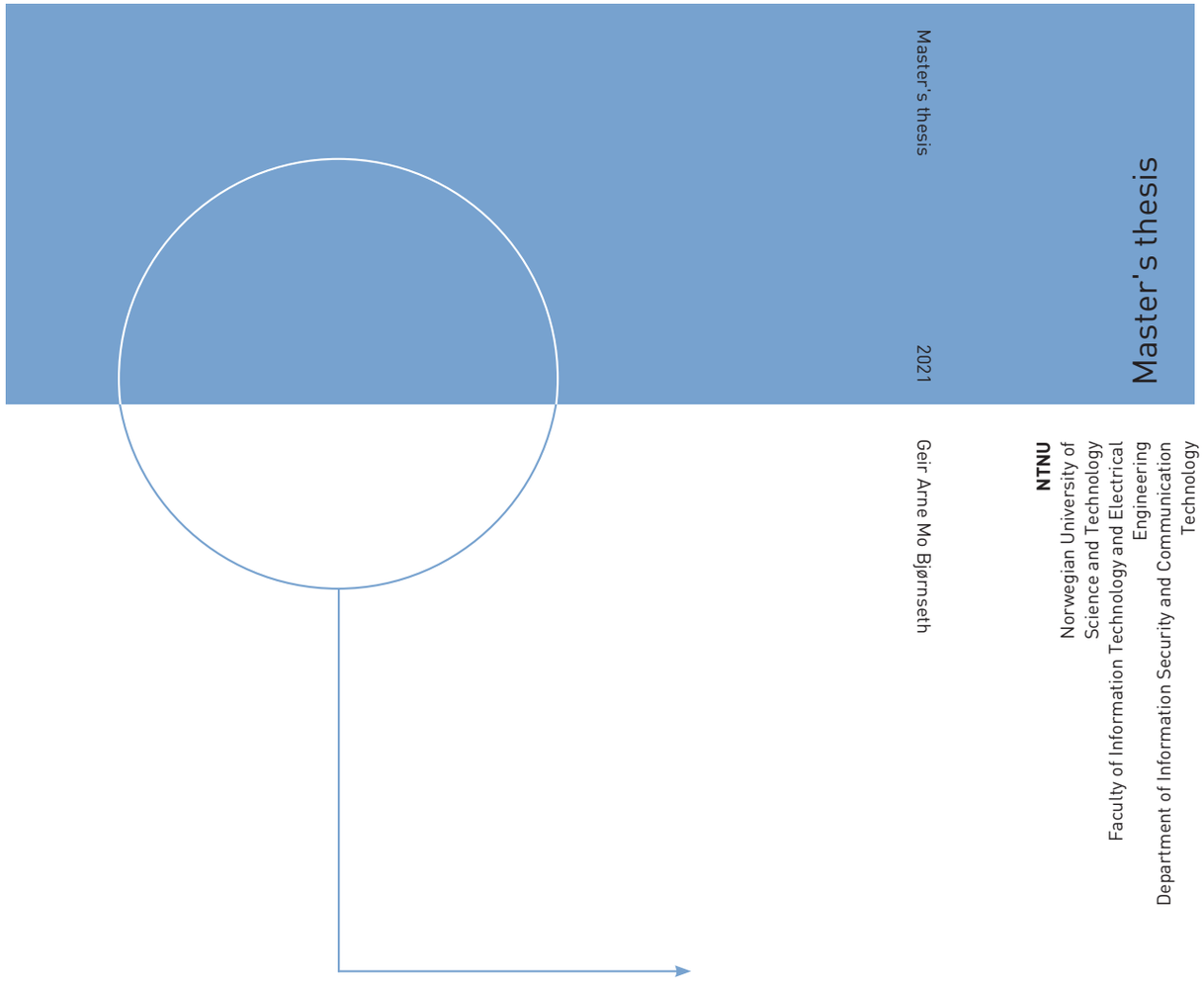

Master's thesis

2021

Geir Arne Mo Bjørnseth

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Information Security and Communication
Technology

Geir Arne Mo Bjørnseth

Studying vulnerability history in an open-source software package

June 2021

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Studying vulnerability history in an open-source software package

Geir Arne Mo Bjørnseth

Master in Information Security
Submission date: June 2021
Supervisor: Basel Katt

Norwegian University of Science and Technology
Department of Information Security and Communication
Technology

Abstract

Resent years has seen an increased focus on creating secure software with tools and frameworks like Microsoft Security Development Life Cycle and OWASP Software Maturity Model, but still we see well known and well documented vulnerabilities like injections, cross site scripting and buffer overflow in lists over most common vulnerabilities. Writing secure software can therefore be a challenging task, and research into security vulnerabilities can help to understand and improve software security. Many of these studies focus on the quantitative aspect of the subject, like vulnerability lifespan, effect of code review coverage on vulnerabilities, and metrics like commit sizes in vulnerable code. Such studies can be helpful in gaining insight into general trends of vulnerability evolution or insight into measures and scoring systems to identify vulnerable code. On the other hand, they give little insight into what causes the vulnerabilities to emerge and evolve and is the question we will try to answer in this thesis.

To answer the question about how vulnerabilities emerge and evolve, we study the vulnerability history in the Libarchive Open-Source Software (OSS) package. With an exploratory qualitative approach, we analyse artefacts like code updates, vulnerability reports and discussions related to the vulnerabilities in the OSS package and identify patterns and phenomena behind the vulnerabilities. We also perform an analysis into the Socio-Technical System (STS) surrounding the vulnerability handling in the OSS package. Based on our analysis we present a Vulnerability Evolution model describing the phenomena behind the vulnerabilities and the influence of the STS into these phenomena. We also present memory safety taxonomy describing the types of errors, sinks, and fixes behind the vulnerabilities. This taxonomy builds on a previous buffer overflow vulnerability taxonomy by Schuckert *et al.* [1]. Together the model and the taxonomy serve as tools to understand how vulnerabilities emerge and evolve and can be used to improve development process to produce secure code.

Sammendrag

I løpet av de siste årene har vi sett et økt fokus på utvikling av sikker programvare med rammeverk og verktøy som Microsoft Security Development Life Cycle og OWASP Software Maturity Model, men fortsatt ser vi kjente og godt dokumenterte sårbarheter som injections, cross site scripting og buffer overflows i lister over de mest vanlige sårbarhetene. Det å skrive sikker kildekode kan derfor være en utfordring og studier rundt sårbarheter i kildekode kan derfor være til hjelp for å forstå og forbedre programvare sikkerhet. Mange slike studier setter søkelys på det kvantitative aspektet rundt programvaresikkerhet, som for eksempel levetiden til sårbarheter, effekten av kodegjennomgang på sårbarheter, eller målinger som størrelsen på kodeendringer i sårbar kode. Slike studier kan gi innsikt i generelle trender rundt programvaresårbarheter eller innsikt inn i hvordan sårbarheter kan måles i kildekode. På den andre siden gir slike studier liten forståelse for hvordan sårbarheter oppstår og utvikler seg, og dette spørsmålet er teamet for dette prosjektet.

For å svare på spørsmålet om hvordan sårbarheter oppstår og utvikler seg i kildekode vill gi gjøre en studie av sårbarhetshistorikken i det åpne kildekode prosjektet Libarchive. Med en undersøkende og kvalitativ tilnærming analyserer vi artefakter rundt sårbarhetene i kildekoden, som kodeoppdateringer, sårbarhetsrapporter og diskusjoner. Ut ifra denne analysen vill vi identifisere mønstre og fenomener rundt sårbarhetene. I tillegg vil vi også gjøre en analyse av det Sosio-Tekniske systemet rundt sårbarhetshåndtering i prosjektet. Basert på disse analysene presenter vi en sårbarhetsmodell som beskrive fenomenene rund kodesårbarheter og hvordan det Sosio-Tekniske systemet spiller inn i disse fenomenene. Vi presenterer også en taksonomi for minnerelaterte sårbarheter, med type feil, steder for feil og rettelser av feil for denne typen sårbarheter. Denen taksonomien bygger på en tidliger buffer overflow taksonomi av Schuckert *et al.* [1]. Sammen gir modellen og taksonomien økt forståelse for hvordan sårbarheter oppstår og utvikler seg i kildekode og kan benyttes som verktøy for å forbedre utviklingsprosessen og sikkerheten i kildekode.

Acknowledgements

A special thanks to Basel Katt for being my supervisor, and all the feedback and guidance he provided during the work on my thesis.

G.A.M.B

Contents

Figures

Tables

Code Listings

Acronyms

CCSS Common Configuration Scoring System. 15

CMSS Common Misuse Scoring System. 15

CVE Common Vulnerabilities and Exposures. 8, 16, 21, 22, 25, 27, 32, 44, 51, 62, 65

CVSS Common Vulnerability Scoring System. 15, 16

CWE Common Weakness Enumeration. 37, 39, 40, 42, 44

OSS Open-Source Software. iii, 1–3, 5, 16, 19–25, 27, 45, 46, 52, 53, 55, 57, 58, 60, 65, 67, 69, 70

SDLC Software Development Life Cycle. 2, 24, 70

STS Socio-Technical System. iii, 2, 3, 22, 23, 25, 52, 57–59, 65–67, 69, 70

Glossary

DokuWiki Open-Source wiki software. 60, 62–65, 70

Libarchive Is an open-source C programming language archive providing streaming access to different archive formats. iii, 1, 3, 5, 12, 16, 17, 25, 27, 28, 30–37, 41, 43–46, 48–57, 59, 60, 62, 64–66, 69, 70, 77

Libsndfile Is an open-source C library for reading and writing files containing sampled audio data. 60–62, 65, 70

Chapter 1

Introduction

1.1 Topic covered by the thesis

Though there has been an increased focus on creating secure software over the last years, with tools and frameworks like Microsoft Security Development Life Cycle [3] and OWASP Software Assurance Maturity Module (SAMM) [4], vulnerable software continues to be a problem. Looking at lists like OWASP Top Ten [5] and CWE Top 25 [6] we still find well know and well documented vulnerabilities like injections, cross site scripting and buffer overflow.

In software development we also see that vulnerabilities often are introduced during code maintenance, that existing vulnerabilities often are missed, and that implemented defence against vulnerabilities often are incomplete [7]. One example is the OpenSSL Heartbleed vulnerability [8], which due to missing input validation allowed reading of protected memory of the vulnerable versions of OpneSSL. The vulnerability was introduced in 2012 and not fixed until 2014 [8]. Another example, also due to improper input validation, is the Shellshock vulnerability in Bash shell, which allowed custom code execution [7]. A fix was released on disclosure in 2014, but five further vulnerabilities and fixes followed before the issue was fixed [7].

As these examples shows, writing secure software is a challenging task, and research into security vulnerabilities can help us understand and improve software security and reduce the risk of exploitation. One approach when studying security vulnerabilities is to develop measures, scoring systems and categorisation of vulnerabilities that can give insight into general trends of vulnerability evolution and help to classify vulnerable source code. This can be helpful in preventing vulnerable software reach production systems or identify vulnerabilities already in production, but it does not help in understanding how vulnerabilities emerge and how they can be prevented form happen in the first place. This will be the focus of this thesis project, and with an exploratory qualitative approach we will study the vulnerability history in the Libarchive Open-Source Software (OSS) package to explore the question of how vulnerabilities emerge and evolve in code.

1.2 Keywords

Vulnerability, Open-Source Software, Socio-Technical System, Exploratory Study

1.3 Problem description

Many studies into security vulnerabilities focus on the quantitative aspect of the subject. This can be the number of vulnerabilities discovered after the software package *end-of-life* and number of vulnerabilities inherited from previous versions of the software packaged [9], the relationship between bugs and vulnerabilities [10], the effect on code review coverage on vulnerabilities [11] or metrics such as size of commits via code churns in vulnerable source code files [12]. Such studies can be helpful in gaining insight into general trends of vulnerability evolution or insight into useful measures and scoring system used to identify vulnerable source code. On the other hand, they give little insight into what cause vulnerabilities to emerge. The target of this project is to investigate this problem and try to find answers to what cause vulnerabilities to emerge, how they evolve and how they can be avoided in the future. In answering this question, the study will consider both social and technical aspects related to how vulnerabilities emerge and evolve.

1.4 Justification, motivation and benefits

With knowledge into how vulnerabilities emerge and evolve in a software project, changes can be made to the Software Development Life Cycle (SDLC) to prevent the vulnerabilities from occurring in the first place. Secure software can be developed from the beginning and resources used to identify and fix vulnerabilities can be put to better use elsewhere.

