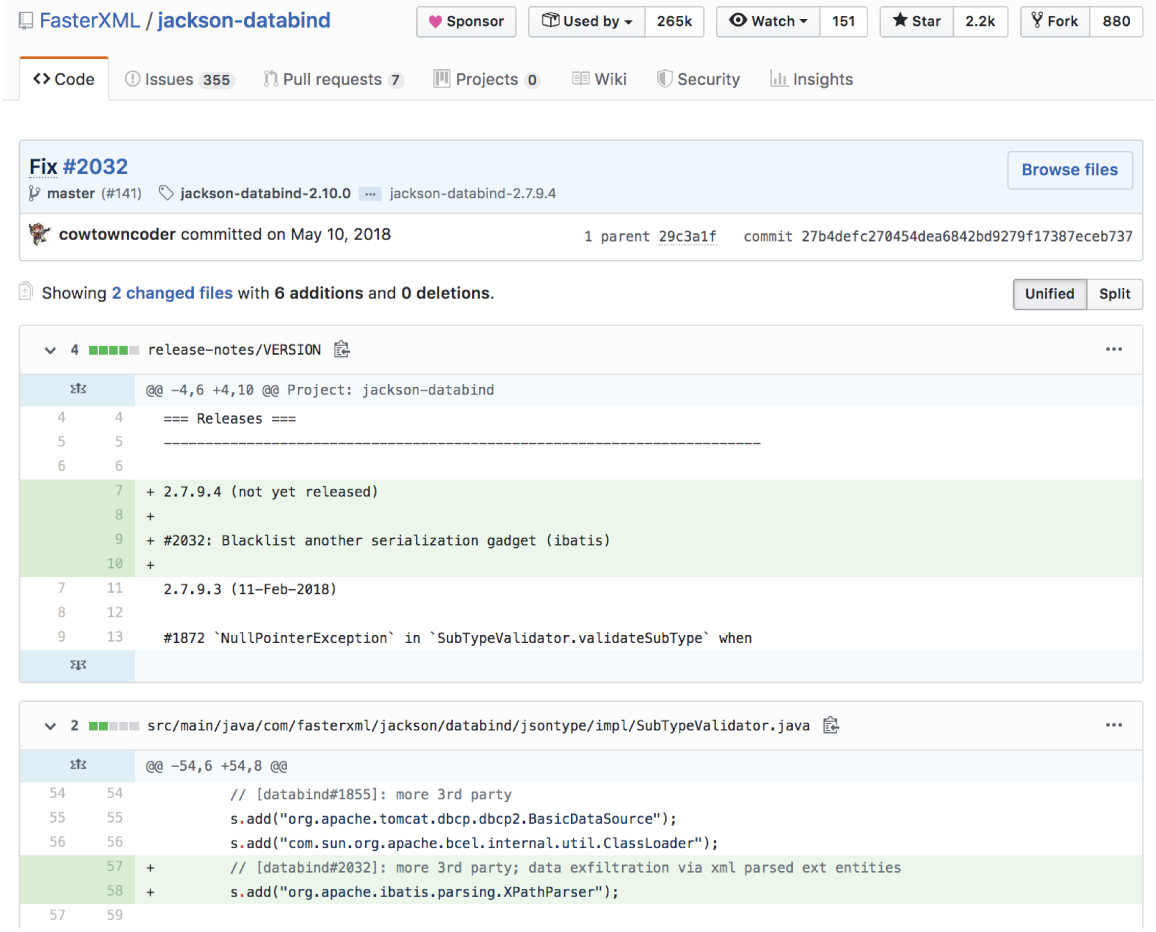


Figure 3.5: An example related issue with hyperlink.

Inspecting the test files in this step is also quite helpful, the added or modified tests essentially practice the changed functionality, which helps in understanding the fix. Parameters of the test cases and the setup done to prepare the test input were also helpful in understanding the changes in the commit. For example, the test case input shown in Figure 3.7 helps understanding the changes in the commit as we will see in listing 4.8.

External Information Sources. We checked the issues and pull requests related to the fixing commits whenever they were linked or mentioned in the commits. The related issues provide information on the security bug and in some cases provide steps on how to reproduce the error, which helps maintainers pinpoint the problem in the code.

Challenges. Here we mention the challenges faced during our manual analysis of the

```
25      /**
26       * Set of well-known "nasty classes", deserialization of which is considered dangerous
27       * and should (and is) prevented by default.
28       */
29      protected final static Set<String> DEFAULT_NO_DESER_CLASS_NAMES;
30      static {
31          Set<String> s = new HashSet<String>();
32          // Courtesy of [https://github.com/kantega/notsoserial]:
33          // (and wrt [databind#1599])
34          s.add("org.apache.commons.collections.functors.InvokerTransformer");
35          s.add("org.apache.commons.collections.functors.InstantiateTransformer");
36          s.add("org.apache.commons.collections4.functors.InvokerTransformer");
37          s.add("org.apache.commons.collections4.functors.InstantiateTransformer");
38          s.add("org.codehaus.groovy.runtime.ConvertedClosure");
39          s.add("org.codehaus.groovy.runtime.MethodClosure");
40          s.add("org.springframework.beans.factory.ObjectFactory");
41          s.add("com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl");
42          s.add("org.apache.xalan.xsltc.trax.TemplatesImpl");
43          // [databind#1680]: may or may not be problem, take no chance
44          s.add("com.sun.rowset.JdbcRowSetImpl");
45          // [databind#1737]; JDK provided
46          s.add("java.util.logging.FileHandler");
47          s.add("java.rmi.server.UnicastRemoteObject");
48          // [databind#1737]; 3rd party
49  //s.add("org.springframework.aop.support.AbstractBeanFactoryPointcutAdvisor"); // deprecated by [databind#1855]
50          s.add("org.springframework.beans.factory.config.PropertyPathFactoryBean");
51
52  // s.add("com.mchange.v2.c3p0.JndiRefForwardingDataSource"); // deprecated by [databind#1931]
53  // s.add("com.mchange.v2.c3p0.WrapperConnectionPoolDataSource"); // - "" -
54          // [databind#1855]: more 3rd party
55          s.add("org.apache.tomcat.dbcp.dbcp2.BasicDataSource");
56          s.add("com.sun.org.apache.bcel.internal.util.ClassLoader");
57          // [databind#2032]: more 3rd party; data exfiltration via xml parsed ext entities
58          s.add("org.apache.ibatis.parsing.XPathParser");
59
60          DEFAULT_NO_DESER_CLASS_NAMES = Collections.unmodifiableSet(s);
61      }
```

Figure 3.6: An example of checking the context of the changes.

commits. Commit messages were not always useful and well formed, many commit messages were short and did not give any information about the change nor link to the related issue. For example, the message shown in Figure 3.3 is not easy to understand on its own. Commit messages were also brief in some cases, containing one or two words. For example, the commit message shown in Figure 3.5 does not mention any information that helps in understanding the commit.

When we inspected the code files, looking at the diff and the context of the change was not enough to understand the code. The default number of lines in the context of the change is not sufficient to understand the fix, and inspecting the code file at the commit is necessary. For example, in the commit shown in Figure 3.5, GitHub shows only a part of the code like: `s.add("org.apache.ibatis.parsing.XPathParser");`

In order to find out what 's' in this context is, we check the code file touched by the

```
71 +     @Parameters(name = "{0}")
72 +     public static Iterable<String> fileNames() {
73 +         return Arrays.asList("file.txt", "../file.txt", "..\\file.txt", "/absolute/file.txt", "c:\\absolute\\file.txt");
74 +     }
```

Figure 3.7: An example of test case parameters.

commit and look for the code surrounding the changes in order to get a better understanding of the context. In this example, we navigate to the Java file `SubTypeValidator`. We find that 's' is a *HashSet* that contains fully qualified names of classes that shouldn't be deserialized as shown in Figure 3.6.

In many cases, we had to consult the documentation on some terms used in the project code, for example: 'Zone ID' in the project UAA is related to the tenant. We also had to look for documentation on how specific functionality works, for example, fixing commits in the Spring OAuth had to deal with specific step in OAuth, which wouldn't be understandable without getting to know the flow of OAuth. In order to provide full picture to the readers of our work, we included brief explanation of the cases in order to help understanding the regualr flow of steps for a specific functionality and how they could go wrong.

