The Extent of Orphan Vulnerabilities from Code Reuse in Open Source Software

Anonymous Author(s)

ABSTRACT

Motivation: A key premise of open source software is the ability to copy code to other open source projects (white-box reuse). Such copying accelerates development of new projects, but the code flaws in the original projects, such as vulnerabilities, may also spread even if fixed in the projects from where the code was appropriated. The extent of the spread of vulnerabilities through code reuse, the potential impact of such spread, or avenues for mitigating risk of these secondary vulnerabilities has not been studied in the context of a nearly complete collection of open source code.

Aim: We aim to find ways to detect the white-box reuse induced vulnerabilities, determine how prevalent they are, and explore how they may be addressed.

Method: We rely on World of Code infrastructure that provides a curated and cross-referenced collection of nearly all open source software to conduct a case study of a few known vulnerabilities. To conduct our case study we develop a tool, VDiOS, to help identify and fix white-box-reuse-induced vulnerabilities that have been already patched in the original projects (orphan vulnerabilities).

Results: We find numerous instances of orphan vulnerabilities even in currently active and in highly popular projects (over 1K stars). Even apparently inactive projects are still publicly available for others to use and spread the vulnerability further. The often long delay in fixing orphan vulnerabilities even in highly popular projects increases the chances of it spreading to new projects. We provided patches to a number of project maintainers and found that only a small percentage accepted and applied the patch. We hope that VDiOS will lead to further study and mitigation of risks from orphan vulnerabilities and other orphan code flaws.

CCS CONCEPTS

• **Software and its engineering** → *Software configuration management and version control systems.*

KEYWORDS

code reuse, CVE, security vulnerabilities, git

ACM Reference Format:

Anonymous Author(s). 2021. The Extent of Orphan Vulnerabilities from Code Reuse in Open Source Software. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The rapid growth of high quality open source software (OSS) has significantly increased the different kinds of software that can be built upon, thus potentially enhancing developer productivity [28], increasing code quality [12], and improving software security [15] [17]. A key feature of open source code is that it may be copied into new projects[1], but such copying may bring vulnerabilities or other issues [15]. We define "orphan vulnerabilities" as vulnerabilities in copied code that still exist in a project after they are discovered and fixed in another project. In some cases, the copying is a result of forking, and the link to the original code is readily available. In other cases, especially when the copying is a result of many iterations, the link to the original code may not exist. Either way, the vulnerable code is publicly exposed until the orphan vulnerability is fixed or the vulnerable code is removed. The aim of this study is to determine if the ability to copy OSS code actually results in widespread orphan vulnerabilities. Orphan vulnerabilities present significant risk for several reasons. First, an exploit for such vulnerabilities may be widely known, making it easier to attack software with known vulnerabilities [6]. Second, the code in such repositories may be copied to other projects that may not be aware of the vulnerability. Third, code in such repositories may be built into applications and run by unsuspecting users. Fourth, if a substantial number of OSS projects contain known and unfixed vulnerabilities, OSS may suffer reputational damage as a dump of low quality code where it may be hard to find high-quality projects.

To better understand and address the problem of copied and unpatched code, we first would like to create a tool that, given a vulnerability fix in one project, identifies all other projects that contain either still vulnerable or fixed code. Such a tool, if widely deployed, would have at least two positive impacts: inform maintainers and users of still vulnerable projects about the risks of the vulnerability in their code and warn users that contemplate reusing such code about the unpatched vulnerabilities.

Second, we want to determine if and how the still vulnerable projects may differ from the patched ones. For example, we expect that the more active projects are more likely to fix known vulnerabilities than the less active projects. This would suggest that the risks posed by unpatched projects may be attenuated by, presumably, more narrow deployment. Linus's Law states that "given enough eyeballs, all bugs are shallow" [32]. This would suggest that projects with more developers are less likely to contain vulnerabilities. But little empirical evidence exists to support this [11]. We want to see if our results support Linus's Law.

Third, we would like to understand how quickly patches to known vulnerabilities propagate to unpatched projects. We expect that older vulnerabilities are more likely to be fixed in a project than the more recent ones as it takes time and effort for project maintainers to patch their project. Presence of such a trend would

[1] Subject to licensing terms of the original and target project.

suggest that convenient tools supporting such patching may speed up the deployment of patches.

Fourth, we want to determine if the tool we introduced detects vulnerabilities of a different kind than one of the most widely known tools, dependabot [9], to determine if the approach used in our tool is practically relevant or if developers may safely rely on dependabot.

Fifth, we would like to identify how many of the projects that contain orphan vulnerabilities are not just forked from the original project where the vulnerability was fixed. Since many forks are done simply to contribute a patch, not to start a new development, it would not be surprising if such forks are not updated and do not patch their code. For any developer, it would be easy to look up the origin of the fork to get the most authoritative code. However, it may be harder to do with cloned projects. If, on the other hand, many of the projects are not forks, it would be much more difficult for potential users to identify such authoritative versions.

Sixth, we would like to understand to what extent the still vulnerable projects are willing to accept patches of the vulnerability offered to them. For example, while dependabot creates warnings and provides patches, not all projects are willing to accept them as the patches may break functionality.

To produce the tool, VDiOS, we build on top of World of Code (WoC) [24] infrastructure that attempts to approximate the source code in public git version control systems and provides cross-references among versions of the code, projects, and changes to the code.

To answer our research questions, we employ a mixed methods approach where we analyze large volumes of data to select candidates for a case study. Such an approach is suitable for our investigation because on one hand we have a very large and complex datasource representing almost all open source code, and we need computational approaches to select meaningful examples for our case study. The case study approach is needed because we have limited understanding of the problems, and a case study approach provides "an in-depth, multi-faceted exploration of complex issues in their real-life settings" [35]. We carefully pick the subjects (vulnerabilities) to shed light on all of the above research questions.

It is important to note that here we are exclusively focused on the so-called white box reuse where the source code is copied instead of employed as library/system call. Furthermore, we only consider matching any exact version of the vulnerable code, though the approach can be straightforwardly extended to cases where the copied code has been modified and does not match exactly any of the known fixed or vulnerable versions.

We succeeded in building VDiOS, a tool that identifies projects with orphan vulnerabilities, and applied it in four cases investigating four vulnerabilities in PNG library, OpenSSL, and xz compression (written in Go language). None of the vulnerabilities were reported by dependabot in thousands of vulnerable projects that are not forks of the original projects. Only a fairly small fraction of projects accepted the pull request fixing the known vulnerability. On the positive side, we found older vulnerabilities to be more likely to be fixed, and the still-vulnerable projects tended to be less active than the patched ones.

In summary, our work makes the following contributions:

- We provide a working approach to find file-level exact code reuse in any language across all open source repositories.
- We provide an approach to trace the version of a single file across all repositories and version history where either parents or descendants of that file reside.
- We provide a tool to implement our approach.
- We conduct a case study with four cases to answer our research questions regarding vulnerabilities that are spread via file-level code reuse.