1.5 Research questions

This thesis will try to answer the following question:

- *How does vulnerabilities emerge and evolve in an OSS package?*

To help answer this question we will answer the following sub-questions:

- *What insight into vulnerability evolution can be gained by studying artefacts, including software components, attacks and changes to the component due to the attack, and reporting and other dialogues surrounding the vulnerabilities, related to the vulnerabilities?*
- *What code patterns do we find behind the types of errors causing vulnerabilities and the fixes to these?*
- *What phenomena can explain the emergence of vulnerabilities?*

- *What insight into vulnerability evolution can be gained by analysing the Socio-Technical System (STS) surrounding the vulnerability handling in the OSS project?*

1.6 Contributions

In this thesis project we have studied the vulnerability history in the Libarchive OSS package. By gathering observations on artefacts related to the vulnerabilities, including code changes, vulnerabilities reports and dialogues around the vulnerabilities, we have gained insight into the phenomena surrounding the evolution of vulnerabilities. From this knowledge we will present the following contributions:

- A model of the phenomena and the socio-technical system surrounding the emergence and evolution of vulnerabilities in a OSS package.
- A memory safety taxonomy presenting types of errors, sinks and type of fixes to memory vulnerabilities. This taxonomy builds on a previous buffer overflow taxonomy developed by Schuckert *et al.* [1].

1.7 Thesis outline

The thesis consists of six main chapters as follows:

- Chapter 1 gives the introduction to the thesis project and research questions we will answer.
- Chapter 2 gives background information into to the thesis topic and present related work and an introduction into the Libarchive Open-Source Software (OSS) package.
- Chapter 3 present the choice of methods used in the making of this thesis.
- Chapter 4 present the results from our study into the vulnerability history in Libarchive.
- Chapter 5 present our developed vulnerability model and discuss the result.
- Chapter 6 gives the conclusion and proposal for further work.

Chapter 2

Background and related work

This chapter contains background information and related work to the research questions presented in Chapter 1. Section 2.1 present a survey into studies related to software security vulnerabilities. In Section 2.2 we give an introduction into vulnerability scoring used to categorise and prioritise vulnerabilities. Last, in Section 2.3 we give a short introduction into Libarchive, the OSS package studied in this thesis project.

2.1 Related work

This thesis focuses on the question of how security vulnerabilities emerge and evolve in an open source software package and answering the question through observations of artefacts related to vulnerabilities in the software package. Studying vulnerabilities can help us to better understand and improve software security, and vulnerability studies is an extensive research field. In contrast to this thesis however, many of the existing studies focus on a quantitative approach trying to provide measures of the health of software security, vulnerability trends, etc. Other studies describe source code patterns of vulnerability categories like SQL injection and cross site scripting, and there are also studies into the effect of security knowledge of developers and developers ability to fully understand the security implications in all parts of a software project. All this gives insight into the question of how vulnerabilities emerge and evolve, and a survey of these studies follows in this section. The survey builds the literature review started during the work on our project plan report.

2.1.1 Vulnerability measures and metrics

In their paper, Ozment and Schechter [13] examined the code base of the OpenBSD operating system to determine if security is increasing over time. They found that 62% of the vulnerabilities reported during the time of the study was introduced prior to the first version of OpenBSD included in the study, version 2.3 (referred to as foundational vulnerabilities). This is explained by legacy code constituting

a large part of the total code base. In version 3.7 of OpenBSD, 61% of the code base is foundational, meaning that it was introduced in, and has been unchanged since, or prior to version 2.3 released 7.5 years earlier. The study also found the median lifetime of a vulnerability to be 2.6 years. The median lifetime was calculated as the time elapsed between the release of a version and death of half of the vulnerabilities in that version. Last, the study also found a decrease in reported vulnerabilities from 0.051 per day at the start of the study to 0.024 at the end [13]. An argument from this is that software grows more secure over time, with fewer reported vulnerabilities and large part of the vulnerabilities originated in legacy (foundational) code. But there is also an interesting point that vulnerabilities are introduced early in the software lifetime and tends to live on for a long time.

The findings from [13] is partly confirmed by Massacci *et al.* [9]. Examining vulnerabilities in the Firefox web browser from version 1.0 to version 3.6 they found a significant statistical difference between local vulnerabilities (found and fixed in same version) and inherited vulnerabilities (discovered in this version but applicable to previous versions) or foundational vulnerabilities (originated in version 1.0). Foundational vulnerabilities are found to be significantly more than they should be, and inherited ones less than they should be. As in [13], this can be explained by legacy code, or slow code evolution. 40% of the code base in version 3.6 originated from version 1.0. The study also found that many vulnerabilities are discovered after end-of-life of a Firefox version (after-life vulnerabilities). The after-life vulnerabilities accounted for at least 30% for version 1.0 of Firefox [9]. There is a difference in the definition of foundational vulnerabilities between [13] and [9], where [13] define this as vulnerabilities that existed at the start of the study while the definition in [13] is vulnerabilities introduced in version 1.0. But again, we see that vulnerabilities are introduced early in the life of the software tends to live on for a long time, possibly explained by the influence of legacy code [9].

Shin *et al.* [14] examined if software metrics obtained from source code and development history are discriminative and predictive of vulnerable code locations. The examined metrics are code complexity, code churn and developer activity, and the goal was to guide security inspection by predicting vulnerable files through these metrics. The code churn metrics is a measure of the number of check-ins and amount of code changes during development. Performing a case study on the Firefox web browser and Red Hat Enterprise Linux Kernel they found discriminating power of at least 24 of 28 metrics for both of the projects. In the code complexity category 14 different metrics was used related to internal complexity in a file, coupling between files and density of comments. Complex files can be difficult to understand, test and maintain and therefore more vulnerable. Highly coupled code will have more input from external source code or use interfaces to external modules, that can be either difficult to trace or implemented wrongly to cause vulnerability. Low comment density in a file can tell if a novice developer contributed to a file, or if the code was developed in a hurry. Both can

be indications of vulnerabilities in the code. In the code churn category, three metrics were used. These metrics were the total number of changes (check-ins) for a file, the total number of changed code lines since creation and the total number of new lines added since creation. Each change to a file brings risk of introducing a new vulnerability and the metrics counts different changes to a source code file. The developer activity category consists of 10 different metrics, divided into developer network centrality, developer network cluster and contribution network. A central developer will have better understanding of the source code and coding practice and thus contributing to fewer vulnerabilities than non-central developers. A cluster of developers might not communicate about software security and vulnerable source code files might be more likely to be changed by multiple separate developer clusters than neutral files. A file changed by many developers that also has changed many other files has an unfocused contribution and might be more likely to be vulnerable than a neutral file. Metrics in all categories proved discriminating powers in both case studies. The historic metrics of code churn and developer activity showed better prediction performance than the complexity metrics [14]. In other words, we see that the number of changes to file and who contributed to these changes can tell if a file is vulnerable. More changes, by many different developers or different clusters of developers can be more vulnerable than other files. Code complexity can also contribute vulnerabilities.

Similar vulnerability metrics are explored by Meneely *et al.* [12]. Analysing vulnerabilities in the Apache HTTP Server project they explored the size, interactive churn, and community dissemination of vulnerability-contributing commits. The size of the commit is calculated as either an absolute number changed lines to a source code file, the number of changes relative to the total number lines of code after the commit or the sum of code churn to the file 30 days prior to the commit. The interactive code churn metrics measures if vulnerable-contributing commits are associated with churns that affects other developers and if such commits are related to new committers to the code. Community dissemination are measures of how long a vulnerability remains in the system, how often they are part of original source code import, how often they occur in files already patched for different vulnerabilities and if they are likely to be noted in change logs and status files [12]. The result from the study partly confirms findings from [14]. The vulnerability-contributing commits were on average 608.5 lines of churns to 42.2 on non-vulnerable commits. A vulnerability-contributing commit is also on average affected by 1.78 authors to 1.01 on non-vulnerable commits, and 41.9% of the vulnerable-contributing commits was changed by new contributing authors [12]. So, large commits, many contributing authors and new authors can be indicators of vulnerable source code files. Looking at the community dissemination measures, the median number of days from a vulnerability-contributing commit to fix was 853 days. 13.5% of the vulnerability-contributing commits were in original source code and 26.6% was in known vulnerable files. 48.6% the vulnerability-contributing commits were mentioned in change logs and status files [12]. The length of the existence of vulnerabilities confirms the findings in [13] and [9],

but the findings in this study does not find original source code to be the main contributor to vulnerabilities. Instead, vulnerabilities looks to be the result of evolution of the project, though the study does not look at the influence of legacy code to the vulnerabilities [12]. That under 50% of the vulnerabilities are mentioned in change logs and status files can be an interesting point to explore in this project when exploring how vulnerabilities occur in a project.

Another vulnerability metrics is the effect of code reviews which is explored by Thompson and Wagner [11]. Working on a data set gathered from GitHub consisting of 3126 projects in 143 languages, with 489,038 issues and 382,771 pull requests, they found that the code review coverage had a significant impact on software security using a combination of quantification techniques and regression modelling. Researching the effect of code review coverage on reported issues in general and security related issues in particular, the study found a small but significant relationship between number of unreviewed pull requests and the log number of both reported issues in general and reported security issues. The study also found a small but significant relationship between the log mean number of review comments per pull requests and the number of issues in a project. Projects with higher number of review comments per pull requests tends to have fewer issues. However, the same relationship was not found between number of code review comments and security related issues. In other words, code reviews appear to reduce the number bugs in general and number of security issues or vulnerabilities in particular [11]. The code review practice and the effect of this could therefore be one area of interest in this project when studying artefacts related to vulnerabilities in a open source project.

The question of how the number of vulnerabilities in a software package evolve over time is explored in [15] and [16]. In their paper, Mitropoulos *et al.* [15] used FindBugs on every version of the Maven repository. Across projects they found no significant increase or decrease in in security issues over time, and they also found that the average lifetime of a security issue was between two and three versions. Another finding is a significant, but not always strong, correlation between categories of bugs, meaning that you do not find only certain categories of bugs in a project [15]. In [16], Edwards and Chen [16] examined historic releases of Sendmail, Postfix, Apache HTTP and OpenSSL using static source analysis and entry rate in the Common Vulnerabilities and Exposures (CVE) dictionary. They found a statistically significant correlation between the number security issues identified by the analyser and the number of occurrences in CVE. Though the rate of CVE entries in general started to drop three to five years after initial release, analysis of the issues reported by the static analyser showed that software quality not always improved with new releases. Large amount of code changes can decrease quality [16].

Munaiah *et al.* [10] studies the connection between vulnerabilities and software bugs through an analysis of the Chromium project. On the question if a source code file previously fixed for bugs is likely to be fixed for future vulnerabilities, they found a statistically significant correlation between post-release bugs

and pre-release vulnerabilities in source code files. On the other hand, there was also many counterexamples to this leading to a week overall association. They also found a week association between bugs and vulnerabilities, leading to a limited ability for bugs in a source code file to predict or indicate vulnerabilities in the file. Also, none of the source code files with highest bug density was in the files with highest vulnerability density, and source code files with the most severe vulnerabilities did not have a corresponding increase in number of bugs. The study also tested code review as a vulnerability prediction metric. On the question if a source code file reviewed by more bug review experienced developers had fewer vulnerabilities, they found limited effect of the on the occurrence of future vulnerabilities [10].

From these studies we see that vulnerabilities tends to live in a system for longer period of time, and that they to some extent tend to be introduced in initial releases of the system. The amount of legacy code also influences vulnerabilities in a software package, but to various degree in the different studies. Changes to code also introduce changes, and metrics like code complexity code churn proves to predict vulnerabilities. Other metrics like code reviews and bug counts are weaker metrics with various results in predicting vulnerable source code files.

2.1.2 Vulnerability prediction model

An area related to vulnerability measures and metrics is vulnerability prediction models. As described in section 2.1.1, [14] used code complexity, code churn and developer activity as metrics to predict vulnerable files. The motivation behind the prediction is to help prioritise which source code files to review in search for vulnerabilities. The vulnerability measures and metrics are, however, manually designed features and fails to capture semantic and syntactic features of source code [17].

Dam *et al.* [17] uses the deep learning Long Short-Term Memory (LSTM) to learn semantic and syntactic features in code and the learned features was used to predict vulnerable source code files. In their approach each source code file is split into methods with any class declaration treated as a special method, and the methods is feed into a Long Short Term Memory (LSTM) system to learn a vector representation of the system. The method vectors are then aggregated into a single feature vector with syntactic and semantic features. Syntactic features are local to project and can include method and variable names, while semantic features are general features across projects. These features are used to build and train a vulnerability predicting model. The method was evaluated on a dataset containing the source code from 18 Android application, and compared against vulnerability prediction using software metrics, Bag-of-Words (BoW) and Deep Belief Networks. The results show improvement both for within-project prediction (training and testing on same project) and cross-project prediction (training on one project, testing on another). In both scenarios, using syntactic and semantic features, either separate or joint, gave better prediction than the three benchmark

methods. Interestingly, the software metrics gave lowest performance both for within-projects and cross-project prediction [17].

In their study, Pang *et al.* [18] propose a prediction technique combining N-gram analysis and feature selection algorithm to predict vulnerable source code components. The features are continuous sequence of tokens containing N-grams of different size (1-grams like "public", "class", etc. or 2-grams like "public class", etc.). Feature ranking was used to exclude a large number of features and Support Vector Machine (SVM) was used as machine learning algorithm. The prediction method was evaluated on the source code from four Android applications. The result showed an average accuracy, precision and recall on 92.25%, 95.78% and 87.21% when the technique was applied on the four projects. When applied to a cross-project scenario the average results are 63.37%, 66.69% and 62.96% for accuracy, precision and recall. The proposed prediction technique was not benchmarked against other techniques or features [18]. Again, we see that other features than software metrics and measures can give good results in predicting vulnerable source code.