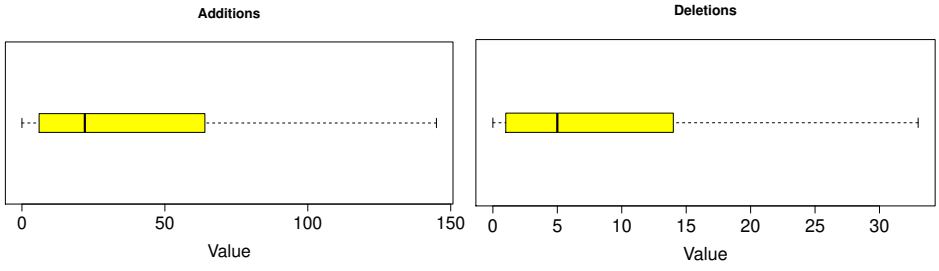
Chapter 4

Results

4.1 Complexity Measures

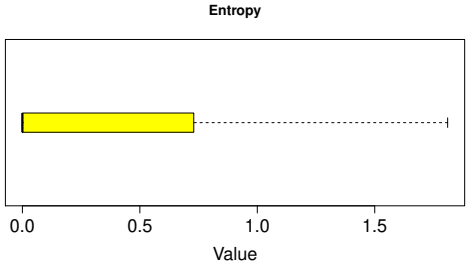
We start with a preliminary analysis of the changes in the commits, we look at four metrics which are the number of lines added per commit, the number of lines deleted per commit, the number of code files changed, and the entropy. Figure 4.1 show boxplots of the studied metrics. We also added the figures with the outliers plotted in the appendix, figure A.1.

We note from the metrics values that the median number of code-files changed per commit is (1) and the median entropy per commit is (0.15), this indicates that the changes typically involve few code files. We also note that the median number of code lines added per commit is (21) and the median number of code lines deleted per commit is (4), which indicates that the number of changes needed in the fixing commit is typically small. Our results are in line with previous results from Li and Paxson (2017), they conducted a large-scale qualitative study of security fixes in order to characterize and compare them to other non-security bug patches. They studied several metrics to conduct their comparison. Among their findings, security commits overall are statistically significantly less complex and smaller than non-security bug patches. For example, the median security commit diff involved 7 LOC compared to 16 LOC for non-security bug fixes. Also, 70% of security patches affected one file, while 55% of non-security bug patches were equivalently localized.

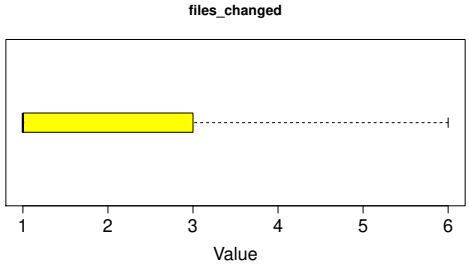


(a) Additions

(b) Deletions



(c) Entropy



(d) Files_Changed

Figure 4.1: Metrics (outliers eliminated)

In summary, the metrics indicate that security fixes are more localized and have lower footprint in their changes than other bug fixes.

In addition to the median values we were curious to inspect why the number of lines added per commit had considerable difference between the 1st and 3rd quartile (as seen from the plots). So we inspected commits that fixed the same security issues but had huge difference in the value of their metrics, and we found they essentially carry the same intended changes.

For example: a fixing commit[1] related to the Deserialization of untrusted data issue has 5 code files changed, 140 added lines and 6 deleted lines, while another similar commit[2] which is related to the same issue has 2 changed code files, 50 added lines and 1 deleted line. Essentially, both of these two commits carry similar changes, which is extending the 'ObjectInputStream' and having a whitelist for classes allowed for deserialization.

Another example related to Zip slip reveals similar findings, a fixing commit[3] has 4 code

[1]https://github.com/apache/logging-log4j2/commit/5dcc19215827db29c993d0305ee2b0d8dd05939d
[2]https://github.com/facebook/buck/commit/8c5500981812564877bd122c0f8fab48d3528ddf
[3]https://github.com/looly/hutool/commit/9f8a801c7b98b75ee681c0988e1a58bcfdc21756

files changed, 65 added lines and 17 deleted lines, and it essentially has the same changes as another commit[4] which has 1 code file changed and 5 added lines. Both commits essentially check whether the path of the extracted file is within the path of the desired extraction directory.

Also for XML External Entity processing issues, the commit[5] has 17 files changed, 88 added lines and 11 deleted lines, while another commit[6] that has similar changes has 1 code file changed, 5 added lines and 6 deleted lines (2 of the deleted lines were blank lines), and yet both commits have similar fixes. Both commits disable XML features like Document Type Definition and External Entities.

We conclude from the examples that the fixing commits related to the same issues may have slightly different metrics based on the nature of the code base in which they are applied, hence, getting a real grip of the issues and the solutions requires more effort than quantitative analysis. This motivated us to do further manual analysis of the fixing commits in order to provide developers with an understanding of the common problems from the community and the practices used to mitigate those problems and secure the underlying applications.

4.2 Categorization

We categorized the studied commits into seven main categories with subcategories in most of them. We explain each category in detail in the next section, table 4.1 shows the main categories with the percentage of cases we found under that category.

As we can see in the table, Validation is the largest category, this is due to the fact that it includes many subcategories related to wide spectrum of issues from Input Validation to Http Headers validation and certificate validation.

Authorization issues also have large the number of cases, this category includes subcategories related to the OAuth 2.0 vulnerabilities and privilege escalation.

[4] https://github.com/apache/hadoop/commit/65e55097da2bb3f2fbdf9ba1946da25fe58bec98
[5] https://github.com/apache/uima-uimaj/commit/39909bf21fd694f4fb792d1de8adc72562ead25e
[6] https://github.com/apache/cayenne/commit/6fc896b65ed871be33dcf453cde924bf73cf83db

Table 4.1: Categories of security issues fixes with their percentages

Category	**Percentage**
Deserialization of untrusted data	10%
Zip slip	9%
XML External Entity Processing	17%
Validation	27%
Authentication & Authorization	26%
Race conditions	5%
Information Exposure	6%

The categories on Deserialization of untrusted data, Zip slip, an XML External Entity Processing contained mainly similar issues, but as we will describe in later sections, we found edge cases in those categories that are worth mentioning. Race conditions category had the least number of cases in our studied commits. As we explained in section 3.1, our studied commits represent a statistically significant number of commits from the studied dataset, we believe the prevalence of the issues to be representative of the issues from the studied dataset.

4.3 Cases Analysis

In the following sections we provide example cases from each category to illustrate the security issues and their solutions. Our goal is to help developers understand those issues and their fixes by providing code snippets from fixing commits in open source projects. We also provide the necessary background to explain the issues and the context of the surrounding code.

4.3.1 Deserialization of untrusted data (10%)

Serialization is the activity of turning an object to a byte stream in order to send it or store it. Deserialization is the opposite activity of turning a byte stream to an object. There are many formats for sending or storing the serialized object, like XML, JSON and YAML. The problem is that if you deserialize untrusted data you might end up with malicious code that makes your system vulnerable to Remote Code Execution. It is advised when

deserializing objects to check the class of the object before actually resolving it.

This is done by extending the `ObjectInputStream` and overriding the `resolveClass` method. When you check the class of the deserialized object, you will check it against a list of allowed classes (which is called whitelisting), or you will check it against a list of disallowed classes (which is called blacklisting). Obviously, whitelisting is more restrictive than blacklisting since it prevents deserialization of classes not specified in the list. The following commit shows how to extend `ObjectInputStream` and override the `resolveClass` method, it also shows the use of whitelisting[7].

```
1  /** A ObjectInputStream that will deserialize only RemoteDaemonicParserState. */
2  public class ParserStateObjectInputStream extends ObjectInputStream {
3  private Set<String> whitelist;
4  public ParserStateObjectInputStream(InputStream inputStream) throws IOException {
5      super(inputStream);
6      whitelist = new HashSet<>();
7      whitelist.add(RemoteDaemonicParserState.class.getName());
8  }
9  @Override
10 protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException,
       ↪ ClassNotFoundException {
11     if (!whitelist.contains(desc.getName())) {
12         throw new InvalidClassException(desc.getName(), "Can't deserialize this
               ↪ class");
13     }
14     return super.resolveClass(desc);
15     }
16 }
```

Listing 4.1: extending the `ObjectInputStream`

[7] https://github.com/facebook/buck/commit/8c5500981812564877bd122c0f8fab48d3528ddf

The application uses serialization to loads/saves state, it only needs to allow one class for deserialization, hence, it whitelists `RemoteDaemonicParserState`.

Specifying the exact class name is not the only way to declare whitelisted or blacklisted classes, it is also possible to specify the package name like [8], which would whitelist (or blacklist) all the classes under that package.

Another approach is to use patterns (wildcard or regex) to specify the classes. And in some cases we found that both whitelisting and blacklisting were used together, with precedence given to the blacklist as in [9].