Our primary objective is to reduce security vulnerabilities in software by identifying cases where a known vulnerability has been fixed, but copies of versions that are still vulnerable are still in use in other projects. This is a well-known security risk in the software supply chain. The Open Web Application Security Project (OWASP) lists "Using Components with Known Vulnerabilities" in its Top 10 Web Application Security Risks (OWASP Top 10) [31]. The software supply chain is a significant source of data breaches [23], with one estimate suggesting 80% of such breaches come from supply chain vulnerabilities [27]. Finding file-level duplication and locating where a file originated helps identify vulnerable or buggy code.

In the rest of the paper we start from the general background on the data used in the study in Section 2, discuss our research methodology in Section 3, present our VDiOS tool in Section 4, and the results from our case study in Section 5. We then discuss these results in Section 6. Finally we present limitations in Section 7, related work in Section 8, and conclude in Section 9.

2 BACKGROUND

2.1 Software Reuse

Software reuse is the practice of using existing software components when building new software systems [22]. There are two types of software reuse, often referred to as black-box and white-box reuse. Black-box reuse refers to external code that is used by a project but generally not committed into the project's repository. This may include, for example, linkable libraries. Black-box reuse is code that is not modified by the developer. White-box reuse refers to the case where source code is reused by copying the original code and committing the duplicate code into a new repository. White-box reuse code may be modified by the developer. White-box reuse results in multiple copies of the source code across multiple repositories. These copies may be changed, and therefore, there may be multiple different versions of the code. This paper specifically looks at white-box reuse. We look at code reuse on the individual file level, not at the function or method level.

White-box reuse presents several challenges. Vulnerabilities and other bugs may be found and fixed in a copy of the code that exists in one project, but the fixes may not get propagated to all projects that use the file. Similarly, useful enhancements may have been added to different versions of the code. The result is that fixes to known vulnerabilities as well as other bug fixes and enhancements may exist in one project but not in other projects. Also, license terms may not be properly propagated from the original code, causing license violations for developers who do not know the origins of the code. For quality and security reasons, it is important to understand where the reused code came from, who has worked on it, and if

better versions of it exist in other repositories. Knowing where else the code exists can help identify if there are known vulnerabilities in the code by seeing known vulnerabilities in other projects where the same code exists.

2.2 World of Code

Due to the vast quantity of open source software available from many different public source code repository hosting platforms, it has traditionally been too computationally intensive to find origins of a duplicated piece of code and all revisions of that code across all open source projects. Therefore, previous research on code reuse has typically looked at a relatively small subset of open source software. However, a new innovation, World of Code (WoC) [24], opens up new research possibilities in this area. WoC provides an infrastructure that makes it possible to efficiently find all versions of reused source code files across all of the major source code repository hosting platforms. We build on the WoC infrastructure to find file-level code duplication in any language from a WoC's expansive collection of open source software. Additional tools that build on WoC, such as Developer Reputation Estimator (DRE) [2], can be used to help identify the best of several versions of a file.

The tool described in this paper, VDiOS, uses the World of Code infrastructure to find duplicate code across many public source code hosting platforms. WoC is a nearly exhaustive and continually updated collection of open source software along with tools to efficiently extract and analyze the extremely large set of code. Without the infrastructure provided by WoC, it would not be possible to find such a complete collection of code copied (and possibly modified) across such a large collection of code in many repositories across many hosting platforms.

Since most open source software today is stored in git repositories, WoC uses similar constructs to store the data. For example, blobs, trees, and commits in WoC are identical to the same objects in git and are referenced with a sha1 hash just like git.

Black-box reuse can be detected with static analysis techniques that look for dependencies. These dependencies can be checked against public sources like libraries.io. But white-box reuse, which is the subject of this paper, requires access to the source code for all projects from which code may be reused. WoC provides not only the near complete collection of open source software, but also organizes its databases for efficient searching.

WoC provides a number of mappings that allow us to efficiently extract the information that we need. WoC maintains a database of several objects including blobs, files, commits, projects, and authors, allowing for efficient mappings. For example, given the contents of a file, we compute the SHA-1 hash (using the same mechanism that git uses) that identifies the blob. We then use WoC's blob to commit mapping to get the SHA-1 hash of the commit. The commit to project and commit to time author mappings give us the project name (from which we can identify the git repo from which it came) and the author and time of commit (which helps us identify where the file originally came from). We also use the blob to old blob mapping to find old versions of the source code of a particular file.

3 RESEARCH METHODOLOGY

We conducted an exploratory case study to better understand issues surrounding the spread of software security vulnerabilities caused by copying open source software. We chose to use an exploratory case study because we are in the early stages of understanding the problem and possible solutions. We hope to generate ideas to mitigate these types of security vulnerabilities and spur additional academic research. The case study approach allowed us to look at a small number of widely-reused projects in-depth and within their real-life context. This in depth examination allowed us to increase our understanding and gain insights that would otherwise be difficult to obtain.

Consistent with best practices conducting case studies, we investigated a small number of cases in depth and in their context using multiple data sources and emphasizing qualitative data and analysis while also collecting significant quantitative data. The subject of each case is a known vulnerability (as described by the Common Vulnerabilities and Exposures (CVE) database [40] hosted at MITRE) and the open source project containing the vulnerable code as described by the CVE entry.

We examined in detail four specific cases of known software security vulnerabilities that have been fixed in their original project repository. We used multiple cases to increase the confidence of the results and increase generalization of the results. We avoid making broad generalization claims based on just four cases, although we believe that our results provide insights that are applicable to a broader range than just our four specific cases. We carefully selected these four cases by searching for vulnerabilities in popular open source projects that have been widely copied. We used VDiOS to screen out cases of known vulnerabilities in code that is not widely copied. We specifically selected common cases, not unique or edge cases. We selected a vulnerability in libpng[2] that was in the code for a long time, allowing for many copies over that time. We selected a new and an old vulnerability in OpenSSL[3] to highlight differences in the age of the vulnerability. OpenSSL was chosen in part because it is critical to Internet security. We selected the xz package[4] written in the Go language to contrast with the other projects that were all C language projects. We selected cases that we believe are representative of the broader group of known vulnerabilities in open source software. Our cases represent both literal replication (because they are representative of the broader group of known vulnerabilities) and theoretical replication (because we compare an old vs new vulnerability in the same project and projects in different programming languages).

We examined these four cases in context by looking at specific open source projects that reuse vulnerable code and that are hosted on public hosting platforms including GitHub, Bitbucket, SourceForge, and others. By looking at the cases in context, we see a realistic picture of vulnerable code reuse in the real world rather than contrived results we might get in a traditional lab-based study.

Multiple data collection techniques provided corroborating evidence. We used artifacts, observations, and direct contact with project maintainers (through pull requests and issues), allowing

[2] http://www.libpng.org/
[3] https://www.openssl.org/
[4] https://github.com/ulikunitz/xz

us to gather more insights than using just one method. Looking at artifacts (the code actually committed in real world repositories) provides a concrete view of actual practice without any biases. Our observations allow for some qualitative analysis. Contacting project maintainers provides more insight into their willingness to address issues once they are aware of the vulnerabilities.

Case studies tend to focus more on qualitative data than quantitative data. Our study contains both. VDiOS produces significant quantitative data, which we report in detail. We also attempt to describe behavior based on our observations and interactions with project maintainers. Our results are also being used to craft survey questions for future research to collect more qualitative data to explain the behavior of project maintainers.