As [17] showed, there are differences in the performance between vulnerability prediction methods, and in [19] Jimenez *et al.* [19] compared vulnerability prediction methods using a dataset with all vulnerable Linux components from 2005 to 2016. The three main vulnerability prediction methods that were compared are *software metrics*, *text mining* and *inclusion and function calls*. The software metrics method uses features like the one discussed in section 2.1.1, and the features from [14] was used in this study. The main idea behind the text mining method used in the study is feature selection without any human interaction. The source code files are split into token, a vector of unigrams (2-grams) is created from the tokens, and the frequency of each unigram is calculated. A list of all unigrams present in all files are created and feature ranking is used in the feature selection. Last, random forest is used as machine learning algorithm. Though not the same, the approach has similarities to [18]. The inclusion and function call method build on an assumption that vulnerable files share similar sets of imports and function calls. A feature vector containing imports and function calls for each file are created and SVM was used as machine learning algorithm. The different methods were tested in different scenarios looking at the ability to differentiate between vulnerable and buggy files, the discriminative power in a realistic environment, and the ability to predict future vulnerability using past data. Overall, the text mining and inclusion and function calls methods performed better than the software metrics, tough the software metrics method also performed well in some scenarios to example using a realistic data set [19]. What is interesting with regards to this project inclusion and function calls method, which suggest that there are particular third-party components and API to look for when identifying vulnerability emergence and evolution.

In another study, Morrison *et al.* [20] explores the challenges with using vulnerability prediction models. They replicated a vulnerability prediction model using metrics like code complexity, code churn, code coverage and dependency met-

rics, similar to the metrics discussed in section 2.1.1, and the model was used on two versions of the Windows Operating System. The vulnerability prediction was performed both on source file level and on binary level. The binary level prediction gave a precision on 75% and recall on 20%. At the file level the result was below 50% for precision and below 20% for recall. The challenge is that a binary can consists of large number of code lines, and a manual inspection of the source code for the binary is not realistic. On the file level the workload is more realistic, but with low performance it is still questionable what is gained from the prediction [20]. We have seen that other prediction models can have better performance than software metrics, but the question of what is gained by predicting vulnerable files is still valid. For this project, the findings underscore that preventing vulnerabilities from occurring is important.

2.1.3 Source code patterns and vulnerability categories

Another research area of interest for this thesis project is source code patterns of vulnerabilities. In [21], [22] and [1], patterns of SQL injection, cross site scripting and buffer overflows are explored respectively. The papers present taxonomies for the respective vulnerabilities, and the SQL injection and cross site scripting analysis is done using open source PHP projects while the buffer overflow analysis was conducted on the Firefox web browser. The taxonomies give insight into how the vulnerabilities occur, and for SQL injections and cross site scripting the results show that missing or improper input sanitisation is a major source for these vulnerabilities. Better education and training of developers to increase knowledge of the vulnerabilities are one suggested solution, but as the results for the buffer overflow this vulnerability goes beyond critical functions and just learning simple vulnerabilities are not enough [1, 21, 22].

Another vulnerability category of interest is integer overflows. Many buffer overflow vulnerabilities are are caused by errors in processing of integers, in particular determining memory buffer sizes or memory locations. Exploiting these flaws, an attacker can cause buffer overflows, write to a selected memory location or execute arbitrary code [23]. Dietz *et al.* [24] studies integer overflow bugs in C and C++ programs. In their study they group the integer overflows into four categories of intentional and unintentional, well-defined and undefined integer overflows. In C/C++ unsigned integer overflow is defined behaviour, and intentional use of this behaviour is not vulnerable. Signed integer overflows are undefined behaviour in C/C++ and intentional and unintentional use of signed overflows gives design errors and implementation errors respectively. The study describes these as possible "*time bombs*", the implementation might work as expected in given circumstances but give unexpected results in other. Testing for integer overflow in 1172 of the top 10000 Debian packages the study found that 35% of these packages triggered an integer overflow, and 16% invoked integer overflows with undefined behaviour [24].

In their paper, Wressnegger *et al.* [23] specifically studies integer overflow vul-

nerabilities caused by migration from 32-bit to 64-bit systems. Due to changes in the width of integers and the larger address space available on 64-bit systems, codes that securely runs on 32-bit systems might be vulnerable on 64-bit systems. The study defines five patterns of 64-bit migrations issues where code behaviour changes between 32 and 64-bit systems. These are *new truncation*, *new signedness issues*, *dormant integer overflows*, *dormant signedness issues* and *unexpected behaviour of library functions*. When testing for the different patterns in 198 Debian packages and the 200 most popular C/C++ GitHub projects, the study found that the different patterns occurred between 9.58% (*dormant integer overflows*) to 68.41% (*unexpected behaviour of library functions*). In a case study, the paper also describes two 64-bit migration related vulnerabilities in the Libarchive open source software package. Libarchive is the software package studied in this thesis project and is of interest for us [23].

2.1.4 Developer-centred security

The developer's role in understanding, considering and implementing security measures is another research area of interest. One example of this is the considerations of coupled code in the complexity metrics from [14]. Highly coupled code can have input from external source code and integrating external components happens through an application programming interface (API). This can to example be challenging due to constraints and call order and wrong implementation through APIs, API misuse, is known problem in software that can lead to vulnerabilities to example due to missing parameter validation [25].

In an empirical study of API-misuse bugs by Gu *et al.* [25], 830 randomly selected API-misuse bugs from six open source programs was studied. On average 17.05% of all bugfix related commits was API-misuses related, showing that API-missus are common bugs in code and not corner-cases. The common API-misuse cases are improper parameter using, improper error handling and improper causal function calling. APIs abstracts the underlying implementation details, and certain conditions must hold whenever an API is invoked. If these preconditions, like input validation, interrelations among input variables or return values, are not meet API-misuse bugs occur. 14.29% to 19.51% of the API-misuse bugs was caused by improper parameter using. Improper error handling bugs happens when the return value from an API is not checked before proceeding. Of all the analysed API-misuse bugs in the study, improper error handling caused between 19.51% to 34.13%. Improper causal function calling caused between 27-21% and 42.54% of the API-misuse bugs and occur when the second function in a causal relationship is not called [25]. Knowing that API-misuse can lead to vulnerabilities, these types of bugs is of interest when analysing what causes vulnerabilities in this project.

The question of why developer misuse API is addressed by Oliveira *et al.* [26], referring to misunderstandings and misuse of APIs as blind spots. A study was conducted where 109 developers from four countries solved programming puzzles involving Java APIs known to contain blind spots. The results show that developers

are less likely to correctly solve puzzles with blind spots compared to puzzles without blind spots. Interestingly, the result found no support for developers technical and professional experience were associated with the ability to detect blind spots. Programmers generally trusts APIs and given that even security minded developers might miss vulnerabilities in API functions. The study also found that API blind spots particularly had an impact on puzzles involving I/O operations and more complex programming scenarios [26].

A broader perspective on API-misuse and blind spots is examined by Pieczul and Foley [27]. In their study they analyse what they refer to as *the dark side of the code*; the security gap that can exists between expected and actual behaviour in a contemporary application consisting of high-level programming languages, software frameworks and third party components. Through an example using the Java method **WebUtils.snapshot()**, creating snapshot image of a given URL, they show how this method can be exploited to access resources in the local network where the application is hosted either local webpages or customs files from the web server file system. This behaviour is not clear from documentation or source code for **WebUtis.snapshot()**, and the paper argues that the level of abstractions makes cognitive efforts to anticipate security problems much harder for developers. The complexity of today's systems introduces security gaps between the high-level expected behaviour and the actual low-level behaviour. This increases the likelihood of introducing vulnerabilities. The paper argues for using runtime verification approach to check actual behaviour against a model of expected behaviour to check for vulnerabilities [27].

Developer's blind spots is also further explored by Oliveira *et al.* [28]. in their paper investigating the hypothesis that vulnerabilities are blind spots in developer's heuristic-based decision-making process [28]. A study was conducted with 47 developers from various background where the participants were asked to answer questions about six programming scenarios not knowing that the study was security related. The results aligned with the hypothesis that security is not part of the developers heuristics in their normal programming task. With a short working memory, humans only keep a limited number of elements readily available at the time, and security seems not to among those elements. Developers tends to focus on known elements of functionality and performance. There is also an issue that developers normally assume common cases for inputs in piece of code, while the vulnerabilities lie in the uncommon cases. To find these cases requires to see through a complexity of fault analysis, and developers must use a significant cognitive effort while people normally prefer to use as little effort as possible to solve a problem. The study also found, as in [26], that developers often trust code from third party components like APIs. Another finding in the study is that if primed about the possibility of finding vulnerability, developers could change their mindset towards security [28].

Pieczul *et al.* [29] uses the expression *symmetry of ignorance* when analysing the problems in contemporary software development with increasing complexity of software layers and components, and where everyone through an open source

software project can become a developer. In this environment the developer cannot be experts in every security aspects of the software components they use, and the development of secure software becomes a challenge. Through *user-centred* security it is acknowledged that end-users are not to blame for bad security in the computer system, but in today's world the end-user can also be a developer consuming a third-party component through an API. The symmetry of ignorance exists between the developer and the end-user, where the end-user is ignorant of the implementation while the developer is ignorant of the user domain. In contemporary systems this symmetry of ignorance plays out across many stakeholders in the system. Developers are both producers and consumers of interfaces and thus both ignorant of how their own interfaces are consumed, while being ignorant of how interfaces they use are implemented. There are also other stakeholders beyond the developers and end-users, like system administrators and architects. Because of this symmetry of ignorance, the paper argue that the user-centred security should not be limited to just end-users and developers but include all producers and consumers of interfaces and that we need to recognise that there is both expertise and ignorance distributed across all stakeholders [29].

Votipka *et al.* [30] analysed results from 94 project submissions to the *Build it, break it, fix it* secure-coding competition. Vulnerabilities in the submissions were categorised into three categories, *No-implementation*, *Misunderstandings* and *Mistakes*. The result showed that No-implementation and Misunderstandings were more prevalent than Mistakes. No-implementation was used for vulnerabilities when there was no attempt to implement necessary security mechanisms, Misunderstandings was vulnerabilities caused by failed attempts of security implementations and Mistakes was used on vulnerabilities where there was an attempt on correct security implementation but there were mistakes leading to vulnerabilities. This result shows that the developers did not fully understand the security concepts. In the No-implementation category, unintuitive mistakes (to example securing against side-channel attacks) was the most common cause of the vulnerabilities. In the Misunderstandings category, conceptual errors (to example insufficient randomness) were the most common cause of vulnerabilities. This shows that even when developers try to provide security, they fail to consider all unintuitive ways to attack a system, and when security control was implemented the developers was not able to identify or understand correct usage of the security control. Complexity in the programming problem and the solution was often the source of Mistakes [30]. These findings confirm what we have discussed earlier about developers blind spot and heuristics and software complexity as causes for vulnerabilities.

In [7], Pieczul and Foley [7] analysed the evolution of security defence in the Apache Struts open source software package over a 12-year period. Trough the analysis of vulnerabilities and the code changes and other artefacts like related discussions they observed the phenomena of *dark side of the code* and developers blind-spot. The security issues in the low-level details of used components are not accessible to the developers and developers does not correlate security issues to

their current world, instead they assume common and not edge cases. The study also found opportunistic fixes in response to vulnerabilities. Instead of implementing fixes related to the root cause of the problem, developers choose fixes that are more convenient to implement and do not disrupt the existing code. Counter-intuitive fixes were another observed phenomenon. This relates to the complexity that can arise when implementing security controls. Wrong implementations of interfaces might introduce vulnerabilities [7].

This thesis uses a methodology similar to [7], analysing artefacts related to vulnerabilities. Being a qualitative research, the result might both confirm the findings in [7] or identify other answers to how vulnerabilities emerge and evolve. All the findings from the studies reviewed in this chapter is of interest in this project when analysing this question.

2.2 Vulnerability scoring

Another area of interest when studying software vulnerability is vulnerability scoring, which helps organisations categorise and priorities reported vulnerabilities. One set of vulnerability scoring specifications are the *Common Vulnerability Scoring System (CVSS)*, *Common Configuration Scoring System (CCSS)* and *Common Misuse Scoring System (CMSS)* [31]. CVSS address vulnerabilities caused by software flaws, like input validation errors. CCSS measures and scores vulnerabilities related to software configuration issues, which are use of security configuration settings that negatively affects the software security. CMSS addresses software feature misuse vulnerabilities where a software feature also provides a path to compromise security [31]. CVSS is released in several versions, where the last version is 3.1 from 2019 [32]. CVSS version 2 is from 2007 [33] but is often found used together with version 3 for compatibility reasons.