Void Deserialization

We came across an interesting special case[10] in deserialization, which is the deserialization of **`void`**. **`void`** is used in Java to represent the **`null`** type, it is not possible to create an instance of **`void`** since its constructor is private[11]. In `x-stream` project[12] which is a library to *"Serialize Java objects to XML and back again"*, a commit adds code to prevent the deserialization of **`void`**. The security report mentions that having **`void`** in the input would crash the instance and cause denial of service.

The following check is made in the added code

```
1 if (type == void.class || type == Void.class) {
2     ex = new ConversionException("Type void cannot have an instance");
3 }
```

Listing 4.2: Prevent void Deserialization

4.3.2 Zip Slip (9%)

Zip files are in some cases used as input to the application, the contents of the zip file would be extracted to proper locations on the application server. However, some security

[8] https://github.com/apache/spark/commit/772a9b969aa179150aa216e9efd950e512e9d0b4
[9] https://github.com/apache/camel/commit/adc06a78f04c8d798709a5818104abe5a8ae4b3
[10] https://github.com/x-stream/xstream/commit/6e546ec366419158b1e393211be6d78ab9604ab
[11] https://docs.oracle.com/javase/8/docs/api/java/lang/Void.html
[12] https://github.com/x-stream/xstream

issues may arise if the paths of the files are not checked and validated prior to extraction. The term *"Zip Slip"* was popularized by Snyk for this vulnerability, Snyk has identified this issue in multiple open source application and reported the issue to the maintainers under the name of Zip Slip[13]. It is also common to refer to this problem as zip path traversal.

The issue of zip slip occurs when the contents of the zip file are crafted so that they have relative paths, meaning they have '..' in their path. Upon extracting the files -without having the proper checks in place- this would lead to writing the files in directories that may belong to system files, meaning they would overwrite system files or other important files. This vulnerability may also lead to remote code execution. To protect your application from the zip slip vulnerability, the application code should make sure that *every* file from the zip would end up in the specified extraction directory *before* actually extracting the files. This is done in many fixing commits by checking that the absolute path of each file from the zip actually starts with the absolute path of the extraction directory and throwing an exception otherwise.

The following code snippet from a commit[14] in Hadoop shows the mitigation steps in terms of code:

```
String targetDirPath = toDir.getCanonicalPath() + File.separator;
for(ZipEntry entry = zip.getNextEntry();
    entry != null;
    entry = zip.getNextEntry()) {
    if (!entry.isDirectory()) {
        File file = new File(toDir, entry.getName());
     if (! file.getCanonicalPath().startsWith(targetDirPath))
        throw new IOException("expanding " + entry.getName()
         + " would create file outside of " + toDir);
```

Listing 4.3: Prevent Zip Slip

[13] https://snyk.io/research/zip-slip-vulnerability
[14] https://github.com/apache/hadoop/commit/fc4c20fc3469674cb584a4fb98bac7e3c2277c9

Corrupt Zip Files

Zip file format has a header that holds information about the files, the encryption algorithm along with other information. Zip file utilities read the file as a stream of bytes, the header information is parsed by reading chunks of the input and parsing it (done via offset + length). A problem that may occur in zip utilities when dealing with corrupt files is that the information in the header will not match the content of the file, and this may lead to problems when processing the file. An example of such a case[15] is found in *apache commons-compress* which is a library that defines an API for working with compressed files and archive files, the problem was that one piece of information from the header had a `long` data type and was being compared to an `int` data type in a loop. This didn't cause a problem with uncorrupt zip files, but when a corrupt zip file was used for the input, the value of `long` variable was larger than `MAX_INT` value, and the comparison would never terminate because of `int` overflow. The issue was fixed by having the loop counter datatype changed from `int` to `long` so that the overflow doesn't happen and the loop would terminate even if the zip file is corrupt. This kind of problem might not be common, but the awareness that it makes us pay attention to is that testing against corrupt zip files should not be forgotten. This is not only important to zip file utility maintainers, but also to users of these utilities. Applications that accept zip files as input or store their information in zip files *should not miss the checks for corrupt files in their test cases.*

4.3.3 XML External Entity (XXE) Processing (17%)

XML is a markup language designed to create documents that are human and machine readable. XML is used in wide range of applications like communication protocols and configuration files.

An XML document is made up of elements, which optionally may have attributes. The way the elements are nested defines the XML document structure.

```
1 <?xml version="1.0" encoding="UTF-8"?>
```

[15] https://github.com/apache/commons-compress/commit/2a2f1dc48e22a34ddb72321a4db211da91aa933b

```
<thesis>
<author>Mouafak</author>
<department>Software Engineering</ department >
<year>2019</ year >
</ thesis >
```

Listing 4.4: XML file example

An XML document is well formed if the tags (like <thesis>, <author>) are closed and properly nested, the attribute values must be quoted.

An XML document type definition (DTD) is declared in the DOCTYPE element. The DTD defines the structure, elements and attributes of an XML document, and is used to validate the XML document. The DTD can be internal (within the XML file) or external. Lines 3 to 11 in the example above is for internal DTD, if the DTD is external, those lines would be replaced with `<!DOCTYPE thesis SYSTEM "thesis.dtd">`

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE thesis
[
<!ELEMENT thesis (author,department,year)>
<!ENTITY % chapter_content "section | paragraph" >
<!ENTITY % section_content "subsection | subsubsection" >
<!ENTITY % paragraph_content "subparagraph" >
<!ELEMENT chapter (%chapter_content;)>
<!ENTITY chap1 SYSTEM "chap1.xml">
]>

<thesis>
<author>Mouafak</author>
<department>Software Engineering</department>
```

```
16 <year>2019</year>
17 <chapters>
18 <chapter>&chap1;</chapter>
19 </chapters>
20 </thesis>
```

Listing 4.5: XML document with DTD

XML allows custom entities to be defined within the DTD using a syntax similar to the following example: `<!DOCTYPE mydtd [ <!ENTITY myentity "my entity value"> ]>` The previous definition means that any usage of the entity reference `&myentity;` within the XML document will be replaced with the defined value `"my entity value"`.

XML external entities are a type of custom entity whose definition is located outside of the DTD where they are declared. The declaration of an external entity uses the SYSTEM keyword and must specify a URI from which the value of the entity should be loaded. For example: `<!DOCTYPE mydtd [ <!ENTITY ext SYSTEM "file:///path/to/file"> ]>`

Parameter entities are entities used to group elements and attribute lists in order to refer to them using the parameter entity. Lines 6 to 8 are examples of Parameter Entities. A parameter entity can be distinguished with '%'

XXE attacks XML external entity (XXE) attacks occur when an XML file that contains a reference to an external entity is processed without securing the parser. XXE attacks could lead to exposure of sensitive information if the attacker uses the URI to refer to a local file on the system. It also could lead to denial of service attack if the URI refers to `/dev/random` file which will block the execution until the file has enough data[16].

Disabling XXE attacks OWASP has a document[17] that explains how to prevent XXE attacks for many parsers. They explain that "the safest way to prevent XXE is always to

[16] https://en.wikipedia.org/wiki//dev/random

[17] https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html#general-guidance

disable DTDs (External Entities) completely", which basically means that XML validation will no longer be available to the project. In cases where it is essential to validate XML documents, the solution would be to prevent external DTD and external entities as explained by OWASP: "If it is not possible to disable DTDs completely, then external entities and external document type declarations must be disabled in the way that's **specific to each parser**". A commit[18] from *apache kafka* shows disabling DTD for `SAXParserFactory` and `DocumentBuilderFactory`, basically commits that fix XML issues are more or less using the same syntax for the fixes which is disabling features in the parsers.