Our primary data source is World of Code. We also collect some data directly from the source code hosting platforms like GitHub, Bitbucket, SourceForge, and others. Data collection is accomplished using the VDiOS tool that we developed specifically for this research project. VDiOS is described in detail in section 4.

We started by selecting a sample of known vulnerabilities, identifying all affected (and fixed) versions of the source code files in the primary repositories and using WoC to identify all other OSS projects that have versions of the code that either precede (in version history) the affected version or is modified past it without applying the patch. We codified this algorithm as a tool that can be used for any vulnerability or any other type of defect. We then obtained and analyzed the numbers, activity states, and properties of the affected, patched, and potentially patched projects. Furthermore, we manually investigated many instances of cases where the code is still vulnerable to identify if the project is still active, if the defect has been fixed, and if not, whether the maintainers are willing to accept the patch.

4 THE VDiOS TOOL

In this section we describe VDiOS (Vulnerability Detection in Open Source), our tool for finding file level code reuse across all open source repositories and tracing the version of a single file across all repositories and version history. We build on the WoC infrastructure to find duplicate files at a scale that has traditionally been computational infeasible.

VDiOS takes the contents of a file and finds all duplicate versions of that file or any revision of that file across all of the open source software available in WoC. These vulnerable files are then traced back to the open source project in which they are contained. These projects may be hosted on many different source code hosting platforms such as GitHub, Bitbucket, SourceForge, etc. VDiOS displays a URL link to the project and the affected file or files within the given project.

Our approach looks for file-level reuse, that is, exact copies of entire files. We include all files in the version control history when looking for duplicate files. This allows us to find files that were duplicated and then modified.

When looking for Security Vulnerabilities, VDiOS has the ability to separate revisions of files into two lists: revisions that contain the vulnerability and revisions that do not contain the vulnerability. This allows VDiOS to identify projects that are still vulnerable, projects that used to be vulnerable but have now been fixed, and projects that used to be vulnerable and have changed but we do not know if the change fixed the vulnerability.

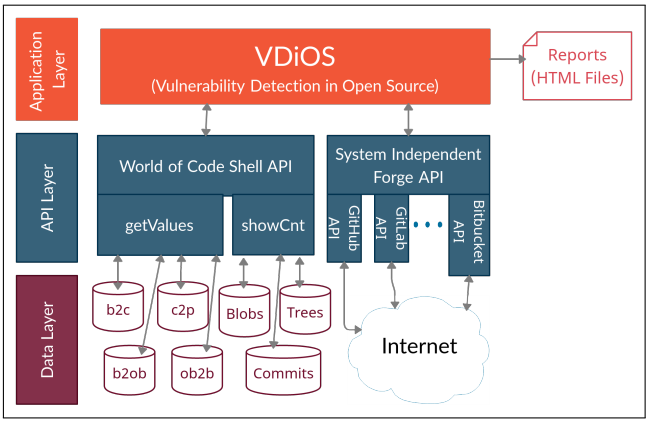

Figure 1: VDiOS Architecture Diagram

4.1 Architecture

VDiOS is implemented as a layer on top of the World of Code (WoC) shell APIs [25] as shown in figure 1. The WoC shell APIs provide a convenient way to access the WoC data. Specifically, VDiOS needs access to WoC's data maps as well as information about the objects. WoC stores data maps in a way that allows VDiOS to efficiently look up information. The specific information we need pertains to blobs, commits, projects, files, authors, and times.

There are two primary WoC shell APIs used by VDiOS: getValues and showCnt. showCnt is used to show the content of the basic git objects blob, tree and commit. getValues is used to access the WoC data maps to get the following information:

- blob to commit (b2c) finds all commits of the specified blob.
- commit to project (c2p) finds all projects containing the specified commit.
- commit to Project (c2P) is like c2p except it finds deforked [26] projects.
- commit to parent commit (c2pc) and commit to child commit (c2cc) finds the parent and child commit respectively from a given commit.
- commit to time author (c2ta) finds the time of the commit and the author of the commit.
- blob to old blob (b2ob) finds the predecessor of the given blob. old blob to blob (ob2b) is the inverse of b2ob.

VDiOS also retrieves a small amount of data directly from the source code hosting platform (GitHub, Bitbucket, GitLab, etc). A system independent interface allows VDiOS to use a single call to get data, hiding the platform specific details. A system dependent layer, which calls the appropriate API (for example, the GitHub API), provides a "glue layer" to connect to the popular hosting platforms to retrieve the data. The system dependent layer can be extended to support additional hosting platforms as needed.

The VDiOS output is a set of reports generated in HTML format for viewing in a web browser.

4.2 Algorithm

VDiOS is divided into four phases, each of which is described in this section.

The first phase identifies all of the blobs that contain the vulnerability and all the blobs that contain the fix. Starting with a commit that fixes a vulnerability, VDiOS finds the relevant blob or blobs in that commit. When looking for a security vulnerability, it is likely that not only is the revision before the fix vulnerable, but the predecessors of that revision are also likely to be vulnerable. VDiOS uses WoC's blob to old blob mapping or commit to parent commit mapping recursively to find all predecessor blobs. If we know the commit that introduced the vulnerability, VDiOS looks at only blobs between the breaking commit and the fixing commit. It is highly likely that all of those blobs will contain the vulnerability. VDiOS next finds the descendent blobs using WoC's old blob to blob mapping or commit to child commit mapping. These blobs are highly likely to contain the fix. Manual inspection of these lists can be done at this point to confirm that the blobs in the first list are vulnerable and the blobs in the second list are fixed. At the end of phase one, we have two lists of blobs. The first list contains one or more blobs that contain the vulnerability. The second list contains zero or more blobs that are fixed.

The second phase searches for all projects in WoC that contain a duplicate of any of the vulnerable blobs identified in phase one by using WoC's blob to commit mapping and commit to project mapping. Note that VDiOS looks for duplicates in any revision within a project. That is, it will find all projects that have ever contained the vulnerable blob even if it has been fixed or removed in the most current version. At the end of phase two, we have a list of all projects that have ever contained one of the potentially vulnerable versions of the file.

The third phase checks if the blob(s) in question are in the most current revision of the project. In this phase, VDiOS looks through the projects found in the second phase. Those are projects that have at some point in time contained a known vulnerable blob. We now want to find out if the project still contains a vulnerable blob. Using the hosting platform's API, we find the most current revision of the file. Now we look to see if that revision matches any of the vulnerable blobs. If so, we know the project still contains the vulnerable code. Next, we look to see if that revision matches any of the known good blobs. If so, we know that the vulnerable file has been fixed. If we do not find a potentially vulnerable or known good file, then we know that the project has contained a vulnerable file, that file has been changed, but we do not know if the change fixed the vulnerability.

The final phase generates the reports in HTML format for viewing in a browser. The first page of the report shows the commit that fixed the vulnerability (if applicable). Next, it has a link to a list of blobs and filenames where the vulnerability was fixed, a link to a list of ancestors of those blobs (which presumably contain the vulnerability), and a link to the descendants of those blobs (which presumably all contain the fix). Finally, it has links to lists of vulnerable projects, not vulnerable projects, and projects where the vulnerable file has been changed but we do not know if it is fixed or if it is still vulnerable. For each of the three categories (vulnerable, not vulnerable, and unknown), a report provides more detailed information.