All three measurement and scoring system are organised into three groups - base, temporal, and environment metrics [31]. The base metrics measures characteristics of a vulnerability that is constant over time and a cross environments, and consists of two sets of metrics, exploitability, and impact, which measures the vulnerable and impacted components respectively. The temporal group refers to characteristics of the vulnerability that might change over time but not across user environment, to example can an easy-to-use exploit kit increase the CVSS score, and an official patch decrease the score. The environmental group looks at characteristics of the vulnerability that is unique to the user environment and includes presence of security controls that might mitigate some consequences of a successful attack. In general, base and temporal metrics are applied by application or security product vendors and environment metrics are applied by end-user organisations. The base metrics are the only mandatory metrics, while the temporal and environment metrics can be omitted [32]. The specific metrics in each group varies between the three different scoring systems, but each metric is given a score and from these scores a total vulnerability score is calculated. This score is presented together with a vector string, which is a formatted string which contains

each string assigned to each value [32].

The CVSS score is what we normally find linked to vulnerabilities in CVE databases. Between version 2 [33] and 3 [32] of the CVSS score, we find changes in changes in the base metric and environment metric group together with changes in the scoring system. In the base metric group, there is changes in version 3 to reflect whether physical access to the system is required, whether human users other than the attacker must participate in a successful attack, and there was also added a scope metric to capture if a vulnerability in one component impacts resources in other components beyond it security scope. The environment metric group in version 3 was rewritten to include the new "Modified Base Metric" score. This makes it possible for an organisation to modify the base metric to reflect differences between their systems and others [32, 33].

An alternative scoring system, Predictive Prioritisation, is presented by tenable [34]. This scoring systems assign Vulnerability Priority Rating (VPR) to a vulnerability after analysing vulnerability characteristics in seven categories. These are past threat patterns, past threat source, vulnerability metrics, vulnerability metadata, past hostility, affected vendor and exploit availability from threat intelligence. According to tenable, the VPR score gives a better foundation for prioritising vulnerabilities than traditional scoring systems like CVSS. One problem with CVSS mentioned by tenable, is that changes in the scoring criteria in the last version has increased the number of vulnerabilities rated as high or critical. This makes prioritisation harder when handling vulnerabilities [34].

Scoring systems like CVSS or Predictive Prioritisation can be useful when defining vulnerability metrics and features. For this project the scoring, and the basis for the scoring, can also be of interest when analysing a vulnerability and the handling of the vulnerability.

2.3 Introduction to Libarchive

In this thesis we study the vulnerability history of the Libarchive [2] Open-Source Software (OSS) package. The main criteria behind selecting Libarchive as our OSS package was the number of reported vulnerabilities over the last 10 years, the distribution of the vulnerabilities over this time period, and the activity in the project. These criteria and the method we used to select a suitable OSS package is described in more details in Chapter 3, and the process behind selecting Libarchive is described in Chapter 4. This section gives a brief introduction into Libarchive as a background for further reading of the thesis.

Libarchive [2] is an open source C programming library that offers read and write access to streaming archives in a variety of different archive formats. The distribution also includes *bsdtar* and *bsdcpio* which is implementation of tar and cpio using Libarchive [2]. The supported archive formats are [2]:

- TAR (read and write)
- RAR (read only)

- ISO9660 (read and write)
- ZIP (read and write)
- 7Zip (read and write)
- CAB (read only)
- MTREE (read and write)
- PAX (read and write)
- CPIO (read and write)
- SHAR (write only)
- AR (read and write)
- XAR (read and write)
- LHA/LZH (read only)
- WARC (read and write)

We find Libarchive used across different software. It is used in the operating systems FreeBSD, NetBSD, macOS and Windows, and ports of the project is used Debian and Gentoo Linux. Further, Libarchive is also used in different individual software like package managers, archiving tools and file browsers [2].

The structure of Libarchive[2] consists of several independent APIs, which can be used separately from each other. The different APIs has an object-like interface implementing a C structure reference. These objects have a similar life cycle where there is **new()** function creating the object, then different functions are invoked to configure the object, operations are performed on the object, and last the object is destroyed with **free()** or **finish()**. The configuration functions fill in function pointers in the structure. If a function is not used the pointer remains NULL and the associated coded will not be linked to the executable. This makes Libarchive usable in space constrained applications [2].

When reading an archive file Libarchive implements a bidding process, where different modules supporting different archive formats inspects the incoming data [2]. The different archive formats modules contain a bid-function with knowledge about how to recognise the given archive format. The module gives a number indicating how certain the module is that the format is recognised. When reading an archive file, the first block of data is read and presented to the bid-function in each register archive format module, and the module with the highest bid is selected and a reader for that archive format is initiated. The bidder uses a "peak ahead" functionality in the archive readers, making it possible for several bidders to inspect the incoming data simultaneously without consuming the data. In addition to validate archive format signatures, the bidder functions do a more thorough validation of to example check sums, initial header bytes being octal, etc. This reduces the number of false positives [2].

A simple example of using Libarchive is given in Code listing 2.1. The example read an archive file, either with support for all archive formats or with support for ZIP and 7Zip format specifically. The example prints the entry names of the archive file without reading the entry data.

In the example the archive read object is created in line 5 and the configuration of the filter and format support is done in line 6 to 14. The filter modules

recognise compression and encoding formats and works the same way as archive format modules, with bid-functions used to identify the correct compression and encoding [2]. In our example all filters are included. Internally, initialisation functions allocate workspaces and register additional functions, and the core reader initiates an initial filter and recursively hands the most recent filter to each available filter and format bidders in turn. The bidders use the internal read-ahead API to look at the next bytes in the stream and returns a positive bid if this is a stream that it can handle. Next, there is an alternation between reading headers and data [2]. In our example we read the header but skip reading the data, which we see in the while-loop in line 21.

Code listing 2.1: Libarchive example reading archive file entries [2]

```
1  struct archive* a;
2  struct archive_entry* entry;
3  int r;
4
5  a = archive_read_new();
6  archive_read_support_filter_all(a);
7
8  if (allFormats) {
9          archive_read_support_format_all(a);
10 }
11 else {
12         archive_read_support_format_zip(a);
13         archive_read_support_format_7zip(a);
14 }
15
16 r = archive_read_open_filename(a, archiveFilePath, 10240);
17
18 if (r != ARCHIVE_OK)
19         exit(1);
20
21 while (archive_read_next_header(a, &entry) == ARCHIVE_OK) {
22         std::cout << archive_entry_pathname(entry) << std::endl;
23         archive_read_data_skip(a);
24 }
25 r = archive_read_free(a);
26
27 if (r != ARCHIVE_OK)
28         exit(1);
29
30 std::cout << "---␣End␣of␣Archive␣---" << std::endl;
```

Chapter 3

Methodology

In this chapter we describe in detail the methods used in this thesis. The thesis project follows a qualitative approach by gathering observations from artefacts related to vulnerabilities in an OSS package to gain insight into how vulnerabilities emerge and evolve in software. This can be categorised as an exploratory study in the field of observational research. The main tasks in the project are a literature review, the identification of an OSS package to analyse, data collection of relevant artefacts related to the vulnerabilities in the OSS package, data analysis of the collected artefacts and development of a model describing the phenomena surrounding the emergence and evolution of vulnerabilities. These steps are summarised in Figure 3.1, and are described in more details in the following sections after a discussion of observational research and how this project follows this methodology.

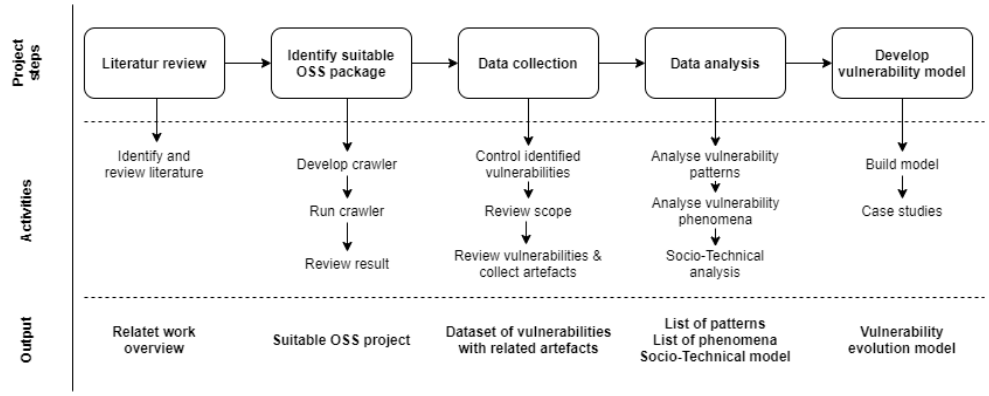

Figure 3.1: Project workflow

3.1 Observational research

Observational research is a research methodology that is suited to answer broad and open-ended research questions [35]. It is a research category covering obser-

vation of phenomena, symptoms, and systems, and in general it includes sensing and data mining in real world systems to discover interesting artefacts. Exploratory studies are a subset of observational research and objective of these studies is to get insight and understanding of the phenomenon under study. Often the scope and the data set of the study is large, and the data set is not in immediate control of the investigator or is collected after the fact [35].

This project is conducted as an exploratory study when we gather insight into how vulnerabilities emerge and evolve in an OSS package and describe the phenomena behind this evolution. These broad research questions and the goal to gain insight into vulnerability evolvement suits an exploratory study.

In this project we identify a suitable OSS package with a sufficient number of reported vulnerabilities over a 10-year period, and then collect relevant data related to the identified vulnerabilities. This is a qualitative data set containing artefacts like code updates, vulnerability reports, code reviews, developer discussions, etc. With the time frame of 10 years, we will use the longitudinal methodology which is one of several methodologies that can be used in an exploratory study [35]. The goal in such studies is to collect information over time to ensure that the entire life cycle or complete context of the system under study is considered [35]. This method fits our project when we study how vulnerabilities emerge and evolve in a software package as this package mature over time.

3.2 Literature review

To gain knowledge into the existing work around software vulnerability studies we perform a literature review as part of our project, and focus on studies into vulnerability measures and metrics, vulnerability prediction methods, studies into source code patterns and vulnerability categorisation, and studies around developer-centered security. We started the literature review during the work on our project plan report, and we will build on that work and expand the review with new literature if needed as we proceed with our project. The literature review is presented in Chapter 2.

3.3 Identify suitable open source software package

The main criteria when selecting an OSS package is the number of vulnerabilities and the lifetime of the project. The selected OSS package must have at least 10 to 20 vulnerabilities over the past 10 years. This criteria is in line with what was used in [7] where security evolution in the Apache Struts open source package was analysed over a period of 10 years with a total of 20 identified key security related updates [7]. With a time-frame of 10 years and at least 10 and 20 vulnerabilities we will be able to analyse how security related issues are handled and how vulnerabilities emerge and evolve as the software package mature over time. Other parameters when selecting the OSS package is:

- Distribution of the vulnerability over the 10-year period.
- Activity in the open source project both terms of developer involvement and usage of the software package.
- The programming language and domain of the software package.

The evaluation of these parameters is a manual and objective assessment based on the projects identified after the first criteria of number of vulnerabilities over the past 10 years. Our selected OSS package will be a project where the reported vulnerabilities are fairly distributed over the 10-year period and where there are sufficient current activity in both development and usage of the software package. The last parameter of programming language and domain will be assessed against our own knowledge in the area.

To identify an OSS package with an extensive number of vulnerabilities we focus on vulnerabilities tracked in the CVE database [36] and follow an approach similar to the one described in [21] with adjustments to fit our project. In [21] a crawler is created to retrieve SQL injection vulnerabilities. For each entry in the CVE database the CVE Details [37] record is retrieved. This record provides information about vulnerability categories and possible links to GitHub commits which can be used to determine programming language [21].

In contrast to the crawler described in [21] we are not interested in identifying one specific category of vulnerabilities, but the number of vulnerabilities reported in different OSS packages over the last 10 years. The crawler therefore only counts CVE entries from 2009 or later based on the year part of the CVE ID. The crawler supplements each CVE entry with additional information from CVE Details [37] and then look for GitHub URLs in references on the entry. These URLs will be used to identify open source projects through the project owner and project name in the GitHub URL. Checks are added to the crawler to handle the potential case of GitHub references to more than one open source project on a single CVE entries. If such cases are found they are reported separately. For each identified open source project, the crawler retrieve the project description and the list of used programming languages from GitHub, and present a list off all identified open source projects with more than 10 CVE entries since 2009 or later.

From this list we do a manual review of the open source projects based on the additional criteria listed above and look at the distribution of the vulnerabilities over the time period since 2009, the activity and usage of the OSS package, and the main programming language and domain of the software package. From this review we select a suitable open source package to analyse in this thesis project.