```
1  import javax.xml.XMLConstants;
2
3  spf = SAXParserFactory.newInstance();
4  spf.setNamespaceAware(true);
5  spf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
6  spf.setFeature("http://xml.org/sax/features/external-general-entities", false);
7  spf.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
8  spf.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd",
       ↪ false);
9  spf.setXIncludeAware(false);
10
11 dbf = DocumentBuilderFactory.newInstance();
12 dbf.setNamespaceAware(true);
13 dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
14 dbf.setFeature("http://xml.org/sax/features/external-general-entities", false);
15 dbf.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
16 dbf.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd",
       ↪ false);
17 dbf.setXIncludeAware(false);
18 dbf.setExpandEntityReferences(false);
```

[18] https://github.com/apache/karaf/commit/1ffa6d1c4555cab9737d76b49142528b57cfdfc

Listing 4.6: XXE Prevention For The Used Parsers

4.3.4 Validation (27%)

Validation essentially means making sure that the input does not break the application. Many fixes of the security issues were related to validation and verification, and the related issues span a wide spectrum. In this section we report the most common ones with examples from the fixing commits.

Path Traversal

Applications that work with files should validate the paths of the files, whether the application reads or writes a file based on user input, validation of the path (sometime referred to as URI) is essential to avoid directory traversal attacks. Directory traversal attacks happen when the path name contains a relative path "..", which leads to accessing files on the filesystem, or writing files in unallowed paths that may lead to overwriting system files.

We have spotted this issue in multiple projects. Validation is done by checking that the path doesn't contain relative path, and may check for slashes in the path (forward and backward slashes if the app has both windows and unix support). The following code snippet from a fixing commit[19] in *apache lucene-solr* adds validation of the `filename`

```
1 // Throw exception on directory traversal attempts
2 protected String validateFilenameOrError(String filename) {
3     if (filename != null) {
4     Path filePath = Paths.get(filename);
5     filePath.forEach(subpath -> {
6     if ("..".equals(subpath.toString())) {
7         throw new SolrException(ErrorCode.FORBIDDEN, "File name cannot contain ..");
```

[19] https://github.com/apache/lucene-solr/commit/3a4f885b18bc963a8326c752bd229497908f1db

```
8       }
9       });
10      if (filePath.isAbsolute()) {
11          throw new SolrException(ErrorCode.FORBIDDEN, "File name must be relative");
12      }
13      return filename;
14      } else return null;
15  }
```

Listing 4.7: File Path Validation

A fixing commit[20] in *apache camel* also manages the filepath of the attached files. The essential change in the commit is to call `FileUtil.stripPath` method for file names.

```
String fileName = FileUtil.stripPath(part.getFileName());
```

Looking at the added test `shouldSanitizeAttachmentFileNames` gives a hint on what the `stripPath` method should handle

```
1  public static Iterable<String> fileNames() {
2  return Arrays.asList("file.txt", "../file.txt", "..\\file.txt", "/absolute/file.txt",
       ↪ "c:\\absolute\\file.txt");
```

Listing 4.8: Test input in `shouldSanitizeAttachmentFileNames` test case

We look up the body of the method `stripPath` at the same fixing commit[21] (although it was not part of the change):

```
1  /**
2  * Strips any leading paths
3  */
4  public static String stripPath(String name) {
5  if (name == null) {
```

[20] https://github.com/apache/camel/commit/63c7c080de4d18f9ceb25843508710df2c2c6d4

[21] https://github.com/apache/camel/blob/63c7c080de4d18f9ceb25843508710df2c2c6d40/camel-core/src/main/java/org/apache/camel/util/FileUtil.java

```
6       return null;
7   }
8   int posUnix = name.lastIndexOf('/');
9   int posWin = name.lastIndexOf('\\');
10  int pos = Math.max(posUnix, posWin);

12  if (pos != -1) {
13      return name.substring(pos + 1);
14  }
15  return name;
16  }
```

Listing 4.9: Inspecting the code of related file `FileUtil.java`

Also the following commit[22] in *Spring Framework* deals with the same issue, it adds a method to check for invalid paths, in this commit a fix for windows filepath checking was added.

```
1   private boolean isInvalidEncodedPath(String resourcePath) {
2   if (resourcePath.contains("%")) {
3       // Use URLDecoder (vs UriUtils) to preserve potentially decoded UTF-8 chars...
4       try {
5           String decodedPath = URLDecoder.decode(resourcePath, "UTF-8");
6           return (decodedPath.contains("../") || decodedPath.contains(".."));
7       }
8       catch (UnsupportedEncodingException ex) {
9       // Should never happen...
10      }
11  }
12  return false;
13  }
```

[22] https://github.com/spring-projects/spring-framework/commit/13356a7ee2240f740737c5c83bdccdacc30603a

Listing 4.10: File Path Validation

Improper Validation of Certificate with Host Mismatch

Establishing a secure connection in client-server communications involves using the TLS (Transport Layer Security) protocol, the protocol involves exchanging a set of messages between the client and the server to acquire pieces of information necessary to establish the secure connection. TLS is used when creating a secure http or websocket connection, and takes place just after the TCP connection has been established.

Authenticating the identity of the server and its certificate is one of steps done in the handshake. To verify the certificate, two steps are involved, one of them is to make sure it is valid (through security chain), and the other one is to make sure it matches the server name. If the later part is overlooked, a Man in the middle attack may be possible if a malicious host (who has the valid certificate) provides data.

Many security fixing commits had to deal with the later step, which is verifying that the hostname of the server matches the certificate.

The following code snippet from *apache/activemq*[23] provides an example on how to properly do hostname verification for certificate validation:

```
1 import javax.net.ssl.SSLParameters;


3 if (verifyHostName) {
4     SSLParameters sslParams = new SSLParameters();
5     sslParams.setEndpointIdentificationAlgorithm("HTTPS");
6     sslEngine.setSSLParameters(sslParams);
7 }
```

Listing 4.11: Hostname Verification

[23] https://github.com/apache/activemq/commit/bde7097fb8173cf871827df7811b3865679b963d

The lack of hostname validation was an issue in many other projects [24], [25], [26]

HTTP Headers

Clients communicating with servers through the http protocol can make use of headers. Headers are key-value pairs which contain additional information sent with the http request or the response.

Header value validation There are predefined headers which have pre-defined roles. It is also possible to set a new header and its value. The `Range` header[27] is used to specify the part(s) of the document that the server should return. It is possible to specify multiple parts of the document to be returned in a single request. However, a security fixing commit from *Spring framework* shows that it is a bad practice to skip validation of the requested ranges. The vulnerability report[28] indicates that a denial of service attack is possible due to the lack of validation on the requested ranges.

The fixing code sets a limit on the number of requested ranges (arbitrary number), and does additional checks to verify that the ranges don't overlap and don't exceed the length of the requested resource.

The following code snippet from *spring-framework* project[29] shows the changes in terms of code:

```
1 private static final int MAX_RANGES = 100;
2 //...
3 Assert.isTrue(tokens.length <= MAX_RANGES, () -> "Too many ranges " + tokens.length);
4 //...
5 Assert.isTrue(total < length, () -> "The sum of all ranges (" + total + ") " +
      ↪ "should be less than the resource length (" + length + ")");
```

[24] https://github.com/apache/tomcat/commit/2835bb4e030c1c741ed0847bb3b9c3822e4fbc8a
[25] https://github.com/apache/activemq/commit/bde7097fb8173cf871827df7811b3865679b963d
[26] https://github.com/apache/qpid-proton-j/commit/0cb8ca03cec42120dcfc434561592d89a89a805e
[27] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Range
[28] https://nvd.nist.gov/vuln/detail/CVE-2018-15756
[29] https://github.com/spring-projects/spring-framework/commit/c8e320019ffe7298fc4cbeeb194b2bfd6389b6d

Listing 4.12: `Range` Header Validation

CRLF Injection CRLF are the Carriage Return and Line Feed characters, also referred to as ‘\r \n‘. These characters are used in http communication to indicate the end of one header and the start of the next header (single CRLF) or to indicate the end of http headers and the start of the website content (double CRLF).
CRLF injections could lead to serious attacks based on the injected header or content. They happen if the application input is used in the http communications directly. A commit[30] in *eclipse-vertx/vert.x* adds *http header CRLF validation* by checking header names and values for occurrences of ‘\r‘ or ‘\n‘ characters.

The following code snippet illustrates the checks for the characters

```
1 public static void validateHeader(CharSequence value) {
2     for (int i = 0;i < value.length();i++) {
3     char c = value.charAt(i);
4     if (c == '\r' || c == '\n') {
5         throw new IllegalArgumentException("Illegal header character: " + ((int)c));
6     }
7 }
```

Listing 4.13: CRLF Validation

4.3.5 Authentication and Authorization (26%)

Authentication an Authorization are common terms in access management, authentication refers to validating the user identity while authorization has to deal with permissions to access specific resources. Authentication takes place before authorization and involves providing credentials to validate the user identity.

[30] https://github.com/eclipse-vertx/vert.x/commit/1bb6445226c39a95e7d07ce3caaf56828e8aab72

Authorization Vulnerability

Authorization manages user access to certain functionality or resources in the system. Improper authorization means that the check done for restricting access to these resources is not done properly. This leads to allowing users access to resources they shouldn't be able to access.

An issue in *Jenkins favorite plugin* allowed a user to toggle the favorites of another user, this was possible because the `user` was being passed as a parameter to the toggle favorite action and without proper checks in the code. The fixing commit[31] resolved the issue by getting the user whose favorites to toggle from the logged in user and not from url parameters.

The code in the commit changes the code to get a user from '`User user = jenkins.getUser(userName);`' to '`User user = User.current();`'

Another example[32] from *CloudFoundry User Account and Authentication Server (UAA)* allowed privileged users in one zone to perform a password reset for users in a different zone. *UAA* is a multi tenant identity management service and the concept of zone id corresponds to tenants. Zone ids are logically bound to entities managed by UAA[33].

The fixing commit added zone id to `String code` which is the authorization code obtained from the authorization server, it expires at specific timestamp.