5 RESULTS

In this section, we present the results of our case study involving four cases that demonstrate some of the security problems caused by orphan vulnerabilities. The four cases are four known security vulnerabilities that have now been fixed in popular open source projects. Our case study looks at projects that copied vulnerable files before the files were fixed in the original project from where they were copied. We look at two vulnerabilities within the widely used cryptography library OpenSSL. The first vulnerability is very recent and the second, heartbleed, is relatively old. We look at one recent vulnerability in a Go language package supporting xz compression. We look at one vulnerability that was fixed more than three years ago in the mature and proven open source PNG [5] graphics library libpng, which is very widely copied. Our case study looks at code written in different languages to show that our approach is language agnostic. It works the same regardless of the language.

We find tens of thousands of open source projects that contain files with known vulnerabilities even though the vulnerabilities have been fixed in the original project from where the vulnerable file was copied. Many of the vulnerable projects appear to be inactive, but some are clearly still active. In some cases the fix is recent and project maintainers have not had much time to apply patches. In other cases the fix is several years old, and yet many projects still contain the vulnerable code. Patches we provided were only accepted by a small percentage of project maintainers.

5.1 Case 1: CVE-2021-3449 in OpenSSL

OpenSSL is a very widely used open source cryptography library implementing Secure Socket Layer (SSL) and Transport Layer Security (TLS) [33]. Projects that incorporate OpenSSL play a vital role in Internet security. This was made clearly evident with the discovery in 2014 of the security vulnerability in OpenSSL known as heartbleed [7]. OpenSSL is the leading cryptography library used for email and website encryption and for software security in many other open source software packages. In this case study, we look at two vulnerabilities in OpenSSL. First, we look at the most recent (as of this writing) known vulnerability in OpenSSL. This vulnerability, described in CVE-2021-3449 [42], allows a maliciously crafted renegotiation ClientHello message to crash a TLS server. OpenSSL considers this a high severity vulnerability [30]. It was fixed in March 2021. Since it was only recently discovered and fixed, we might expect to find a number of projects that still contain the vulnerable code. The second OpenSSL vulnerability we look at, heartbleed, is discussed in the next section, 5.2.

The first OpenSSL vulnerability we look at, CVE-2021-3449, was introduced in the file ssl/statem/extensions.c in commit c589c34e61 in January 2018 and fixed in commit 02b1636fe3 in March 2021. According to the OpenSSL vulnerabilities list[5] "All OpenSSL 1.1.1 versions are affected by this issue. Users of these versions should upgrade to OpenSSL 1.1.1k." Since the vulnerability only existed

[5]https://www.openssl.org/news/vulnerabilities.html

in a few versions of OpenSSL, we expect to find a relatively small number of projects that use one of the vulnerable versions.

Following the algorithm described in section 4.2, VDiOS finds 56 revisions of extensions.c that contain the vulnerability. That is, there are 56 revisions between the commit that introduced the vulnerability and the commit that fixed the vulnerability. VDiOS finds three revisions of the file that contain the fix and are thus known to be not vulnerable to this specific issue. Additionally, VDiOS finds the following:

- 1,614 projects contain one of the known vulnerable revisions of ssl/statem/extensions.c in the most current revision of the project.
- 11 projects contain one of the known fixed revisions of ssl/statem/extensions.c in the most current revision, meaning it used to be vulnerable but now it is fixed.
- 1,079 projects contain a revision of ssl/statem/extensions.c that is not in either the list of vulnerable blobs or the list of fixed blobs, meaning that the project contained a potentially vulnerable blob in the past, the blob has been modified in the most current version, but we do not know if the modification fixed the vulnerability.
- 253 projects used to contain a vulnerable version of the file but the file has since been removed.

For further investigation of these projects, we used mappings in WoC (p2P) [**?**] to see how many of these projects are forked. Deforking 1614 vulnerable projects resulted into 132 projects. To see if they are active projects or not, we looked to see if they have any commit in the past 6 and 18 months. We found that 23 of them have at least one commit in the past 6 months, and 64 have at least one commit in the past 18 months. To have an idea about their impact in the OSS community, we looked at the number of stars [4] each of these projects have. We observed that four of these projects have more than 10,000 stars and 10 have more than a thousand stars, implying their wide impact in the OSS community.

5.2 Case 2: CVE-2014-0160 in OpenSSL

We next look at the OpenSSL heartbleed vulnerability. Heartbleed, described in CVE-2014-0160 [38], is a very serious vulnerability [44] that was fixed in 2014. Due to a bounds check error in the TLS heartbeat extension, the bug allows disclosure of information that should be protected. Since this was a high profile serious vulnerability that was fixed seven years ago, we expect not to find many, if any, active projects still using code vulnerable to heartbleed. We use VDiOS to test this hypothesis and then investigate the projects we find that still contain the heartbleed vulnerability.

Heartbleed was introduced by commit 4817504d06 on December 31, 2011, in the files ssl/t1_lib.c and ssl/dl_both.c. The first release of OpenSSL with this vulnerability was release 1.0.1 on March 14, 2012. The vulnerability was fixed two years later by commit 731f431497f made on April 7, 2014, and released in release 1.0.1g on April 7, 2014. VDiOS first finds all revisions of the file ssl/t1_lib.c between the December 2011 commit that introduced the vulnerability and the commit in April 2014 that fixed the vulnerability. It finds 90 vulnerable revisions of ssl/t1_lib.c. Following the procedure described in section 4.2 above to find projects containing the vulnerability, we discover the following results:

- 121 projects contain one of the known vulnerable revisions of ssl/t1_lib.c in the most current revision of the project.
- 3,156 projects contain one of the known fixed revisions of ssl/t1_lib.c in the most current revision, meaning it used to be vulnerable but now it is fixed.
- 211 projects contain revisions of ssl/t1_lib.c that is not in either the list of vulnerable blobs or the list of fixed blobs, meaning that the project contained a potentially vulnerable blob in the past, the blob has been modified in the most current version, but we do not know if the modification fixed the vulnerability.

Because of the very serious nature of heartbleed [10], we believe it is important to investigate all 121 projects that contain a known vulnerable version of ssl/t1_lib.c. We find the following information about these 121 projects:

- 110 of the projects are forks that were all forked between when the vulnerability was released in 2012 and when it was fixed in 2014 and that have had no activity on the project since before the vulnerability was fixed in 2014.
- Three of the projects are clones that were all cloned between when the vulnerability was released in 2012 and when it was fixed in 2014 and that have had no activity on the project since before the vulnerability was fixed in 2014.
- The remaining eight projects have had some activity (commits or issues) dated 2017 or later, well after the vulnerability was fixed. These projects are a potential concern, and therefore, we investigated these eight in more depth.