3.4 Data collection and analysis

When a suitable OSS package is identified and selected, relevant artefacts related to the vulnerabilities are collected. We first do a manual control of the vulnerabilities identified by the crawler. This list contains all vulnerabilities with GitHub references against the OSS package. We perform a manual search for the selected

OSS in the CVE database [36] to identify any unreferenced vulnerabilities that should be included in our data set. Depending on the selected software package and the number of identified vulnerabilities it might be necessary to limit the number of vulnerabilities to include in this project. If needed, we will do this by limiting the scope to relevant portion of the software or groups of related vulnerabilities.

The relevant artefacts related to the vulnerabilities are the vulnerability report, code updates, bug reports, posts in discussion forums, blog posts, news articles, etc. In [7], where a qualitative approach similar to this project was used, they reviewed vulnerability-publications, code-updates, related discussions on the development mailing list and other publications often contributed by the vulnerability report [7]. From the identification of the OSS package described in section 3.3 we have the relevant vulnerability reports and this will form the basis for collecting other relevant artefacts. Through the GitHub references on the CVE entries, we know that the OSS package use GitHub [38] for source code hosting and is where we will find the source code with version and commit history. GitHub also provides an issue tracking system where bugs and other issues are reported, but projects might choose to use other tracking systems. There might also be other open sources like discussion forums with relevant data related to the development of the software. The data collection is manual task where we from the basis of the reported vulnerabilities collect source code with commit history leading up to the vulnerability together with other relevant artefacts related to the changes in the source code. We then collect changes to the source code in response to the vulnerability together with other artefacts including the vulnerability report. The collected data is then analysed to try to understand what choices leading to the vulnerability being introduced to the code and what choice were made in response to the vulnerability. This process is repeated for the selected portion of vulnerabilities, and we will identify common patterns and behind errors and fixes to the vulnerabilities and try to identify general phenomena explaining how the vulnerabilities emerged in the code. In [7], the analysis is summarised in aggregated changes over releases that resulted in a published security release. This gives more reliable understanding of the developers intentions than possible incomplete changes between security releases [7]. We follow the same approach.

This artefact-first approach is described as an archaeological method in [29]. By studying layers of artefacts over time and reconstructing progression to actual time we can gain insight into what and how developer activities impacted the security over time. We can also gain insight into how software components were used, or for what purpose they were built, by studying parameters passed to the component, etc. [29].

In addition to analyse the collected artefacts related to the vulnerabilities, we also analyse the Socio-Technical System (STS) surrounding vulnerabilities in the OSS package. An OSS project consists of both a community of practices, a social culture, technical practices, processes, and an organisational structure, and is therefore broader than just a technical definition [39]. When trying to answer the question of how vulnerabilities emerge and evolve by studying the vulnerab-

ility history in an OSS package, we also need to understand both the technical and social aspects of OSS development [39], and a STS analysis will help us gain this knowledge. With this knowledge we will be able to explain how the identified vulnerability patterns an phenomena occur by identifying elements in the STS that influence them, and this will give us a broader understanding of vulnerability evolvement. We therefore adopt a STS model developed by Kowalski [40] in this project, and use this to summarise and categorise our findings. The STS model is depicted in Figure 3.2.

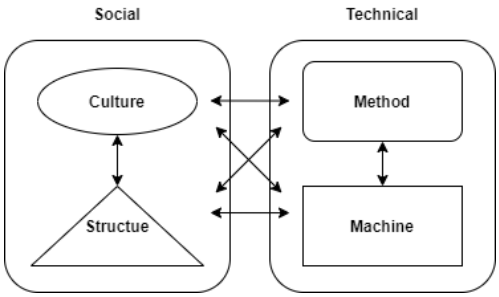

Figure 3.2: Socio-Technical system [40]

The STS model is divided into two subsystems, social and technical, which each consists of further subsystems [41]. In the social subsystem we find a cultural and a structural subsystem, and in the technical subsystem we find methods and machines. The inter-dependencies between the elements forms a homeostasis state, and changes makes the system adjust to maintain equilibrium [41]. By using the STS model, we get an structural analysis of the relationships between the different elements within the social technical subsystems [39]. We use information from the collected artefacts related to the vulnerabilities together with collected information regarding the OSS project like coding guidelines, vulnerability disclosure procedures, etc. in this analysis. Together, the analysis of the vulnerability history and the STS help us answer the question of how vulnerabilities emerge and evolve in an OSS package.

3.5 Vulnerability model

From the analysis of the artefacts related to the vulnerabilities in the selected OSS package and the STS surrounding the vulnerability handling in this package, we develop a model describing the phenomena behind the emergence and evolution of vulnerabilities. The model will be built from identified common patterns and phenomena in our analysis, and from our analysis of the STS surrounding the vulnerability handling, and we will also use the results from the related work discussed in Section 2.1 in the development.

The developed model will answer our research question of how vulnerabilities emerge and evolve in an OSS package, and be a tool that can be used in a soft-

ware project to improve the SDLC to improve the software security by preventing vulnerabilities to emerge in the code.

To test our developed model, we perform a case study into two additional OSS packages, by applying our developed model on the vulnerability history in these two projects. The projects used in this case study are selected from our list of potential suitable OSS packages identified by our crawler described in Section 3.3.

Chapter 4

Results

In this chapter we present the results from our thesis project. We begin with the identification of Libarchive as our suitable OSS package for vulnerability analysis. We then describe the data collection and gives an overview of the Libarchive vulnerability history, before we present the results from the analysis of vulnerability patterns, identified vulnerability phenomena and the result from the STS analysis.

4.1 Selecting an open source software package

To identify and selecting a suitable OSS package for our project we used the crawler as described in Section 3.3. The data source for crawler was the CVE entries database as of 26.07.2020 [36]. Our main criteria when selecting the OSS package was for the package to have a total of at least 10 to 20 vulnerabilities over the last 10 years. In the crawler we set the cap year to 2009 based on the year part in the CVE ID. This returned a total of 139909 CVE entries, and through the GitHub reference in these entries we identified 4222 OSS packages. Returning only OSS packages with more than 10 CVE entries we found a total of 216 packages. The number of CVE entries in these OSS packages varied from 10 to 962. No CVE entries was reported to have GitHub references to more than one OSS package.

From the list of 216 OSS packages with more than 10 vulnerabilities since 2009 we did a manual review of the different project based on the other criteria listed in Section 3.3 and looked at the distribution of the vulnerabilities in the period since 2009, the activity in the projects and the type of projects. The project with most vulnerabilities, 962, was the Linux kernel and was assessed to be too extensive for our project and not considered further. The review also found several GitHub repositories used to track vulnerabilities in other OSS projects. These were excluded from further review together with projects with little or no resent activity in the project and projects with little distribution in time of the vulnerabilities. This gave a list of 28 possible OSS packages that could be suitable for our thesis project. These projects are listed in Table 4.1.

From this list we choose the Libarchive C library as our OSS package for further vulnerability analysis in this thesis project. Of the CVE entries counted by our

Table 4.1: Possible open source software packages for further analysis

Project	Time period	No. CVE
ImageMagic	2014 - 2020	437
FFmpeg	2011 - 2020	81
Bento4	2017 - 2019	57
Libarchive	2013 - 2020	52
Exiv2	2017 - 2019	50
OpenJPEG	2015 - 2020	45
Jackson Databind	2017 - 2020	45
GPAC	2018 - 2020	39
LibRAW	2013 - 2020	34
ExponentCMS	2016 - 2017	34
LimeSurvey	2014 - 2020	33
OpenEMR	2014 - 2019	31
Ansible	2013 - 2020	30
Jenkins	2013 - 2017	28
Piwigo	2016 - 2020	28
PrestaShop	2018 - 2020	28
TeamPass	2012 - 2020	25
Symfony	2016 - 2020	20
Libsndfile	2014 - 2019	18
Pillow	2014 - 2020	17
YARA	2016 - 2019	16
OpenEXR	2009 - 2018	15
Squid	2018 - 2020	15
DokuWiki	2012 - 2018	14
Foreman	2012 - 2019	14
Webadmin	2011 - 2019	11
Salt	2013 - 2020	11
YANG	2019 - 2019	10

crawler Libarchive have 52 vulnerabilities distributed between 2013 and 2020, which is a good distribution in the selected time period and above our minimum of 10 to 20 vulnerabilities. We also found that the project is active with new releases of the library up to first half of 2020, and it is found used both in operating systems and in other software packages. Last, we also find the C programming language and the domain of file archive to be within scoop of our project. This make Libarchive a suitable project for further analysis of the vulnerability history within an OSS package. An introduction to Libarchive is found in Section 2.3.

4.2 Scope and data collection

The result from the crawler used to identify suitable OSS projects returned a total of 52 Libarchive vulnerabilities from 2009 or later. To control this result we performed manual search for the term "libarchive" in the CVE database[36]. This search returned a total of 73 vulnerabilities. In addition to the 52 Libarchive vulnerabilities returned by the crawler the search returned:

- 14 vulnerabilities reported in other software where Libarchive was the vulnerable component. The scope in this thesis project is the vulnerability history in an OSS package and we decide to exclude these 14 vulnerabilities and keep the focus on CVE entries directly linked to Libarchive.
- Seven vulnerabilities directly linked to Libarchive but with no GitHub references. Of these seven did three date to before our cap year of 2009 and is excluded in the further analysis.

After a review of the remaining vulnerabilities another three were excluded from the list due to being disputed or only related to development code. These three are:

- CVE-2010-4666
- CVE-2011-1779
- CVE-2019-11463

This gives a list of 53 Libarchive vulnerabilities which form the data set in this thesis project. The number of vulnerabilities is above our minimum requirement of 10 to 20 vulnerabilities, and the list is well distributed over a time period from 2011 to 2020. The total list of the 53 Libarchive vulnerabilities is found in Table A.1 in Appendix A.

The first Libarchive release containing fixes to any of the vulnerabilities in our data set is Libarchive version 3.0.0a. The last release before that is Libarchive version 2.8.5 and this release is set as the initial Libarchive version in our analysis. Version 2.8.5, released in September 2011, is the last release in Libarchive 2 before the first release of Libarchive 3 in November 2011 with Libarchive version 3.0.0a. The release of Libarchive 3 introduced breaking changes in the library API, and new archive format support is added in the subsequent version 3 releases [2]. By analysing code updates from Libarchive version 2.8.5 and onward we will gain

insight into how vulnerabilities emerge, evolve, and are fixed in Libarchive and use this knowledge to answer our research question.

For each of the 53 vulnerabilities in our data set relevant artefacts were collected. These are:

- Vulnerability introducing and vulnerability fixing GitHub commit(s). This gives insight into how the vulnerabilities were introduced into the code, what choices were made during the code updates, and why they were made. To example fitting a code update into existing code. Alternatively, we used the the first Libarchive version containing the vulnerability as our vulnerability introducing code change if we are unable to identify the vulnerability introducing commit.
- Issues and pull requests from the GitHub issue tracking system related to the vulnerabilities. These artefacts give information about how the vulnerability was discovered, to example testing tools or techniques, and how it effects the code. We also find discussions around the code updates in these artefacts.
- Discussions from discussion forum. Libarchive have an own discussion forum, and for some of the vulnerabilities we found discussions in this forum. We also collected general discussion around error handling and security testing.
- Blog posts related to the vulnerabilities. These are writ-ups from the researchers that discovered the vulnerabilities and give more detailed information around the vulnerable code and how this could be exploited.
- News articles. For some of the vulnerabilities we found news articles that gave some more information around the vulnerabilities.

A total of 100+ code changes were analysed to track the vulnerability history in Libarchive. In addition, a total of 83 artefacts related to issues, pull requests, discussions, blog posts and news articles were collected.

4.3 Tracking Libarchive vulnerability history

The 53 Libarchive vulnerabilities were introduced and fixed in 13 releases from Libarchive version 2.8.5 in September 2011 to version 3.4.2 in February 2020. Version 2.8.5 is the initial Libarchive release used in this analysis as describe in Section 4.2. The timeline is summarised in Table 4.2 and the full version is found in Table B.1 in Appendix B. The rest of the section describes the Libarchive releases and the vulnerability history in more details.