```
1 default String zonifyCode(String code) {
2     return code + "[zone[" + IdentityZoneHolder.get().getId()+"]]";
3 }


5 default String extractCode(String zoneCode) {
6     int endIndex = zoneCode.indexOf("[zone[" + IdentityZoneHolder.get().getId()+"]]");
7     if (endIndex<0) {
8         return zoneCode;
9     }
```

[31] https://github.com/jenkinsci/favorite-plugin/commit/b6359532fe085d9ea6b7894e997e797806480777
[32] https://github.com/cloudfoundry/uaa/commit/bbf6751bc0d87c4a3aaf21b54e26ce328ab998b
[33] https://docs.cloudfoundry.org/uaa/uaa-concepts.html#iz

```
10      return zoneCode.substring(0, endIndex);
11  }
```

Listing 4.14: Add Zone Id to Authorization Code

OAuth 2.0 vulnerabilities

OAuth is an open standard for authorization, it is used to provide client applications with tokens that allow them to access user information (privileged user resources) on a service without knowing their credentials.

There are two versions of OAuth. OAuth 2.0[34] is the one commonly used. OAuth 2.0 specifies a set of steps required for a client to get a token. There are different flows for OAuth 2.0, they differ based on the type of the client application and how much trust it has[35].

A typical flow of interactions to authorize a client web application would go as follows:

(1) User visits the web application and initiates the authorization process.

(2) Web application redirects user to the authorization server.

(3) User approves access.

(4) Authorization server returns authorization code to the web application.

(5) Web application exchanges the authorization code for access token.

In future communications the web application would use the access token to get access to user data.

However, privilege escalation could happen during those steps if proper validation is missing. We found two cases in Spring and in CloudFoundry UAA where the fixing commit added validation for the authorization request on approval. The validation is done by comparing the fields of the Authorization Request before and after the user approves the

[34] https://tools.ietf.org/html/rfc6749
[35] https://auth0.com/docs/api-auth/which-oauth-flow-to-use

request. Typically those fields should not change in the comparison. The reason for such validation is that the lack of it would allow a malicious user to modify the authorization request sent with the approval, which would make it possible for privilege escalation.

The following fixing commit[36] from *spring-security-oauth* project, modifies the `AuthorizationEndpoint` class by adding a method called `isAuthorizationRequestModified`.

The commit modifies the class method `authorize` by adding code to save the Authorization Request:

```
// Store authorizationRequest AND an immutable Map of authorizationRequest in session
// which will be used to validate against in approveOrDeny()
model.put(AUTHORIZATION_REQUEST_ATTR_NAME, authorizationRequest);
model.put(ORIGINAL_AUTHORIZATION_REQUEST_ATTR_NAME,
    ↪ unmodifiableMap(authorizationRequest));
```

The `unmodifiableMap(authorizationRequest)` is a new method defined in the class with no access modifier, meaning that it is class and package accessible[37], it basically creates an `unmodifiableSet`[38] for the attributes of the authorization request. The method body shows the saved attributes:

```
Map<String, Object> unmodifiableMap(AuthorizationRequest authorizationRequest) {
    Map<String, Object> authorizationRequestMap = new HashMap<String, Object>();


    authorizationRequestMap.put(OAuth2Utils.CLIENT_ID,
        ↪ authorizationRequest.getClientId());
    authorizationRequestMap.put(OAuth2Utils.STATE, authorizationRequest.getState());
    authorizationRequestMap.put(OAuth2Utils.REDIRECT_URI,
        ↪ authorizationRequest.getRedirectUri());
    if (authorizationRequest.getResponseTypes() != null) {
    authorizationRequestMap.put(OAuth2Utils.RESPONSE_TYPE,
```

[36] https://github.com/spring-projects/spring-security-oauth/commit/623776689fdcc8047f5a908c71f348e1f172a97
[37] https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
[38] https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableSet(java.util.Set)

```
9       Collections.unmodifiableSet(new
            ↪ HashSet<String>(authorizationRequest.getResponseTypes()));
10      }
11      if (authorizationRequest.getScope() != null) {
12          authorizationRequestMap.put(OAuth2Utils.SCOPE,
13              Collections.unmodifiableSet(new
                    ↪ HashSet<String>(authorizationRequest.getScope()));
14      }
15      authorizationRequestMap.put("approved", authorizationRequest.isApproved());
16      if (authorizationRequest.getResourceIds() != null) {
17          authorizationRequestMap.put("resourceIds",
18              Collections.unmodifiableSet(new
                    ↪ HashSet<String>(authorizationRequest.getResourceIds()));
19      }
20      if (authorizationRequest.getAuthorities() != null) {
21          authorizationRequestMap.put("authorities",
22              Collections.unmodifiableSet(new
                    ↪ HashSet<GrantedAuthority>(authorizationRequest.getAuthorities()));
23      }


25      return Collections.unmodifiableMap(authorizationRequestMap);
26  }
```

Listing 4.15: Storing `authorizationRequest` In Immutable Dataset

The class method `approveOrDeny` is modified to add the check for modification of the authorization request, and would throw an exception if the fields don't match:

```
1   // Check to ensure the Authorization Request was not modified during the user
        ↪ approval step
2   @SuppressWarnings("unchecked")
```

```
3 Map<String, Object> originalAuthorizationRequest = (Map<String, Object>)
      ↪ model.get(ORIGINAL_AUTHORIZATION_REQUEST_ATTR_NAME);
4 if (isAuthorizationRequestModified(authorizationRequest,
      ↪ originalAuthorizationRequest)) {
5     throw new InvalidRequestException("Changes were detected from the original
          ↪ authorization request.");
6 }
```

Listing 4.16: Verifying `authorizationRequest` Was Not Changed

The annotation `@SuppressWarnings("unchecked")` is to suppress compiler warnings for the cast (getting the authorization request map of attributes from the model).

We found a very similar fixing commit[39] in CloudFoundary UAA. It carries on the same changes as the ones mentioned in the previous fixing commit.

Cross-Site Request Forgery Prevention

A CSRF token is used as means to prevent CSRF attacks, the token is generated on the server and sent to the client to be used in subsequent communications.

CSRF (Cross-site request forgery) attacks are when an attacker illudes the user to make requests that he didn't intend (like clicking a link to change the user password with pre set parameters by the attacker). CSRF tokens mitigate this by adding a piece of encrypted information that is required with each request, which would make constructing a valid request impossible without knowing the CSRF token.

A CSRF token should be generated on the server and saved against the user session. It should be sent to the user in a hidden input field (or via headers in ajax communications). And it should be validated on each subsequent request.

The following fixing commit[40] from *apache cxf* adds CSRF support to the webapp, we will demonstrate the practice using code snippets from the commit.

[39] https://github.com/cloudfoundry/uaa/commit/3f0730a015d10166de23b7e036743c185f0576a6
[40] https://github.com/apache/cxf-fediz/commit/c68e4820816c19241568f4a8fe8600bffb0243cd

The following snippet demonstrates the creation of the token and saving it in the user session on the server.

```
public static String getCSRFToken(HttpServletRequest request, boolean create) {
    if (request != null && request.getSession() != null) {
        // Return an existing token first
        String savedToken = (String)request.getSession().getAttribute(CSRF_TOKEN);
        if (savedToken != null) {
            return savedToken;
    }

    // If no existing token then create a new one, save it, and return it
    if (create) {
        String token =
            ↪ StringUtils.toHexString(CryptoUtils.generateSecureRandomBytes(16));
        request.getSession().setAttribute(CSRF_TOKEN, token);
        return token;
    }
}
```

Listing 4.17: Adding CSRF support to the code file

The following snippet demonstrates adding a hidden field in the html form of the client app for the CSRF token.

```
<div class="form-line">
    <input type="hidden" value="<%=token%>" name="client_csrfToken" />
</div>
```

Listing 4.18: Adding CSRF Token To The Form

The following code snippets shows adding the check for the CSRF token in the public service methods, and also shows the method for checking the CSRF token against the one

stored in the session.