The 113 projects with no activity later than 2014 appear to be inactive projects. Of course any publicly available project containing heartbleed has the potential to be copied and reused, even if the project is not active. We find the remaining eight projects, the ones with activity dated 2017 or later, to be more concerning since they have been active since the vulnerability was fixed, yet they do not contain the fix. We looked into those eight projects in more detail and found the following information:

- One project has several commits this year (2021). This clearly indicates that it is an active project and potentially concerning since it contains the heartbleed code. Upon further investigation, we find that this project contains tools for the purpose of an empirical study of bugs in real world C software. OpenSSL is included as one of the subjects of the study rather than being linked into this project's software. Thus, this project is not vulnerable to heartbleed.
- Two related projects on GitHub have commits in 2018 and 2019, which would seem to indicate that they are still active. One of them added a WhiteSource Bolt[6] configuration file in 2019. The other is a fork of that project and updated its Configure script[7] and Travis CI[8] files in 2018. No other changes have been made to either project since 2013, well before heartbleed was discovered and fixed.
- Two related projects on GitHub have activity more recent than the 2014 fix. One of the projects has two open issues

[6] https://www.whitesourcesoftware.com/free-developer-tools/bolt/
[7] https://www.gnu.org/software/autoconf/autoconf.html
[8] https://travis-ci.com/

from November 2018 where someone asks questions that indicate they are actively using the project. The questions were never answered, and there are no recent commits, which indicates that the project is not active. But this does show that people could be using projects that have been inactive for many years. Another project is a fork of a fork of this project and has one commit in 2018. The commit is only a change in whitespace in a README. There are no other commits since 2013.

- Two related projects on GitLab have changes in 2018 that only affect whitespace in a README. One is a fork of the other. No substantive changes have been made to either project since the 2014 fix of heartbleed.
- One project, which is not a fork, has a number of commits in 2017, indicating that it has been active much more recently than the heartbleed fix. This GitHub project has zero stars, zero forks, and only 27 commits.

Based on the above information, we conclude that heartbleed is virtually eliminated, although not completely eliminated, from active open source software projects. However, a number of inactive projects that are still vulnerable to heartbleed are still readily available online and thus could still be reused.

5.3 Case 3: CVE-2021-29482 in Package xz

Our next case looks at a vulnerability in a popular Go language package. Most of our work to date has studied C language projects. VDiOS is completely independent of the language. We wanted to look at a Go project to demonstrate the language independence of VDiOS and WoC. The project at github.com/ulikunitz/xz is a Go language package supporting xz compression. The project, which is still under development, is subject to the vulnerability described by CVE-2021-29482 [41], which is identified as high severity by the National Vulnerability Database. The vulnerability was fixed in the file bits.go by commit 69c6093c7b on August 19, 2020, and released in release v0.5.8.

VDiOS found 11 versions of the file that are potentially vulnerable and two versions that are fixed. Using these two lists, VDiOS found 7,105 projects that are known to contain a vulnerable version of bits.go in the most current revision and 185 projects that are known to contain a fixed version in the most current revision. Since this is a very new (at the time of our study) vulnerability, it is not surprising that there are only 185 projects containing a fixed version. Only one project was found that contained the vulnerable file in the past but does not currently contain any of the known vulnerable or known fixed versions. We looked into this one case and found that the only difference was that it used the DOS/Windows format with carriage return and line feed ("\r\n") at the end of each line instead of the Unix format with only line feed ("\n"). There were 2,037 projects found that used to contain the vulnerable file but that no longer contain the file at all.

To examine projects further, we used project to deforked project (p2P) mappings [**?**] available through WoC to find out how many of the projects are not forked. Out of 7,105 vulnerable projects, this resulted in 758 unique projects that are not forked and contain this vulnerability. The numbers for not vulnerable projects are 185 and 82 respectively. To see how many of these deforked projects are actively maintained, we looked for those that have at least one commit in the past 6 and past 18 months (at the time of our study). We found that in vulnerable projects, 271 have at least one commit in the past 6 months, and 472 have a commit in the past 18 months. In the case of not vulnerable projects, these numbers are 68 for 6 months and 82 for 18 months. As we can see, the percentage of active projects in vulnerable projects (36% & 62%) are significantly lower than in not vulnerable projects (83% & 100%), which was intuitively expected. Nevertheless, not having a commit in a certain period of time does not mean that the other projects are not being used, and so it is still important to address the vulnerability issue. This already shows the significance of the vulnerability being widely spread.

To investigate the impact of this vulnerability from a different standpoint, we looked at the number of stars each of these projects has been given as a measure of their popularity in OSS [4]. The results show that in vulnerable projects, at least 443 projects have more than one star, 273 more than 10, 101 more than one hundred, 31 more than one thousand, and 10 projects have more than 10 thousand stars. In not vulnerable projects, the numbers are 71, 44, 23, 10 and 4 respectively.

Table 1: Case 3 - number of projects and their percentage from deforked projects

	Vulnerable		Not vulnerable	
Total	7,105		185	
Deforked	758		82	
Last commit < 6 months	271	35.75%	68	82.93%
Last commit < 18 months	472	62.27%	82	100.00%
>1 stars	443	58.44%	71	86.59%
>10 stars	273	36.02%	44	53.66%
>100 stars	101	13.32%	23	28.05%
>1,000 stars	31	4.09%	10	12.20%
>10,000 stars	10	1.32%	4	4.88%
>1 authors	531	70.05%	77	93.90%
>10 authors	254	33.51%	48	58.54%
>100 authors	55	7.26%	17	20.73%
>1,000 authors	10	1.32%	6	7.32%

As we can see in Table 1, the number of stars in projects that fixed the vulnerability is relatively higher than vulnerable projects, which again is what we would intuitively expect. We have also looked at the number of contributing authors in each project using WoC project to author mappings (P2A) which maps the deforked projects to aliased author IDs [13]. Looking at the percentages, it seems that vulnerable projects have relatively fewer developers involved.

5.4 Case 4: CVE-2017-12652 in libpng

Libpng [34] is a very popular open source graphics library for manipulating PNG (Portable Network Graphics) image files. It is an old library, dating back to 1995, and is still actively maintained. Because of its popularity and its very long history, we expect to find many copies in other open source projects, making it a strong case for our study. The libpng source code [43] is hosted on SourceForge [36] and mirrored on GitHub [16].

Libpng was the first case that we studied. Lessons learned from this case were applied to our study of the other cases. Improvements to VDiOS, as described later in this section, were applied based on those lessons learned.

This case specifically looks at the libpng file pngpread.c. That file is the subject of the vulnerability described by CVE-2017-12652 [39], which is labeled as a critical vulnerability in the National Vulnerability Database (NVD) [29]. The vulnerability was fixed in August of 2017 in release 1.6.32. This fix is in commit 347538e and the blob for pngpread.c at that revision is 45b23a7.

Using WoC's blob to old blob (b2ob) mapping recursively, VDiOS finds 951 old versions of the file pngpread.c. The old versions are the potentially vulnerable versions. Using WoC's old blob to blob (ob2b) mapping, VDiOS finds 964 new versions of that file. The new versions presumably all contain the fix. Next, VDiOS looks at each potentially vulnerable blob and uses WoC's blob to commit mapping to find the commits. Once it has the commits, it uses WoC's commit to project mapping to find all of the projects containing the discovered commits. This gives us a list of all projects that have ever contained one of the potentially vulnerable versions of the file pngpread.c. Finally, VDiOS looks at the head commit of each project to see if it contains a version of the file from the potentially vulnerable list, the presumably fixed list, or neither.