4.3.1 Libarchive release September 2011

Libarchive 2.8.5 was released in September 2011 and 24 of the 53 vulnerabilities in our data set is present in this release. These 24 vulnerabilities relate to 8 different archive formats in addition to vulnerabilities found in general or common code used across archive formats. In these vulnerabilities we find out-of-bound

Table 4.2: Libarchive vulnerability timeline summary

Version	Date	No. intr.	No. fixed	Comment
2.8.5	Sept. 2011	24	0	Initial release
3.0.0a	Nov. 2011	15	2	New archive formats RAR, CAB, LHA
3.0.1b	Nov. 2011	4	0	New archive format 7Zip
3.1.0	Jan. 2013	3	0	Release of multi-volume RAR support
3.1.900a	Feb. 2016	2	22	New archive format WARC. Fixes to vulnerabilities from versions 2.8.5, 3.0.0a, 3.0.1b and 3.1.0.
3.2.0	Apr. 2016	0	1	Fix of vulnerability from version 3.1.0
3.2.1	June 2016	0	7	Fixes of vulnerabilities from versions 2.8.5, 3.0.0a, 3.0.1b and 3.1.9a
3.2.2	Oct. 2016	0	4	Fixes of vulnerabilities from version 2.8.5, 3.0.0a and 3.0.1b
3.3.0	Feb. 2017	1	4	New NFSv4 ACL support. Fixes vulnerabilities from version 2.8.5 and 3.0.0a
3.3.3	Apr. 2019	0	4	Fixes vulnerabilities from versions 2.8.5 and 3.0.0a
3.4.0	June 2019	1	7	New archive format RAR v5. Fixes vulnerabilities from versions 2.8.5, 3.0.0a, 3.0.1b, 3.1.900a and 3.3.0
3.4.1	Dec. 2019	0	1	Fixes vulnerability from version 2.8.5
3.4.2	Feb. 2020	0	1	Fixes vulnerability from version 3.4.0

reads, NULL pointer dereferences, integer overflows, infinite loops, missing input validations, and directory traversal and sandbox evasions. The out-of-bound reads in large parts happens on string operations related to archive entry file names and archive entry paths and either happens on array operations on the string buffer or within functions like **memcpy** and **memmove**. The NULL pointer dereferences are also related to file name and file path operations, and both these and the out-of-bound reads are at large part caused by missing validations. There is also one missing input validation that causes an illegal left shift operation on a compression parameter in an archive file. One of the infinite loop vulnerabilities are also related to file name parsing in ISO9660 archives and occur if a directory in the archive entry path is a member of itself. None of the existing input and sanity checks catches this special case. The other infinite loop also relates to reading ISO9660 archive files and occur when rock-ridge extensions are missing. In this case the method returned status "ARCHIVE_OK" causing an infinite retry in the calling method without moving the pointer. The integer overflows are either caused by implicit cast or caused by 32-bit/64-bit issues. One of the directory traversal vulnerabilities is the *bsdcpio* implementation where absolute paths are not rejected and allows for writing of arbitrary files. Another directory traversal or sandbox evasion vulnerability relates to archive write code where non-zero sized hard-links could lead to writing of arbitrary code.

Last, there are two vulnerabilities present in version 2.8.5 or earlier where the vulnerability reports describe buffer overflows in TAR and ISO9660 related code causing denial of service or application crash. Our analysis finds that these two vulnerabilities are caused by the use of **exit()** in error handling in the Libarchive library, causing termination of the calling process. We find that this error handling routine was used after errors in memory allocation in both TAR and ISO9660 related code but also in other parts of the code base in version 2.8.5.

4.3.2 Libarchive release November 2011

In November 2011 Libarchive 3.0.0a and 3.0.1b were released. These versions were test releases before the release of version 3.0.2 in December 2011. Version 3.0.0a added fixes to the error handling related vulnerabilities described above and moved away from using **exit()** on errors and terminating the process. The implemented fix returns error codes and makes it possible to continue processing of the next item or gracefully abort processing without full process termination. This version also added support for the archive formats RAR, CAB and LHA, and in version 3.0.1b 7Zip archive support was added. This introduced 15 vulnerabilities in code related to these archive formats. In addition, four vulnerabilities were introduced together with new functionality in the ZIP, MTREE and ISO9660 archive format support. In these new vulnerabilities we find out-of-bound reads, NULL pointer dereferences, integer overflows and missing input checks or validations leading to different vulnerable behaviour. Two examples are a double free of memory caused by **realloc** with size zero before **free** on the same memory buffer,

and a memory allocation of size zero leading to buffer overflow. Both of these are in RAR-related code. The out-bound read and NULL pointer dereference vulnerabilities are also caused by missing or insufficient input validation and related to malformed archive files, to example archive files reporting zero or negative file sizes. One of the integer overflow vulnerabilities relates to 7Zip archive files and an integer variable could be overflowed in in a calculation using a crafted archive file containing especially large values. This later causes buffer overflows due to the overflowed integer. Another integer overflow relates to writing ISO9660 archive files. In processing of large file names, an explicit cast from size_t to int causes overflows on platforms where *sizeof(int) < sizeof(size_t)*. The overflowed integer is later used in memory allocation leading to buffer overflows and possible writing of arbitrary code. Though the cast from size_t to int is a problem on 32-bit platforms, this vulnerability requires more than 20GB of memory to be exploited and needs in practice a 64-bit platform. Last, we also find an integer overflow where there is a deliberate use of an overflow to define time max/min values on platforms where these are not defined. Since signed integer overflows are undefined behaviour in C, this is a vulnerable behaviour.

4.3.3 Libarchive release January 2013

Libarchive version 3.1.0 was released in January 2013. In this version three new vulnerabilities were introduced. These are related to the RAR, ZIP and MTREE archive formats. Support for multi-volume RAR archive files, where one archive is split across several files, was added in this release. The RAR vulnerability is a use-after-free bug that occur when a special crafted single-volume RAR archive is interpreted as a multi-volume archive. In this case a pointer to the ppmd7 decoder is wrongly freed causing the use after free vulnerability. The ZIP vulnerability was added with support for macOS metadata entry files. When processing an uncompressed file, the compressed size was used in memory allocation while the uncompressed size was written to the memory buffer. No input validation was implemented, and these size fields are controlled by the user and could be exploited by an attacker to cause a buffer overflow. The MTREE vulnerability is an out of bound read introduced when adding support for NetBSD MTREE archive files. When parsing archive entries identifying entry file names a read beyond the string buffer could occur if the file name was the whole entry line.

4.3.4 Libarchive release February 2016

The next release containing security fixes or introducing new vulnerabilities is Libarchive version 3.1.900a in February 2016. This was a test release of Libarchive version 3.2.0 released in April 2016. The release added support for the WARC archive format. A vulnerability in the WARC decoder could be exploited to cause a semi-infinite loop using crafted archive file with a large content length and only a few hundred bytes of data. In the MTREE format code, a revised parsing logic

was added in this version, and a misplaced array length check caused off-by-one vulnerability when reading archive arguments.

A total of 22 security patches were included in this version, and 13 of these relates to vulnerabilities present in version 2.8.5 or earlier. Several of the out of bound and NULL pointer dereference vulnerabilities were fixed by adding validation either through size checks or checks for NULL or empty file names in string operations. In addition, one out of bound read caused by an overlapping **memcpy** operation was fixed by changing to **memmove** which is safe for overlapping buffers. One of the of the out of bound vulnerabilities was in the *bsdtar* implementation and was caused by NULL or empty file names returned by the readers processing the given archive format. This was fixed by adding checks on the returned file names and skip further processing if the file name was missing. Two CVE items reported this vulnerability using crafted RAR and CAB archive files respectively. Though these archive formats were added to Libarchive after version 2.8.5, the vulnerable code could have been exploited with other archive formats. If we look at the discussion in the issues tracking these two vulnerabilities, we see that the return of NULL or empty file names are caused by underlying issues in the RAR and CAB code. In the RAR case this relates to the issues where a crafted single-volume RAR file are wrongly interpreted as a multi-volume archive file, and in the CAB case there are issues in the CAB header that cause the reader to wrongly return empty file names. None of these underlying issues are fixed and the fix of input check in *bsdtar* is deliberately chosen as sufficient to fix this particular vulnerability.

In addition to the out of bound vulnerabilities from version 2.8.5 one integer overflow and one illegal left shift vulnerability from version 2.8.5 was fixed in this version. The integer overflow vulnerability related to a signedness issue in ZIP write functionality running on 64-bit platforms. This was fixed by adding an input check against INT_MAX in *archive_write.c*. This also prevents the integer overflow writing other archive formats. The left-shift vulnerability was exploited through an invalid compression parameter. Validations was added on the parameter to verify the size and prevent the illegal left shift. In addition, there were also added other checks to reject malformed compression data. From the code history we see that these previously missing checks were a known weakness. The previous code had the comment `"TODO: verify more"`, now replaced by new input validations. Last, a directory traversal vulnerability in *bsdcpio* and the infinite loop in the ISO9660 directory parser causes by self-owned directories was fixed. To fix the directory traversal vulnerability a flag was added to rejects absolute paths in archive entries. The flag is set by default. The infinite loop vulnerability was fixed both with a sanity check to reject self-owned directories, and with a path depth counter in the parser. If an archive entry path reaches a depth of 1000 directories the archive entry is rejected.

Of the vulnerabilities introduced in version 3.0.0a, four were fixed in this release. One was a NULL pointer dereference vulnerability in RAR archive functionality. A none-NULL value indicates a filled compression buffer, which could be

exploited to read buffer values from the last archive entry using a crafted archive file. This was fixed by setting the pointer NULL before starting the processing. Another vulnerability fixed was the deliberate integer overflow used to determine time MIN/MAX values if not defined. This was solved by assuming time variables are integer and simply using INT_MIN/INT_MAX as time MAX/MIN. Also, an integer overflow related vulnerability where a size field was read as signed number and then used as an offset was fixed by masking the size field as an unsigned number. Last an out of bound read in parsing of LHA archive entry names was fixed by adding validation of the first byte in path name being NULL.

There are also two vulnerabilities introduced in version 3.0.1b and 3.1.0 that were fixed in this release. One was a NULL pointer dereference vulnerability in 7Zip functionality, which was fixed by adding validations on the input archive formats rejecting illegal or malformed archive files. The other was an out of bound read in the MTREE parsing logic caused by a file name filling the whole line in the archive entry. The parsing logic read the line from end to beginning to identify the file name. In the cases where the file used the whole line this caused the logic to read outside the buffer. The fix reversed the parsing to start from the beginning of the line.

Last, there were three fixes to vulnerabilities that were never introduced in any previous releases. They were added to the code between the release of Libarchive version 3.1.2 in February 2013 and the next release being version 3.1.900a in February 2016. One is an integer overflow in TAR archive format code exploited with a crafted TAR archive file with large sparse entries. This was fixed by adding a check on the calculation against INT64_MAX. Another was a memory leak caused by a misplaced cleanup routine after refactoring the *archive_read_extract.c* and *archive_read_extract2.c* files. This was corrected in this release. The last fix related to WinZIP AES functionality. A missing input check caused a buffer overflow if the entry was too small for the encryption header. Input checks were added to fix this vulnerability and reject malformed entries.

4.3.5 Libarchive release April 2016

In the release of Libarchive version 3.2.0 there was one security patch included. This related to the out of bound read in uncompressed ZIP archive maxOS metadata files. This was caused by different compressed and uncompressed size values in an uncompressed file, and the fix applied different input controls and checks to mitigate the vulnerability. Uncompressed files with different compressed and uncompressed sizes are rejected, and there was added checks of the total compressed size and against both sizes before writing to the buffer.

4.3.6 Libarchive release June 2016

The release of Libarchive version 3.2.1 in June 2016 included seven security patches. Two of these applied to vulnerabilities in version 2.8.5 or earlier. One was an integer overflow in ISO9660 archive files, where a calculation of a size

value defaulted to int causing the overflow. This was fixed by changing the variables to int64. The other fix corrected a memory allocation error in CPIO archive functionality. This was caused by large symlink sizes and the fix rejects symlinks larger than 1MB.

Three of the remaining fixes applied to vulnerabilities introduced in version 3.0.0a. Two of these related to RAR archive functionality, where the first was an out of bound read in a **memcpy** operation. This could be exploited using a crafted RAR archive file manipulating size fields in the file. The fix added input checks to mitigate the vulnerability. The other RAR fix applied to heap buffer overflow caused by a zero-sized memory allocation. This could be exploited using crafted archive files and the vulnerability caused a buffer overflow and possible arbitrary code execution. The fix added checks to reject the zero-size memory allocation, and also added input checks in subsequent operations to reject values below given minimum vales. The last fix related to vulnerabilities from version 3.0.0a is the integer overflow in ISO9660 archive functionality caused by explicit cast in processing large file name. The fix removed the explicit cast and also added validation on each variable in the calculation, rejecting archive entries where any variable exceeded the maximum.

The last two fixes relate to vulnerabilities introduced in the releases of versions 3.0.1b and 3.1.9a. The vulnerability from 3.0.1b was an integer overflow in 7Zip archive code where a crafted archive file with large sub-stream sizes could be used to exploit the integer overflow to create an subsequent heap buffer overflow. A check exits on each sub-stream against the constant UMAX_ENTRY, but not on the summarised total of sub-streams. This check was applied in the fix. The fix to the 3.1.9b vulnerability relates to a of-by-one bug in MTREE archive code. A misplaced input check caused this vulnerability and fix correct this mistake.

4.3.7 Libarchive release October 2016

In Libarchive version 3.2.2 release in October 2016, four earlier vulnerabilities were fixed. One of these was a buffer overflow in *tar/util.c* present in version 2.8.5 or earlier where printing of file names overflowed a buffer if multi-byte characters was included in the file name. An input validation existed but the buffer size was increased to handle this scenario. Two other of the fixes patched vulnerabilities present since version 3.0.0a. One was an out of bound read when parsing multiple long lines in the archive file. An error in the calculation could cause a read outside of the string buffer. The calculation was fix to mitigate the error. The other was a fixed of a sandbox evasion vulnerability. Hard-links of non-zero sizes was mishandled and could be exploited to write arbitrary code. The fix adds check to the links to mitigate the error. The last fix was of an out of bound read in 7Zip archive files. This could be exploited with a crafted archive file containing multiple empty streams, and the fix added checks and reject files with multiple empty streams attributes. The fix also added validations on other archive file attributes to reject malformed 7Zip archives.