```
1  public RegisteredClients removeClient(@PathParam("id") String id,
       ↪ @FormParam("client_csrfToken") String csrfToken) {
2  // CSRF
3  checkCSRFToken(csrfToken);

5  public Client resetClient(@PathParam("id") String id,
       ↪ @FormParam("client_csrfToken") String csrfToken) {
6  // CSRF
7  checkCSRFToken(csrfToken);

9  // more methods here ...

11 private void checkCSRFToken(String csrfToken) {
12     // CSRF
13     Message message = PhaseInterceptorChain.getCurrentMessage();
14     HttpServletRequest httpRequest =
15         (HttpServletRequest) message.get(AbstractHTTPDestination.HTTP_REQUEST);
16     String savedToken = CSRFUtils.getCSRFToken(httpRequest, false);
17     if (StringUtils.isEmpty(csrfToken) || StringUtils.isEmpty(savedToken)
18         || !savedToken.equals(csrfToken)) {
19         throwInvalidRegistrationException("Invalid CSRF Token");
20      }
21  }
```

Listing 4.19: Adding CSRF Validation To The Routes

4.3.6 Race Conditions (5%)

A Race condition basically means that two threads are trying to access the same data, this results in having wrong data in the threads or causing errors because of trying to

use the same resource. Multithreading is used in applications to maximize utilization of the computing resources and/or to avoid blocking the application due to long operations. Applications that make use of multithreading should organize threads access to shared resources. This is usually done by using a synchronized block[41] to surround the critical data. However, thread synchronization is also possible by using accessible fields that represent the processing states of the threads. This would make it possible for a thread to wait until another thread transitions into a valid state.

A fixing commit[42] from *apache directory ldap api* fixed a race condition that compromised the application security. The problem was that a thread could use a connection that should be secured before establishing the TLS connection, i.e. before the ssl connection setup gets finishes. Which would result in leaking information contained in the request.

The following code was added to the `connect` method in `LdapNetworkConnection` class, the code added a check to verify the connection is secure before using the connection, and the check actually waits for the given `timeout` before returning the value of whether the connection is secured or not (calling `handshakeFuture.get` method) [43], [44].

```
1  if ( config.isUseSsl() )
2  {
3      try
4      {
5          boolean isSecured = handshakeFuture.get( timeout, TimeUnit.MILLISECONDS );
6
7          if ( !isSecured )
8          {
9              throw new LdapOperationException( ResultCodeEnum.OTHER, I18n.err(
                   ↪ I18n.ERR_4100_TLS_HANDSHAKE_ERROR ) );
10         }
```

[41] https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html
[42] https://github.com/apache/directory-ldap-api/commit/075b70a733d7af150b3d85684149ff5f029f7fd
[43] https://directory.apache.org/api/gen-docs/latest/apidocs/org/apache/directory/ldap/client/api/future/HandshakeFuture.html#get-long-java.util.concurrent.TimeUnit-
[44] https://directory.apache.org/api/gen-docs/latest/apidocs/src-html/org/apache/directory/ldap/client/api/future/HandshakeFuture.html

```
11      }
12      catch ( Exception e )
13      {
14          String msg = "Failed to initialize the SSL context";
15          LOG.error( msg, e );
16          throw new LdapException( msg, e );
17      }
18  }
```

Listing 4.20: Thread Synchronization Example

4.3.7 Information Exposure (6%)

Critical information used in the application code should not be stored in unsecure format, an example would be user credentials and session information. Also information about the application structure or similar kind of information should not be accessible by means of bad application input through exceptions and stack traces.

Insufficiently Protected Credentials

Storing sensitive information in a secure form would be considered an intuitive practice by developers nowadays, it adds an extra layer of protection so that even if the sensitive information ends up in the hands of an attacker it would be impossible for them to make use of it.

There are two ways to store sensitive information in a secure way, one is by using salting and hashing, and the other is by using encryption keys. Hashing is a one-way method, meaning that it is only possible to get the hash of the input and compare it to previously store hash, while encryption is a two-way method which involves having the encryption keys stored on the server. Salting is an essential practice when using hashing, it essentially means concatenating additional values to the original value before hashing, which will make dictionary attacks impossible to carry on especially if the salt value is not constant.

We found cases related to storing sensitive information in plaintext in some of the fixing commits. This issue is not only still occurring in open source software, but also in recent reports from big tech companies like Facebook and Google, both of which have recent reports on storing user passwords in plaintext. A recent blog[45] from Facebook mentions that *"As part of a routine security review in January, we found that some user passwords were being stored in a readable format within our internal data storage systems"*.
Google also has a recent blog[46] about a similar issue *"we recently notified a subset of our enterprise G Suite customers that some passwords were stored in our encrypted internal systems unhashed"*.

There are multiple reports on plaintext storage of passwords in Jenkins plugins like *Build publisher plugin*, *Coverity Plugin*, *AWS CodePipeline Plugin*, and *AWS CodeDeploy Plugin*. The following code is from a commit[47] fixing plaintext storage for *Build publisher plugin* The changed code imports `hudson.util.Secret` and uses it for storing password information instead of `String`.
Jenkins uses encryption (which uses two way functions to encrypt and decrypt using keys as we mentioned before), and that's why we see in the code that it is possible to get a password from a secret.

```
1 this.secret = Secret.fromString(password);
```

Listing 4.21: Secure Storage Of The Password in `Secret`

And to get the plaintext password from the secret:

```
1 /*package*/ String getPassword() {
2 return secret.getPlainText();
3 }
```

Listing 4.22: Password Retreival From `Secret`

[45] https://newsroom.fb.com/news/2019/03/keeping-passwords-secure/
[46] https://cloud.google.com/blog/products/g-suite/notifying-administrators-about-unhashed-password-storage
[47] https://github.com/jenkinsci/build-publisher-plugin/commit/7f80f0d7c9cd96a2d660eeb8b695297bef064059

Session identification values exposure

Logging is an essential practice in software development, the log is an invaluable source of information to understand/debug issues that happen in the live environment after deploying the application. The issue is that developers might log important information that shouldn't be exposed in the logs. For example, the session id is a token generated on the server and stored on the client by means of a cookie. It is used in later communications to identify the user. Developers may log the session id to track the user interactions with the application during a session. However, if an attacker gets access to live logs, he could use the session id to impersonate active users.

A fixing commit[48] in *Cloud Foundry UAA* **removed** session id logging, the issue was that the class `UaaAuthenticationDetails` had the following code in its `toString()` method

```
1 if (sessionId != null) {
2     if (sb.length() > 0) {
3     sb.append(", ");
4     }
5     sb.append("sessionId=").append(sessionId);
6 }
```

Listing 4.23: Removed `Session Id` From `toString` Method

Which means when saving the class in the audit logs, the session id would be saved.

However, if tracking the user interactions with the application is essential, a better practice is hash the session id before logging it. This would still make it possible to track user interactions during the session without revealing the session id to the naked eye.

Timing attacks

Storing sensitive information in a hashed format is not enough, practices used for comparison of the hashed values are also quite important. The comparison algorithm used to compare the hashed values should not reveal how much of the input matches the stored

[48] https://github.com/cloudfoundry/uaa/commit/a61bfabbad22f646ecf1f00016b448b26a60daf

value. The revealed information can be used in *Timing Attacks*. Timing Attacks are possible if comparison of the stored hash is done byte-by-byte and the result is returned on the first mismatch. This would allow the attacker to measure the time it takes to compare the provided value with the stored value (from multiple attempts), and finally craft an input that matches the stored value. The workaround for timing attacks is to use constant length comparison algorithms that doesn't return on the first mismatch, but rather they should compare all bytes of the input value and the stored value and return the final true/false comparison result based on that.

The following commit[49] from *eclipse/jetty.project* fixes the timing attack by comparing all bytes from the byte arrays `b1` and `b2` and accumulating the results of the comparison in the boolean variable `result`.