The results are as follows:

- 63,441 projects contain one of the potentially vulnerable blobs in the most current revision, even though it was fixed in the original file more than three years ago.
- 458 projects contain one of the presumably fixed blobs in the most current revision, meaning it used to be vulnerable but now it is no longer vulnerable.
- 20,274 projects do not contain blobs from either of the two previous lists, meaning that the project contained a potentially vulnerable blob in the past, the blob has been modified in the most current version, but we do not know if the modification fixed the vulnerability. We manually inspected the first 10 of those projects and found that two out of the 10 projects still contain the vulnerability. In those two cases, the file was modified, but the specific vulnerability was not fixed. In the remaining eight cases, the vulnerability was fixed.
- 28,376 projects used to contain a vulnerable version of the file, but the file has since been removed.

We see that over sixty thousand projects contain a vulnerable version of the file. We selected a subset of those projects to analyze in more detail. To select the subset, we first selected projects that have a commit within the last 18 months to eliminate long dormant projects. Next, we selected non-forked projects to get a list of independent projects. Finally, when one commit went to multiple projects, we selected the first one that VDiOS found and eliminated the remaining duplicates. This process of elimination leaves us with 1,457 projects. From those 1,457 projects, we randomly selected 88 projects to analyze in more detail. In looking at these projects, we find that they copy the entire contents of libpng, not just selected files.

Our first step is to verify that the 88 projects do indeed contain the vulnerable code. We manually inspected all of the projects and found six false positives. There were four projects that had deleted the vulnerable file and two projects that had fixed the vulnerable file. We removed those six projects from further analysis, leaving 82 projects. We investigated these six cases to understand why VDiOS produced false positives. In all six cases the reason was timing. We ran VDiOS to produce the results in early February 2021 and analyzed the results over the next two months. WoC is continuously updated, but will always be a little bit behind what is live on the source code repository hosting platforms. We ran VDiOS on version S of WoC which was updated in August 2020. We found in those six cases that the vulnerable files had been fixed or removed after the WoC version that VDiOS used to produce the reports and before we verified the results in April 2021. We conclude that VDiOS produced the correct output, but the continuously changing open source projects will be different from our reports to the extent that changes are made after the most recent WoC update. As a result of this discovery, we modified VDiOS to use the APIs of the hosting platforms to get the most current revision of the file. The results presented in this case are based on the original version of VDiOS; this new enhancement to VDiOS is used for the rest of the cases.

For projects hosted on GitHub, we also verified that GitHub's dependabot [9] did not find the vulnerability. While dependabot has similar goals to VDiOS in finding vulnerable dependencies in the software supply chain, it uses a very different mechanism. Dependabot requires that a repository define dependencies in a supported package ecosystem while VDiOS looks for file level code duplication. As expected, none of the projects we found with a vulnerable version of libpng were identified by dependabot. Several of the projects had other issues identified by dependabot, but not the libpng issue we are investigating. This shows that dependabot is enabled for these projects. Clearly VDiOS finds different supply chain dependency vulnerabilities than GitHub's dependabot.

Finally, we wanted to find out if the maintainers of the projects that contain known vulnerable files are willing to accept a patch to fix the vulnerability. For the 82 vulnerable projects, we produced a patch and sent a message to the maintainers through a pull request, an issue, or an email. We waited up to two weeks for responses. Seven project maintainers accepted our pull request with the patch. One project maintainer updated to a newer version of libpng because of our contact. Two project maintainers responded and said they would continue using the existing (vulnerable) code. We received no responses about the remaining projects.

Aside from what we found through these 82 projects, we wanted to have some overall statistics on activity and popularity measures of vulnerable and not vulnerable projects as we had in previous cases. Following the same procedures, we found that the 63,441 vulnerable projects reduce to 9,680 deforked projects from which 660 have at least one commit in the past 6 months and 2095 in the past 18 months. Other than that, there are at least 25 projects with more than 10 thousand stars and 131 projects with more than 1,000 stars which attest the importance of such vulnerabilities. The detailed numbers are presented in Table 2.

6 DISCUSSION

Our research is motivated by the orphan security vulnerabilities caused by code reuse in open source software. Our primary goal

Table 2: Case 4 - number of projects and their percentage from deforked projects

	Vulnerable		Not vulnerable	
Total	63,441		458	
Deforked	9,680		51	
Last commit < 6 months	660	6.82%	20	39.22%
Last commit < 18 months	2,095	21.64%	29	56.86%
>1 stars	2,648	27.36%	34	66.67%
>10 stars	1,089	11.25%	24	47.06%
>100 stars	395	4.08%	15	29.41%
>1,000 stars	131	1.35%	10	19.61%
>10,000 stars	25	0.26%	2	3.92%
>1 authors	4,317	44.60%	45	88.24%
>10 authors	632	6.53%	27	52.94%
>100 authors	173	1.79%	11	21.57%
>1,000 authors	44	0.45%	5	9.80%

is to better understand orphan security vulnerabilities, and how they may be mitigated. In this work, we explore the scope of the problem on a small sample of vulnerabilities and the willingness of project maintainers to fix issues.

The vast quantity of open source projects distributed over different hosting platforms complicates our task. By exploiting the World of Code infrastructure, we build a tool, VDiOS, that collects code reuse data with the coverage and scale that had previously been impractical. VDiOS is targeting orphan vulnerabilities unlike the National Vulnerability Database that identifies a vulnerability in a single project.

First, using VDiOS, we find a very large number of projects with orphan vulnerabilities based on the four vulnerabilities in our case study. As hypothesized, the probability of an orphan vulnerability is lower for more active projects. Also, supporting Linus's Law [32], the probability of an orphan vulnerability is lower for projects with more developers. Orphan vulnerabilities appear to concentrate in inactive or no longer maintained projects, but they are also present in very popular (over 10K stars) and very active projects as well. Orphan vulnerabilities, even if they are in unmaintained or inactive projects, still pose risks. First, a developer might copy code from such projects as, for example, they may have a unique feature that fixed projects lack. Second, code from inactive projects may still be running in existing systems, for example in embedded devices. We, in fact, found a case where someone asked a question about a project that appeared to be inactive, indicating that they were using it. By looking both at relatively old orphan vulnerabilities and very new orphan vulnerabilities, we observe relatively fewer old orphan vulnerabilities, suggesting that often orphan vulnerabilities are eventually fixed or removed. The time to fix appears to be substantial, providing opportunity for the orphan vulnerability to propagate further. Even very well known and very old vulnerabilities still persist in the orphan form.

Our attempts to gauge willingness of the project maintainers to fix orphan vulnerabilities yielded mixed results, with only a small fraction applying the patch.

Our case study suggests that orphan vulnerabilities are widespread, they take a very long time to be fixed, or they persist. They exist not only in forks or abandoned projects but also in highly active and popular projects as well. Even once an orphan vulnerability is identified and the fix provided to a maintainer, only a small fraction act upon the suggested fix. We conclude that orphan vulnerabilities pose an ongoing problem that needs to be addressed not just by identifying and providing fixes to the projects, but also by providing screening tools to projects reusing source code and by educating the open source development community.