4.3.8 Libarchive release February 2017

Libarchive version 3.3.0 was released February 2017. This version included extended NFSv4 ACL support, and a refactoring of the *archive_acl_from_text_l()* method in *archive_acl.c* caused a NULL pointer dereferece vulnerability. The release also fixed four vulnerabilities from earlier releases. One was another NULL pointer dereference vulnerability from version 2.8.5 or earlier, in the *archive_string.c* source code file. An validations was added to the method *archive_strncat_l()* to prevent the NULL pointer in subsequent processing. The three other fixes were related to out of bound reads in CAB and LHA archive files present since version 3.0.0a. In the case of the CAB vulnerabilities, a size parameter was changed to a static value to correct the vulnerability. The LHA vulnerability was caused by a negative size value in the archive file, and validation was implemented to prevent the out of bound read.

4.3.9 Libarchive release Abril 2019

Four other vulnerabilities were patched in version 3.3.3 released April 2017. Two of these were fixes out of bound reads in XAR and ISO9660 archive files from version 2.8.5 or earlier. Both of these related to archive entry name processing and input checks were added to either reject empty file names or validate directory sizes. The remaining two vulnerabilities were fixes in RAR and LHA archive functionality from version 3.0.0a. The RAR vulnerability was an of-by-one error due to UTF16 characters in file names and the fix added validations on this. The LHA fix was of another negative size value in archive files causing out of bound reads, and validations was added to reject archives with such values.

4.3.10 Libarchive release June 2019

Libarchive version 3.4.0 was released June 2019 and added support for RAR version 5. Due to missing input validation of the header size in RAR5 archive files an archive file with header size like zero would cause a segmentation fault. In addition, security patches for seven vulnerabilities were included in this version. These related to vulnerabilities present in version 2.8.5 and onward. From version 2.8.5 the infinite loop in ISO9660 archive functionality was fixed. The loop was caused by a failed sanity check returning status ARCHIVE_OK and fixed changed this return status to ARCHIVE_WARN on failed checks. Related to RAR archive files one double free and one use after free vulnerability from version 3.0.0a was fixed. The double free vulnerability was caused a **realloc** with size zero followed by a **free**, resulting in the double free of the buffer. This was fixed with input validation of the size before calling **realloc**. The use after free vulnerability was caused by a missing re-initialisation of parameters after errors in parsing of the last archive header. This caused a missing allocation of a buffer. The re-initialisation was fixed in this version. Another fixed RAR vulnerability relates to mulit-volume RAR support introduced in version 3.1.0. This was also a use after free vulnerability and

was caused by a crafted single-volume RAR file being handled as multi-volume RAR file. A buffer in the ppmd7 is prematurely freed in this situation, causing the use after free bug in subsequent processing. Improved validation of multi-volume RAR files was added to fix the vulnerability. The last three fixes related to 7Zip and WARC archive files and to processing of archive ACL strings. The 7Zip vulnerability was a combination of performance optimisation and a malformed archive file not returning the presumed minimum size. This was fixed by always using the minimum size in the buffer operations. The WARC vulnerability was an invalid size field causing a semi-infinite loop in the decoder reading the archive file. Data was not consumed in a scenario with a large content length and little actual data. The logic to consume previously read data was improved to fix the vulnerability. The last vulnerability fixed was a NULL pointer dereference which could be exploited with an archive file containing a malformed ACL field. This was fixed by implementing input validation of the ACL field.

4.3.11 Libarchive release December 2019 and January 2020

The last two releases containing security patches to the previous vulnerabilities are Libarchive version 3.4.1 released in December 2019 and 3.4.2 from February 2020. In version 3.4.1 a vulnerability in *archive_string.c* present since version 2.8.5 was fixed. Due to an inconsistency in use of a size parameter an out of bound read could occur in a string append method. The logic is corrected in the fix to use correct parameters. In version 3.4.2, The RAR version 5 vulnerability caused by malformed archive headers is fixed. Input validations are implemented to reject zero sized headers and headers whit sizes less than legal minimum size.

4.3.12 Summary

From tracking the Libarchive vulnerabilities we do the following observations:

- We find vulnerabilities in code related to 13 archive formats.
- We find vulnerabilities introduced together with support for new archive formats. To example with support for RAR archive in version 3.0.0a and 7Zip in version 3.0.1b.
- The majority of the vulnerabilities are memory related, either through integer overflows, buffer overflows, or pointer issues like NULL pointer dereferences, use-after-free or double-free vulnerabilities. Other vulnerabilities are variations of infinite loops and directory traversals.
- The implemented fixes centers around new or improved input checks or return value checks.

4.4 Vulnerability categorisation and patterns

From our review of the Libarchive vulnerability history in Section 4.3 we see that the majority of the vulnerabilities center around buffer and integer overflows, im-

Table 4.3: Vulnerabilities per CWE ID

CWE ID	CWE name	No. vuln.
CWE-19	Data Processing Errors	1
CWE-20	Improper Input Validation	7
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	1
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	11
CWE-125	Out-of-bounds Read	17
CWE-189	Numeric Errors	1
CWE-190	Integer Overflow or Wraparound	5
CWE-193	Off-by-one Error	1
CWE-399	Resource Management Errors	1
CWE-415	Double Free	1
CWE-416	Use After Free	2
CWE-476	NULL Pointer Dereference	6
CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	1

proper or missing input validations, and NULL pointer dereferences. This is also confirmed when we summarise the Common Weakness Enumeration (CWE) categories from the 53 vulnerabilities in our data set, listed in Table 4.3. A total of 46 of the vulnerabilities falls into these categories (CWE-20, CWE-119, CWE-125, and CWE-476). There are relationships between some these categories, where CWE-119 is the parent of CWE-125 and also can follow CWE-20 and CWE-190 [42]. In addition, CWE-476, CWE-415 and CWE-416 are memory access vulnerabilities, and as for the buffer overflow related vulnerabilities these vulnerabilities could cause illegal memory access and possible arbitrary code execution [42].

To categorise the vulnerabilities further and gain insight into patterns behind the vulnerabilities we use the taxonomies of errors, sinks and fixes for buffer overflow vulnerabilities by Schuckert *et al.* [1]. The taxonomy was developed after a review of 50 randomly selected buffer overflow vulnerabilities from Firefox, and the goal was to create a categorisation of buffer overflow vulnerabilities from the developers point of the view. In the review of the 50 vulnerabilities the types of errors, involved sinks and patches of the vulnerabilities were considered, and categories were created from these results [1]. From our data set we see that the NULL pointer dereference, Double-Free and Use-After-Free vulnerabilities could be categorised using the same taxonomy, expanding this from a buffer overflow to a memory safety taxonomy. 46 of our 53 Libarchive vulnerabilities are included in this categorisation, and the following sections lists the results. The remaining seven vulnerabilities not included in the taxonomy are listed in Section 4.4.4.

Table 4.4: Types of memory safety errors. Extensions to [1] listed in *italic*

Main category	Subcategory	No.
Variable overflow	Variable overflow allocation	2
	Variable overflow in check	1
	Variable overflow in memory access	1
	Intentional signed overflow	1
Unexpected input	Unexpected input zero	5
	Unexpected input negative	2
	Unexpected input minimum	4
	Unexpected input maximum	4
	Unexpected input NULL	5
Mismatching data types	Unsigned-signed	1
	32-64 bits	4
	int-long	0
Missing return value check		0
Unchecked		6
Invalid index	Invalid index update	0
	Invalid index bound	1
	Invalid index initialisation	0
Unexpected calculation		2
Logical errors		12
Missing initialisation		2

4.4.1 Memory safety taxonomy - Types of errors

The result from categorisation of the types of errors in our data set is listed in Table 4.4. Seven of the 46 vulnerabilities in the categorisation are included in more than one category. These are situations where one type of error causes another. Examples are CVE-2013-0211 and CVE-2016-6250 where the variable overflow allocation error are caused by 32-64 bit data type error, or CVE-2016-4302 where an unexpected zero causes a subsequent input below expected minimum.

In the Variable overflow category, we find four Variable overflow allocations and no Variable overflow check or Variable overflow in memory access. The variable overflow category covers instances of buffer overflows correlated to integer over- or under-flowed variables [1]. We find vulnerabilities in all subcategories from [1] in our analysis. These are instances where the overflowed variable is used in memory allocations causing the allocated memory to be smaller than the input copied into it, instances where the over-flowed variable is used in input checks causing the input check to fail and subsequently buffer overflows, or vulnerabilities where the over-flowed value is used in memory access [1]. In addition, there are one special in CVE-2015-8931. To calculate the values of TIME_T_MIN and TIME_T_MAX and integer overflow or underflow is deliberately triggered. Signed integer overflow is undefined behaviour in C, and such code is therefore vulner-

able. We add the subcategory *Intentional signed overflow* to the Variable overflow category.

In [1], the Variable overflow category is tied to the CWE-190 category (Integer overflow or wraparound). Of the five vulnerabilities from our data set in this CWE category, four is categorised in the Variable overflow category including the intentional signed overflow. The last is categorised under Mismatching data types and is further described below. In addition, the one vulnerability in CWE-189 is categorised as a Variable overflow allocation. This is CVE-2013-0211 described above.

The Unexpected input category covers input not expected by the developer. This could be negative file sizes or content lengths above or below expected maximum or minimum values [1]. In our data set we find vulnerabilities in all four subcategories. In addition, we add the new subcategory *Unexpected input NULL*. In this category we find Out-of-Bound reads caused by NULL or empty strings, and two of the vulnerabilities in the NULL pointer dereference category is categorised in this subcategory. There is also one special case where the first byte of a string is NULL. This is also counted as an unexpected NULL input.

We also categorise one NULL pointer dereference vulnerability in the unexpected input zero category and one in the unexpected maximum category. The former is caused by an unexpected zero length string, the latter is caused by an unexpected number of attributes causing a subsequent NULL pointer dereference. In addition, we find the Double-Free vulnerability in the unexpected input zero subcategory. Reallocation with size zero is causing the double free.

The mismatching data types category are vulnerabilities where values of different data types are assigned to each other [1]. In our analysis we find vulnerabilities in the Unsigned-Signed and 32-64 bits subcategories. The 32-64 bits instances are all implicit or explicit casts between integer types. Among these are CVE-2013-0211 and CVE-2016-6250 described above. Both are exploitable on 64-bit platforms due to cast between variable types and variable size differences on different platforms.

In [1], the Unchecked category are defined as vulnerabilities where user input reach methods vulnerable for buffer overflows [1]. In our analysis we find vulnerabilities related to memory transfer or memory comparison with **memcpy**, **memmove** or **memcmp**. In addition, there is one unchecked value causing a third-party library to read into invalid memory.

The Missing return check category covers vulnerabilities where developers does not check the returned values [1]. In [1], this was typically related to memory allocation. We find no instances of this error in our data set.

In the Invalid index category, we find vulnerabilities where invalid indexes are used in loops [1]. In our data set we find one instance in the invalid index bound category. This is linked to a logical error caused by a misplaced index check.

The Unexpected calculation category includes vulnerabilities where unexpected results are obtained during calculations [1]. In our analysis we find two vulnerabilities in this category. One is caused by parsing an MTREE archive entry

backwards to identify the file name. If the file name is the whole entry and Out-of-Bound read occurred and caused a subsequent unchecked error with **memcpy**. The other is an over-count of the MTREE line size when reading ahead in the archive file.

We also add one new main category, *Missing initialisation*, where the lack of variable initialisation causes memory vulnerabilities. Two vulnerability is added to this category. One is CVE-2015-8926, where a NULL-pointer dereference vulnerability was caused by a pointer not being initialised as NULL. This could cause the RAR archive reader to be tricked into reading the decompression buffer from the last decompressed file. The other is a Use-After-Free vulnerability in CVE-2019-18408, caused by a missing re-initialisation after an ARCHIVE_FAIL is returned. We also putt this vulnerability in the Logical error category due to the missing clean-up after a failure.

The last category covers logical errors made by the developer [1]. We find 12 vulnerabilities in this category and include issues like misplaced cleanup routines causing memory leaks and interchangeable use of compressed uncompressed sizes on uncompressed archive files without checks on the sizes being equal. One NULL pointer dereference vulnerability is placed in this category. This is vulnerability where a NULL value is deliberate is allowed when it should have been rejected. We also find the two Use-After-Free vulnerabilities in this category. One is the missing re-initialisation described above, the other was tied to logical errors in identifying a multi-volume RAR archive file.