```
1 int l1 = b1.length;
2 int l2 = b2.length;
3 if (l1 != l2)
4     result = false;
5 int l = Math.min(l1, l2);
6 for (int i = 0; i < l; ++i)
7    result \&= b1[i] == b2[i];
8 return result;
```

Listing 4.24: Timing Attack Mitigation

Information Exposure Through an Error/Error Message

Exceptions are used to handle runtime errors that may occur in unexpected situations like unexpected input. Exception handling is essential to avoid crashing the application when such errors happen and to make sure to recover the application state (like rolling back a business transaction).

Typically, code that *could* lead to an exception is surrounded by a *try-catch* block, these

[49] https://github.com/eclipse/jetty.project/commit/2baa1abe4b1c380a30deacca1ed367466a1a62ea

blocks could have multiple *catch* statements ordered from the most specific exception class (appearing first) to the most generic one. The code in a catch statement should be able to recover from the specified expected exception. When there are resources that need to be closed, a *finally* statement allows adding code that would be always executed when an exception occurs.

Exceptions have stack trace and error message. A stack trace is the method call stack from the point where the exception occurs to the first method invoked in the thread. The stack trace should be used internally in the application logs to facilitate inspecting the exception and should not be output to the user of the application.

Failing to catch the exception would result not only in crashing the application but also in revealing the stack trace to the user, such revealed information could be used by an attacker to compromise the system, especially since the input that could have cause the exception may be attempted by an attacker.

An example for such a case is from *eclipse/jetty.project* where an intentional bad query sent to the application might reveal the path to the resource directory of the DefaultServlet[50].

The workaround was to catch the most general exception that may arise, and to create and throw a new exception that reports to the user that the input was invalid[51].

```
1 catch (Throwable t)
2 {
3 // Any error has potential to reveal fully qualified path
4 throw (InvalidPathException) new InvalidPathException(pathInContext, "Invalid
      ↪ PathInContext").initCause(t);
5 }
```

Listing 4.25: Catch and Throw Exception

As seen in the above code snippet, it is possible to supply the exception with information that could be used to report the error to the user, in the previous snippet the additional

[50] https://nvd.nist.gov/vuln/detail/CVE-2018-12536
[51] https://github.com/eclipse/jetty.project/commit/53e8bc2a636707e896fd106fbee3596823c2cdc

info was the actual `pathInContext` info and a string `"Invalid PathInContext"`. Exception messages may be formatted and reported to the user of the application, however in localized applications, these messages are also localized and reported to the user in their preferred supported language. However, when providing such functionality, developers should make sure that errors that don't have a localization don't end up reporting the full exception to the user.

An example for such a case is from *apache struts*, where failing to find a localized error message for an exception would result in reporting the whole exception info to the end user. The fix was to use the default error key for an exception if no localized value was found[52].