7 LIMITATIONS

Our internal validation relates to the way the tool operates and the coverage of WoC data. Specifically, VDiOS looks for exact matches at the file level for the set of code versions between the versions that introduced and fixed the reference code. Code fragments copied from within a file may not be detected. First, this provides only a conservative estimate of vulnerable files, as minor modifications to vulnerable (or fixed) files may not be detected. Second, it is fairly straightforward to enhance VDiOS to look for snippets, patches, or more abstract representations of the vulnerability. It was not a priority for our case study but is important to cast a wider net for capturing more orphan vulnerabilities.

VDiOS takes the revision of a file that fixes a vulnerability and then uses WoC's blob to old blob (b2ob) and old blob to blob (ob2b) mappings recursively to find older and newer revisions of the file. Alternatively, it can use WoC's commit to parent commit (c2pc) and commit to child commit (c2cc) to find older revisions (up to the revision that introduced the vulnerability, if we know that) and newer revisions. The older revisions are likely to contain the vulnerability, and the newer revisions are likely to contain the fix. However, that is not guaranteed to be the case. In extremely rare cases projects revert back to vulnerable code even after fixing it. VDiOS allows a manual inspection and modification of the lists of old and new blobs to see if they are actually vulnerable and fixed respectively before moving on to the next phase. This manual intervention solves the problem, but to scale the solution VDiOS will need to be enhanced.

If a developer copies a file and makes a small change before committing for the first time, VDiOS will not find the match. Adding a new copyright notice, making formatting changes to match a style guide, or changing the CR/LF format at the end of lines are examples of inconsequential changes that would affect the ability to find a match. It will only find the match if the initial commit is identical to the copied file or a previous revision of the copied file. VDiOS can be enhanced to catch such and other modifications, and it is the subject of future work.

WoC contains a relatively complete collection of open source software, but the collection is not complete, with some projects missing and a several month delay between the versions of WoC when new projects with vulnerabilities may be created. VDiOS will miss any code that is not included in WoC. Only increasing open source coverage for WoC would address this limitation.

It is important to note that some vulnerabilities are never discovered or fixed, or not reported in public vulnerability databases. In all of these cases VDiOS would not help.

Our findings about the scope and age of orphan vulnerabilities is limited by the relatively small sample of vulnerabilities explored. We hope that by highlighting the scope and seriousness of the

problem with our case study and by building VDiOS, we will spur improvements to VDiOS and wider studies of vulnerabilities in the future.

8 RELATED WORK

Significant amount of research in the area of code reuse is dominated by studies of black-box reuse. Research on white-box code reuse, where code is reused by copying the original code and committing the duplicate code into a new repository, is limited due to the difficulty of searching the entirety of open source software looking for duplicates. Using World of Code (WoC) [24] infrastructure opens new research possibilities in the area of white-box code reuse as described in section 2.2. We use WoC to find cases of code reuse across open source projects.

Gharehyazie et al. [14] looked at the prevalence of cross-project code reuse and report large amounts of code cloned across multiple projects. They find that most cloned code comes from projects in a similar domain. GitHub was the only repository hosting platform that they looked at and Java was the only language. In our work, we look at code in many different languages and from many different repository hosting platforms including GitHub, Bitbucket, SourceForge, GitLab, and more.

Xia et al. [46] performed an empirical study to find the proportion of out-of-date third-party code reused by C language OSS projects. Using OpenCCFinder [45], which used external code search engines Google code search and SPARS [19], they found 123 projects that reused outdated code copied from three original projects. Similar to our findings, they determined that a significant number of OSS projects reused out-of-date code that contain security vulnerabilities. They report that OpenCCFinder only returns "a very small subset" of open source projects. By using our VDiOS tool layered on top of WoC's nearly complete collection of OSS in any language, we are able to find a significantly larger number of projects that reuse vulnerable code.

Decan et al. [8], through empirical study using Java projects that use Maven [37], show that it is common practice to use third-party software components that have known security vulnerabilities, suggesting that what we found for C and Go languages in white-box also applies to black-box reuse in Java. Alqahtani et al. [1] link the NVD[9] with Maven to identify known vulnerabilities in Maven projects. We expand on that by including white-box reuse and by looking at projects in any language that may not use or have management tools like Maven.

Kawamitsu et al. [21] studied code reuse across repositories, but only looked at reuse between pairs of repositories rather than across the full spectrum of open source repositories. They introduce a method to detect code reuse across 2 repositories.

Ishio et al. [20] proposed a method to find the original version of cloned source code files. Their method finds files that are similar, not just files that are exact copies. We only look for exact copies of any revision of the file. Their method may find additional matches that our method would miss due to minor changes in a cloned file before it is committed the first time. Our method may find matches that theirs miss because we run it over a much larger dataset of code repositories.

[9]National Vulnerability Database: https://nvd.nist.gov

Inoue et al. [18] use code search engines such as Google Code Search and Koders to find reused code fragments. They present a tool which takes code fragments and finds cloned fragments using the public code search engines. It is unclear what coverage is provided by these third-party tools.

GitHub's dependabot [9] creates pull requests for projects that rely on vulnerable libraries but only works for GitHub projects and only when dependencies are defined in a supported package ecosystem. VDiOS, on the other hand, looks for file level code duplication and does not rely on supported package ecosystems. VDiOS also works with projects across all repository hosting platforms, not just GitHub.

SZZ unleashed [3] finds information about when bugs were introduced. Currently, VDiOS relies on the user to specify the commit that introduced a vulnerability, but if it is not available, all previous revisions of a file are considered vulnerable. Using SZZ might reduce that set.

9 CONCLUSION

Code reuse through code duplication (white-box reuse) is a common practice in software development. While it has benefits, such as faster development time, lower cost, and improved quality, it also has inherent risks as the reused code may contain security vulnerabilities or other problems. In some cases, those vulnerabilities or bugs may be orphan (known and fixed in other repositories).

In this paper, we describe a case study with four different cases that show the extent of security vulnerabilities in open source software caused by code reuse. We also present a tool, VDiOS, to find file-level code reuse in any language across the entirety of open source software by leveraging the World of Code infrastructure. Using VDiOS, we find very extensive white-box reuse of vulnerable code with a large number of projects that do not appear to fix the upstream vulnerability. These are cases where reused code contains known vulnerabilities or other bugs that persist in open source projects even though they have been fixed in other projects.

Overall, we may conclude that extensive code copying in OSS results in an extensive spread of vulnerable code that may take years to fix and that affects not only inactive, but also highly active and popular projects. We also found that many of the projects may not be willing to patch the vulnerabilities even after being provided a fix.

These findings suggest that addressing unfixed vulnerabilities in OSS requires at least three types of support. On one hand, if a patch is provided, some of the projects are willing to apply it. On the other hand, for projects that do not fix vulnerable code, we need to provide information to potential users of the code that their application still contains unfixed vulnerability. Finally, developers who are contemplating reusing the code in a project that contains unfixed vulnerabilities need to be informed about the risks and provided with suggestions on how to patch or with patches fixing the existing vulnerabilities.

10 DATA AVAILABILITY

Our primary data source is World of Code [24]. The complete set of data produced for all four cases in our case study is available at https://figshare.com/s/0a4aed1675938a0d33b5.

REFERENCES

[1] Sultan S. Alqahtani, Ellis E. Eghan, and Juergen Rilling. 2016. SV-AF — A Security Vulnerability Analysis Framework. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 219–229. https://doi.org/10.1109/ISSRE.2016.12

[2] S. Amreen, A. Karnauch, and A. Mockus. 2019. Developer Reputation Estimator (DRE). In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1082–1085.

[3] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. SZZ Unleashed: An Open Implementation of the SZZ Algorithm - Featuring Example Usage in a Study of Just-in-Time Bug Prediction for the Jenkins Project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation* (Tallinn, Estonia) *(MaLTeSQuE 2019)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3340482.3342742

[4] Hudson Borges and Marco Tulio Valente. 2018. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[5] Thomas Boutell. 1997. PNG (Portable Network Graphics) Specification Version 1.0. RFC 2083. https://doi.org/10.17487/RFC2083

[6] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 516–519. https://doi.org/10.1109/SANER.2015.7081868

[7] Marco Carvalho, Jared DeMott, Richard Ford, and David A. Wheeler. 2014. Heartbleed 101. *IEEE Security Privacy* 12, 4 (2014), 63–67. https://doi.org/10.1109/MSP.2014.66

[8] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 181–191. https://doi.org/10.1145/3196398.3196401

[9] Dependabot. 2021. *Github Dependabot*. https://github.com/dependabot

[10] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755

[11] Danilo Favato, Daniel Ishitani, Johnatan Oliveira, and Eduardo Figueiredo. 2019. Linus's Law: More Eyes Fewer Flaws in Open Source Projects. In *Proceedings of the XVIII Brazilian Symposium on Software Quality* (Fortaleza, Brazil) *(SBQS'19)*. Association for Computing Machinery, New York, NY, USA, 69–78. https://doi.org/10.1145/3364641.3364650

[12] W.B. Frakes and Kyo Kang. 2005. Software reuse research: status and future. *IEEE Transactions on Software Engineering* 31, 7 (2005), 529–536. https://doi.org/10.1109/TSE.2005.85

[13] Tanner Fry, Tapajit Dey, Andrey Karnauch, and Audris Mockus. 2020. A dataset and an approach for identity resolution of 38 million author ids extracted from 2b git commits. In *Proceedings of the 17th international conference on mining software repositories*. 518–522.

[14] M. Gharehyazie, B. Ray, and V. Filkov. 2017. Some from Here, Some from There: Cross-Project Code Reuse in GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 291–301.

[15] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2021. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software* 172 (2021), 110653. https://doi.org/10.1016/j.jss.2020.110653

[16] Glenn Randers-Pehrson. 2020. *glennrp/libpng* . https://github.com/glennrp/libpng

[17] Jaap-Henk Hoepman and Bart Jacobs. 2007. Increased Security through Open Source. *Commun. ACM* 50, 1 (Jan. 2007), 79–83. https://doi.org/10.1145/1188913.1188921

[18] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe. 2012. Where does this code come from and where does it go? — Integrated code history tracker for open source systems. In *2012 34th International Conference on Software Engineering (ICSE)*. 331–341.

[19] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. 2005. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering* 31, 3 (2005), 213–225. https://doi.org/10.1109/TSE.2005.38

[20] T. Ishio, Y. Sakaguchi, K. Ito, and K. Inoue. 2017. Source File Set Search for Clone-and-Own Reuse Analysis. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 257–268.

[21] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue. 2014. Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 305–314.

[22] Charles W. Krueger. 1992. Software Reuse. *ACM Comput. Surv.* 24, 2 (June 1992), 131–183. https://doi.org/10.1145/130844.130856

[23] Nir Kshetri and Jeffrey Voas. 2019. Supply Chain Trust. *IT Professional* 21, 2 (2019), 6–10. https://doi.org/10.1109/MITP.2019.2895423

[24] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus. 2019. World of Code: An Infrastructure for Mining the Universe of Open Source VCS Data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 143–154.

[25] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. 2021. World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empirical Software Engineering* 26 (2021). https://doi.org/10.1007/s10664-020-09905-9

[26] Audris Mockus, Diomidis Spinellis, Zoe Kotti, and Gabriel John Dusing. 2020. A complete set of related git repositories identified via community detection approaches based on shared commits. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 513–517.

[27] Paul Myerson. 2017. *Can't Turn Back Time: Cybersecurity Must Be Dealt With*. https://www.industryweek.com/supply-chain/article/22006116/cant-turn-back-time-cybersecurity-must-be-dealt-with

[28] Frank Nagle. 2019. Open Source Software and Firm Productivity. *Management Science* 65, 3 (2019), 1191–1215. https://doi.org/10.1287/mnsc.2017.2977

[29] National Institute of Standards and Technology. 2021. *National Vulnerability Database*. http://nvd.nist.gov

[30] OpenSSL. 2021. *News/Vulnerabilities*. https://www.openssl.org/news/vulnerabilities.html

[31] OWASP. 2017. *The Open Web Application Security Project OWASP Top 10*. https://owasp.org/www-project-top-ten

[32] Eric Raymond. 1999. The cathedral and the bazaar. *Knowledge, Technology and Policy* 12 (1999), 23–49. https://doi.org/10.1007/s12130-999-1026-0

[33] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446

[34] Greg Roelofs. 2006. *libpng.org*. Retrieved August 4, 2020 from http://www.libpng.org

[35] Crowe S, Cresswell K, Robertson A, Huby G, Avery A, and Sheikh A. 2011. The case study approach. *BMC Medical Research Methodology* (2011). https://doi.org/10.1186/1471-2288-11-100

[36] Slashdot Media. 2020. *SourceForge*. https://sourceforge.net

[37] The Apache Software Foundation. 2021. *Apache Maven Project*. https://maven.apache.org/

[38] The MITRE Corporation. 2014. *CVE-2014-0160*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[39] The MITRE Corporation. 2017. *CVE-2017-12652*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12652

[40] The MITRE Corporation. 2021. *Common Vulnerabilities and Exposures (CVE)*. https://cve.mitre.org/

[41] The MITRE Corporation. 2021. *CVE-2021-29482*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29482

[42] The MITRE Corporation. 2021. *CVE-2021-3449*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3449

[43] Cosmin Truta and Glenn Randers-Pehrson. 2020. *LIBPNG: PNG reference library*. https://sourceforge.net/projects/libpng

[44] Jun Wang, Mingyi Zhao, Qiang Zeng, Dinghao Wu, and Peng Liu. 2015. Risk Assessment of Buffer "Heartbleed" Over-Read Vulnerabilities. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 555–562. https://doi.org/10.1109/DSN.2015.59

[45] Pei Xia, Yuki Manabe, Norihiro Yoshida, and Katsuro Inoue. 2012. Development of a Code Clone Search Tool for Open Source Repositories. *Information and Media Technologies* 7, 4 (2012), 1370–1376. https://doi.org/10.11185/imt.7.1370

[46] Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. 2014. Studying Reuse of Out-dated Third-party Code in Open Source Projects. *Information and Media Technologies* 9, 2 (2014), 155–161. https://doi.org/10.11185/imt.9.155