```
1  protected String buildErrorMessage(Throwable e, Object[] args) {
2  String errorKey = "struts.messages.upload.error." + e.getClass().getSimpleName();
3  if (LOG.isDebugEnabled()) {
4  LOG.debug("Preparing error message for key: [#0]", errorKey);
5  }
6    if (LocalizedTextUtil.findText(this.getClass(), errorKey,
7       getLocale(), null, new Object[0]) == null) {
8    return LocalizedTextUtil.findText(this.getClass(), "struts.messages.error.uploading",
9       defaultLocale, null, new Object[] {
10            e.getMessage() });
11   } else {
12   return LocalizedTextUtil.findText(this.getClass(), errorKey,
13       defaultLocale, null, args);
14   }
15 }
```

Listing 4.26: Using Default Error Key If No Localized Value Is Found

Table 4.2: Cross reference the studied cases with OWASP

Vulnerability	**Mentioned by OWASP**	**OWASP Link**
Deserialization of untrusted data	✓	C1 (2017)
Void deserialization	-	
Zip slip	✓	C2 (2017)
Corrupt zip file	-	
XML External Entity Processing	✓	C3 (2017)
Path traversal	✓	C4 (2015)
Improper Validation of Certificate with Host Mismatch	✓	C5 (2017)
Header validation (Data validation)	✓	C6 (2016)
CRLF injection	✓	C7 (2018)
Authorization vulnerability	✓	C8 (2008)
OAuth 2.0 vulnerabilities	-	
Cross-Site Request Forgery prevention	✓	C9 (2017)
Race conditions	✓	C10 (2009)
Insufficiently Protected Credentials	✓	C11 (2009)
Session identification values exposure	✓	C12 (2015)
Timing attacks	✓	C13 (2017)
Information Exposure Through an Error/Error Message	✓	C14 (2015)

4.4 Cross Referencing with OWASP

In this section we report the results of cross referencing the cases we studied with OWASP. Table 4.2 shows the vulnerabilities names in our study, whether or not they are in OWASP, and the link to their page on OWASP. Three of the cases from our study are not mentioned by OWASP, which are *void deserialization*, *corrupt zip file*, and *OAuth 2.0 vulnerabilities*. However, for the later one, the *OAuth 2.0 Security Best Practice* mentions validation as a mitigation for some of the described attacks.

[52]https://github.com/apache/struts/commit/352306493971e7d5a756d61780d57a76eb1f519a

Chapter 5

Implications, Summary, and Future Work

Implications. Some of the security vulnerability issues we discussed have straight forward solutions even though they were quite common, for example the deserialization of untrusted data and XML External Entity processing issues have straightforward solutions that are more likely to follow a specific structure for the solution, which as we have seen is overriding the `resolveClass` method and providing a whitelist/black list for serialization, or to block features from the XML parser in the case of XML related issues. Other solutions have less common structure between them and require more in depth understanding of the issue and the project structure, for example Race condition issues require a thorough understanding of how the race condition happens, which threads are involved, and how to synchronize them properly using the appropriate fields.

Projects that manage files (read/write) based on user input (requesting) should pay extra attention to the paths they process. This is essentially a simple check that would prevent a huge implication. These attacks are not hard to carry on especially with knowledge about the application structure. This also applies to compressed files and extracting their content, in fact creating a harmful compressed file with files that have relative paths inside it is not a complicated task.

Through our study, we have seen that issues related to Improper Validation of Certificate with Host Mismatch occurred in many different projects, even though the fix is quite simple and does not require huge efforts to add to the code. This shows that developers focus on delivering the required functionality without focusing on security when they are coding, and they are not aware of the security implications of their code.

We also noticed that the nature of the project could have an implication on the types of security issues that may occur. For example, Jenkins and UAA projects both manage resources belonging to many users (multi-tenancy). We have seen multiple distinct issues related to horizontal privilege escalation in these projects (which essentially mean that user has access to other user resources), their solutions were by tying the resources more closely to the users who should have the privilege to access them. In other words, this means that application's domain not only influences the best practices and structure of the application but also the types of security issues that may occur, we hope future research will inspect security issues from a domain-related point of view. On the other hand, some of the studied issues are related to basic functionality that many applications could support regardless of their domain, an example of such kind of issues is the issue related to Information Exposure like Insufficiently Protected Credentials and Session identification values exposure. These types of issues should be brought to the attention of all developers, and we think this makes them a good starting point for developer education on security basics since the functionality related to them is quite common, easy to understand, but it may not be easy to define rules to detect them in the code using static analysis because different projects might use different naming conventions. With the willingness to add security education to developers' curriculum, we think that starting with this type of issues would have a good return on investment.

For projects that are starting out and have the luxury of deciding how to implement the required functionality, we suggest looking into security issues that are imposed by the usage of specific features versus their alternatives, having this kind of investigation at early stages in the project lifetime could help you avoid security issues with low or no additional

development efforts. As an example, we suggest trying to avoid depending on serialization/deserialization and replacing it with a cross platform structured data representation like JSON and YAML. Those serialization formats have libraries that facilitate working with them in most programming languages and don't impose the security issues that serialization does. For XML validation we suggest moving the validation step to an upper layer in the application code and keeping the XML parsers as secure as possible by using the proper configuration.

Finally, we support the claim that expert developers and security experts should not be distinct roles, development teams should have the required expertise to solve security issues with minimum reliance on external security experts. Issues related to thread synchronization stress this need as their solutions require an in-depth understanding of the problematic code and the concerned threads, how they collaborate to deliver the functionality and more importantly how to add code that puts proper constraints on their collaboration to fix the security issue.

Summary. In our thesis, we studied security vulnerability issues and their fixing commits from open source Java projects. Our study started from data validated by the SAP research team. We started our study with a preliminary analysis of code metrics on the fixing commits, we then demonstrated the categories of the studied commits, finally we went through each category explaining the context of the code and how the security issue happened with snippets from the fixing commits.

We have seen that security fixing code has a low footprint on the code base. We categorized the studied commits into seven main categories. Finally, we explained the issues from each category. Our study aims to increase the awareness of software developers on recent common security issues by providing the developers with code examples that they can relate to development tasks to better understand and avoid the issues.

Future Work. Our future work will be on the following points: 1) we will conduct a study with developers who have various expertise levels to assess how they perceive our

findings and the best format to present our results for maximum benefits. 2) we will look at more aspects related to the security issues and the fixes like their lifetime, how they get introduced to the code, whether they are fixed due to reports or as a part of other tasks, and other related interesting aspects like assessing the expertise of the developers who introduce the issues and the developers who fix the issues.

Bibliography

Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M. L., & Stransky, C. (2016). You get where you're looking for: The impact of information sources on code security. In *2016 ieee symposium on security and privacy (sp)* (pp. 289–305).

Assal, H., & Chiasson, S. (2019). 'think secure from the beginning': A survey with software developers. In *Proceedings of the 2019 chi conference on human factors in computing systems* (p. 289).

C1. (2017). *Deserialization of untrusted data - owasp.* https://www.owasp.org/index.php/Deserialization_of_untrusted_data. ((Accessed on 12/15/2019))

C10. (2009). *Testing for race conditions (owasp-at-010) - owasp.* https://www.owasp.org/index.php/Testing_for_Race_Conditions_(OWASP-AT-010). ((Accessed on 12/15/2019))

C11. (2009). *Password storage · owasp cheat sheet series.* https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html. ((Accessed on 12/15/2019))

C12. (2015). *Logging · owasp cheat sheet series.* https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html#data-to-exclude. ((Accessed on 12/15/2019))

C13. (2017). *Authentication · owasp cheat sheet series.* https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html#compare-password-hashes-using-safe-functions. ((Accessed on 12/15/2019))

C14. (2015). *Improper error handling - owasp.* https://www.owasp.org/index.php/

Improper_Error_Handling. ((Accessed on 12/15/2019))

C2. (2017). *Test upload of malicious files (otg-buslogic-009) - owasp.* https://www.owasp.org/index.php/Test_Upload_of_Malicious_Files_(OTG-BUSLOGIC-009)#Zip_files_path. ((Accessed on 12/15/2019))

C3. (2017). *Xml external entity (xxe) processing - owasp.* https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing. ((Accessed on 12/15/2019))

C4. (2015). *Path traversal - owasp.* https://www.owasp.org/index.php/Path_Traversal. ((Accessed on 12/15/2019))

C5. (2017). *Transport layer protection · owasp cheat sheet series.* https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html#use-correct-domain-names. ((Accessed on 12/15/2019))

C6. (2016). *Owasp secure coding practices checklist - owasp.* https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_Checklist. ((Accessed on 12/15/2019))

C7. (2018). *Crlf injection - owasp.* https://www.owasp.org/index.php/CRLF_Injection. ((Accessed on 12/15/2019))

C8. (2008). *Category:authorization vulnerability - owasp.* https://www.owasp.org/index.php/Category:Authorization_Vulnerability. ((Accessed on 12/15/2019))

C9. (2017). *Cross-site request forgery prevention · owasp cheat sheet series.* https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html. ((Accessed on 12/15/2019))

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and psychological measurement*, *20*(1), 37–46.

Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., & Fahl, S. (2017). Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 ieee symposium on security and privacy (sp)* (pp. 121–136).

Ghaffarian, S. M., & Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, *50*(4), 56.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Proceedings*

of the 31st international conference on software engineering (pp. 78–88).

Hemetsberger, A., & Reinhardt, C. (2006). Learning and knowledge-building in open-source communities: A social-experiential approach. *Management learning*, *37*(2), 187–214.

Holzinger, P., Triller, S., Bartel, A., & Bodden, E. (2016). An in-depth study of more than ten years of java exploitation. In *Proceedings of the 2016 acm sigsac conference on computer and communications security* (pp. 779–790).

Imtiaz, N., Rahman, A., Farhana, E., & Williams, L. (2019). Challenges with responding to static analysis tool alerts. In *Proceedings of the 16th international conference on mining software repositories* (pp. 245–249).

Jimenez, M., Papadakis, M., Bissyandé, T. F., & Klein, J. (2016). Profiling android vulnerabilities. In *2016 ieee international conference on software quality, reliability and security (qrs)* (pp. 222–229).

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 international conference on software engineering* (pp. 672–681).

Li, F., & Paxson, V. (2017). A large-scale empirical study of security patches. In *Proceedings of the 2017 acm sigsac conference on computer and communications security* (pp. 2201–2215).

Linares-Vásquez, M., Bavota, G., & Escobar-Velásquez, C. (2017). An empirical study on android-related vulnerabilities. In *2017 ieee/acm 14th international conference on mining software repositories (msr)* (pp. 2–13).

Meng, N., Nagy, S., Yao, D., Zhuang, W., & Arango-Argoty, G. (2018). Secure coding practices in java: Challenges and vulnerabilities. In *2018 ieee/acm 40th international conference on software engineering (icse)* (pp. 372–383).

Mitropoulos, D., Gousios, G., & Spinellis, D. (2012). Measuring the occurrence of security-related bugs through software evolution. In *2012 16th panhellenic conference on informatics* (pp. 117–122).

Nadeem, M., Williams, B. J., & Allen, E. B. (2012). High false positive detection of security vulnerabilities: a case study. In *Proceedings of the 50th annual southeast*

regional conference (pp. 359–360).

Oliveira, D., Rosenthal, M., Morin, N., Yeh, K.-C., Cappos, J., & Zhuang, Y. (2014). It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Proceedings of the 30th annual computer security applications conference* (pp. 296–305).

Ponta, S. E., Plate, H., Sabetta, A., Bezzi, M., & Dangremont, C. (2019). A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th international conference on mining software repositories* (pp. 383–387).

Sabetta, A., & Bezzi, M. (2018). A practical approach to the automatic classification of security-relevant commits. In *2018 ieee international conference on software maintenance and evolution (icsme)* (pp. 579–582).

Shahriar, H., & Zulkernine, M. (2012). Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)*, *44*(3), 11.

Shin, Y., Meneely, A., Williams, L., & Osborne, J. A. (2010). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, *37*(6), 772–787.

Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., & Zhai, C. (2014). Bug characteristics in open source software. *Empirical Software Engineering*, *19*(6), 1665–1705.

Thomas, T. W., Tabassum, M., Chu, B., & Lipford, H. (2018). Security during application development: an application security expert perspective. In *Proceedings of the 2018 chi conference on human factors in computing systems* (p. 262).

Wan, L. (2019). *Automated vulnerability detection system based on commit messages* (Unpublished doctoral dissertation).

Wen, S.-F. (2017). Software security in open source development: A systematic literature review. In *2017 21st conference of open innovations association (fruct)* (pp. 364–373).

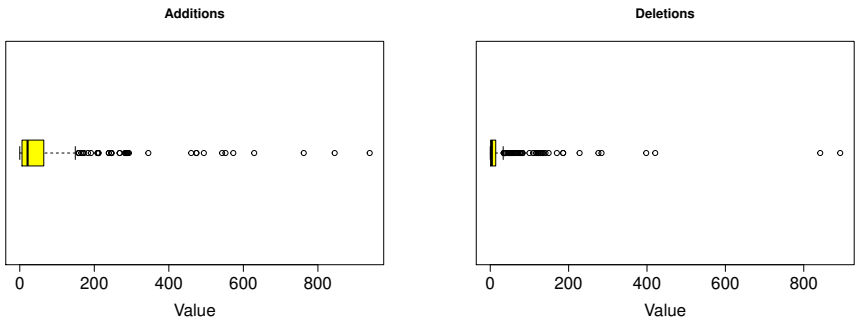
Wen, S.-F. (2018). Learning secure programming in open source software communities: a socio-technical view. In *Proceedings of the 6th international conference on information and education technology* (pp. 25–32).

Wood, J. R., & Wood, L. E. (2008). Card sorting: current practices and beyond. *Journal*

of Usability Studies, *4*(1), 1–6.

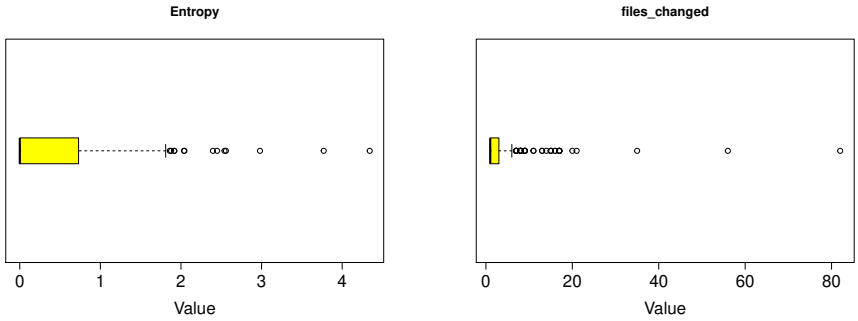
Xia, X., Shihab, E., Kamei, Y., Lo, D., & Wang, X. (2016). Predicting crashing releases of mobile applications. In *Proceedings of the 10th acm/ieee international symposium on empirical software engineering and measurement* (p. 29).

Appendix A



(a) Additions

(b) Deletions



(c) Entropy

(d) Files_Changed

Figure A.1: Metrics (with outliers)