Comparing our results of types of errors with the taxonomy from [1] we find vulnerabilities in all main categories except the missing return check category. As an extension to the original buffer overflow taxonomy, we introduce the subcategory *Intentional signed overflow* in the variable overflow category and the *Unexpected input NULL* in the unexpected input category. We also added the new main category *Missing initialisation* containing errors caused by missing pointer initialisation. Of the six vulnerabilities in CWE category 476 NULL pointer dereference, we find four in the Unexpected input category. Two in the new unexpected NULL subcategory, and the other two in the unexpected zero and unexpected maximum subcategories. Of the remaining two one is in the missing initialisation category and the last is a logical error. The Use-After-Free vulnerability (CWE 415) is caused by an unexpected zero input, and the two Double-Free vulnerabilities (CWE 416) are both caused by logical errors where one of the logical errors causing a missing initialisation.

4.4.2 Memory safety taxonomy - Types of sinks

The results from categorisation of the types of sinks in the data set is listed in Table 4.5. Sinks are the last instance where user input can exploit vulnerable code [1]. The two Use-After-Free vulnerabilities (CWE 416) are categorised in both Pointer read and Pointer write since both read and write behaviour was observed in the security report. This is CVE-2018-1000878 and CVE-2019-18408.

Table 4.5: Types of memory safety sinks. Extensions to [1] listed in *italic*

Main category	Subcategory	No.
Critical functions	Transfer memory	7
	String copy	0
	String scanf	0
	Memory compare	1
	Allocations	3
	Third-party functions	2
Array	Array read	7
	Array write	2
	Array read static	0
Pointer	Pointer read	20
	Pointer write	4
Integers		2

The critical function category contains sinks that are memory-critical functions [1]. In this category we find seven instances of transfer memory sinks. These are unchecked operations with **memcpy**, **memmove** and **memset**. We find no instances of string copy string scanf sinks, but we add three new subcategories. These are *Memory compare*, *Allocations* and *Third-party functions*. The one vulnerability in the memory compare subcategory is caused by an unchecked comparison with **memcmp** causing an Out-of-Bound read. In the third party subcategory we find sinks memory transfer functions in third-party components used by Libarchive. We choose to add this as a separate category to show how third-party components can be sinks. The allocation subcategory consists of three vulnerabilities connected to allocations. One is a memory leak caused by a misplaced free of the allocated memory and one is the Double-Free vulnerability caused by **realloc** with size zero. The last is a **malloc** failing due to out-of-memory. This error was correctly handled in code and discovered using and address sanitizer. Since this was correctly handled, we put this vulnerability in this subcategory and not in the transfer memory subcategory.

The pointer and array category are sinks caused by misuse of arrays and pointers [1]. In the array category we find bot array read and array write sinks, but no static read in arrays in our data set. In the pointer category we also see both reads and writes, and pointer read is the largest subcategory with 20 instances. As mentioned above the two Use-After-Free vulnerabilities are counted in both pointer read and write.

Last, we add a new category *Integers*. In this category we put two vulnerabilities caused by undefined behaviour in C. One is the deliberate signed integer overflow in CVE-2015-8931. The other is left shift of 31 bytes on a 32-bit integer, CVE-2015-8932. This is also undefined behaviour in C, and potential vulnerable code.

Comparing our categories of sinks against the taxonomy in [1] we find vul-

Table 4.6: Types of memory safety fixes. Extensions to [1] listed in *italic*

Main category	Subcategory	No.
Proper input check	Not too high	6
	Black listing	0
	Not negative *or zero*	9
	Not too small	4
	White listing	0
	Not NULL	5
	Is NULL	1
	Matching values	2
Check overflow underflow	Checkable type	0
	Check before calculation	2
	Check before send	1
Check if allocation succeeded		0
Use safe function		0
Fix indexes	Fix index update	2
	Fix index initialisation	0
Change to matching data types		4
Fix calculation		3
Proper allocation		4
Fix logic		8

nerabilities in all the main categories, and as an extension to this taxonomy we add the *Memory compare*, *Allocations* and *Third-party functions* under the Critical functions category. We also add the new *Integers* main sink category. All the NULL pointer dereference vulnerabilities from CWE-476 are placed in the Array read category, and as mentioned above the two Use-After-Free vulnerabilities are placed in both Array read and Array write. The one Double-Free vulnerability is in the new Allocations category.

4.4.3 Memory safety taxonomy - Types of fixes

The types of fixes are listed in Table 4.6. Six of these vulnerabilities has two implemented fixes and are therefore counted in more than one category. One example is CVE-2016-6250 where both changes to matching data types and proper input validations are part of the fix.

The proper input check category covers input checks that was missing, or the developer did not have in mind [1]. In our analysis we extend the Not negative category to also cover *Not zero*. Two vulnerabilities fall into the not-zero part of this category. One is the Double-Free vulnerability caused by **realloc** with new size zero. In addition to this we add three new subcategories. These are *Not NULL*, *Is NULL* and *Matching values*. In the not NULL subcategory, we find the five vulnerabilities from the unexpected input NULL subcategory from the types of errors.

These are all string operations on NULL strings. In the is NULL category we find a Out-of-Bound read caused by existing attributes on a 7Zip archive file when no attributes (NULL) were expected. The fix adds checks to reject the entry if the attributes are not NULL. In the matching values subcategory, we find two vulnerabilities. One is the earlier described vulnerability where the compressed and uncompressed sizes of a uncompressed ZIP archive file did not match. One of the implemented fixes was to compare these values before proceeding. In the other vulnerability an actual field length was compared with the given length as an input check.

In the check overflow underflow category, we have no instances in the checkable type subcategory. In [1] this was related to a *checkdint* class which allowed to check for under- or overflows [1]. No such class or other methods exists in Libarchive. We find two instances in the check before calculation subcategory, where there are checks added to to check if integer overflows occur before the integer value is used. In addition, we added the new subcategory *Check before send*, containing one vulnerability (CVE-2013-0211). In this case a variable is overflowed in a cast when sent as parameter to other methods. The fix checks that the value is not larger than MAX_INT before sending the variable. Since this check affected multiple possible calculations, we keep this as separate subcategory and not as a part of check before calculation. All three vulnerabilities fixes errors from the variable overflow error category.

The fix index category contains fixes to two vulnerabilities in the fix index update subcategory. Both were off-by-one errors, the first caused by a misplaced check the other by a logical error in handling of UTF-16 strings. In the change to matching data types category we find four instances, all from the mismatching data types error category except CVE-2013-0211 which is described above. This category includes fixes to implicit and explicit casting of variables. The fix calculation contains the two vulnerabilities from the unexpected calculation error category, in addition to one from the logical error category. In this vulnerability an Out-of-Bound read occurred due to use of wrong size variables in the and the fix corrects this error in the calculation.

In [1] the proper allocation category contains fixes where the memory allocation was fixed, and the four vulnerabilities in our categorisation also falls into this definition. All caused memory failures due to insufficient memory allocations either through static allocations or errors in returned values used in allocation.

In the two categories "Check if allocation succeeded" and "Use safe functions", we have no instances in our analysis. The allocation succeed category is linked to the missing return check error category were we also have no instances in our analysis, and none of the vulnerabilities in our data set were fixed with use of safe functions.

Last, we add a new category *Fix logic*. In this category we find fixes to vulnerabilities from the logical error category. Examples are fixes to clean-up routines to prevent memory leaks, proper initialisation or re-initialisation of pointers to prevent NULL pointer dereferences or Use-After free errors, or fixes of consume

logic when parsing an archive file to prevent a quasi-infinite loop. We also put the fix to the intentional integer overflow used to find time max/min values in this category. The fix simplified the logic to use INT64_MAX/INT64_MIN as time max/min values.

Comparing our categories of fixes with the taxonomy from [1] we find vulnerabilities in all categories except the "Check if allocation succeeded" and "Use safe function" category. We add the new subcategories *Not NULL*, *Is NULL* and *Matching values* under the proper input check category. We also add the new *Fix logic* category, containing fixes to the memory vulnerabilities not covered by any of the other categories. We find the NULL pointer dereference, Use-After-Free and Double-Free vulnerabilities in the proper input check and the fix logic categories.

4.4.4 Non-buffer overflow vulnerabilities

Above we have categorised 46 of the 53 Libarchive vulnerabilities using the memory safety taxonomy. The remaining seven vulnerabilities are:

- CVE-2011-1777
- CVE-2011-1778
- CVE-2015-2304
- CVE-2015-8930
- CVE-2016-5418
- CVE-2016-7166
- CVE-2019-1000020

The two vulnerabilities CVE-2011-1777 and CVE-2011-1778 are reported as buffer overflow vulnerabilities in ISO9660 and TAR archive file functionality respectively, allowing an attacker to cause denial of service through crafted archive files. Both CVE entries have CWE category 119 "Improper restriction within the bounds of memory buffer". In our data collection and analysis, we find no direct or specific trace of any buffer overflows, but through the applied patches to the errors we see an error through the use of **exit** in error handling. This will cause program termination on errors, and the code shows that this could be triggered through a failed memory allocation. The fix moves away from the use of **exit** in errors (only used in some special cases), and to the use of error/status codes which makes it possible to either terminate in orderly manners or skip the current processing and move on to the next archive entry.

CVE-2015-8930, CVE-2016-7166 and CVE-2019-1000020 are all variations of infinite loops. The first is caused by a special situation in ISO9660 archive files where a directory in an archive entry is self-owned, which leads to an infinite loop in a path-builder method. The second is caused by a stack-compressed archive file (archive compressed multiple times). Without any upper limit of number of compressions an attacker could trigger an semi-infinite chains of compressors causing resource exhaustion and denial of service. The fix applies a maximum of 25 compressions rejecting an archive if more. The last also relates to ISO9660 archive files, and occur due to a logical error where a failed sanity check still returns status

ARCHIVE_OK causing an infinite retry without moving the pointer. The applied fix changes to logic.

The last two, CVE-2015-2304 and CVE-2016-5418, are variations of directory traversals or sandbox evasions. The first relates to absolute paths in the *bsdcpio* implementation, the other relates to hardlinks causing sandbox evasions. These are fixed with rejecting archive entries or stripping archive entry paths of absolute paths.

4.5 Observed vulnerability phenomena

In the above sections we have seen that the Libarchive vulnerabilities in our data set centers around memory related issues, and we have analysed patterns behind the types of errors, sinks and fixes of these vulnerabilities. In the following section we will present the results from the analysis of general patterns behind how vulnerabilities emerge in the OSS package.

4.5.1 "The dark side of the code"

In modern application development the complexity of the application and the manner of their development will cause aspects in their behaviour that is not always considered or fully understood by the developers. This can range from the application level to low-level system calls [27]. The layered nature of applications encapsulates and hides lower-level details resulting in the developer not always understand the operational details of the entire application. As a result, the developer does not know, or have access to the low-level details where the security issues occur [7]. This is described by Pieczul and Foley [27] as "The dark side of the code", a phenomenon which forms a security gap between expected and actual behaviour of the code [27]. In our data set of Libarchive vulnerabilities we see this phenomenon in practise related to integer overflows and misuse of C standard library functions.

As we have seen in Section 4.4 we have different integer overflow related vulnerabilities. These are caused by low-level details as platform issues, undefined compiler behaviour, or integer casting issues not considered by the developer. From the definition of "Dark side" we see that these issues are examples of this phenomena. One of these examples is the intentional signed overflow described earlier (CVE-2015-8931), used to get min/max time values. The problem in this solution is signed integer overflows being undefined C behaviour. Dietz *et al.* [24] describe this type of undefined behaviour as *time bombs*, code that can work today but can break in the future due to optimisations and other changes [24]. Two of the other integer overflow vulnerabilities relates to size differences between variables on 32- and 64-bit platforms. These are CVE-2013-0211 and CVE-2016-6250, relating to ZIP and ISO9660 archive files respectively. The former is caused by signedness issue in cast between **size_t** and **int64_t**. On 32-bit systems this cast is unproblematic with INT64_MAX > SIZE_MAX, but on 64-bit platforms the

width is the same and the sign will change. The overflowed value is used in size compare before writing to an allocated buffer causing a buffer overflow [23]. The latter is another signedess issue related to the check of an archive entry file name length in ISO9660 archives. In this check the file name length is stored as size_t but casted to int. This is also a potential problem on 32-bit systems given the equal variable width of size_t and int, but to allocating enough memory to bypass the file name length check is only possible on 64-bit systems [23]. In addition to these examples, we find integer overflows caused by implicit casts between int64_t and int, and integer overflows caused by variable overflows in calculations.

Of the C standard library misuse vulnerabilities, we find one security error (CVE-2015-8918) caused by the use **memcpy** on an overlapping memory buffer causing a segmentation fault. The copy between overlapping memory buffers is undefined behaviour [43]. In addition, we find two issues caused by undefined or implementation defined compiler behaviour. One issue is the Double-Free vulnerability caused by **realloc** with size zero. A call to **realloc** with new size zero is implementation defined behaviour and the allocated memory might be freed [43], resulting in a double free when the clean-up routine using **free** is called. The other issue is an illegal left shift on an integer (CVE-2015-8932). The root cause a malformed archive file with an invalid compression parameter, but this error resulted in a left shift of 31 bytes on a 32-bit integer. As for the intentional signed overflow a left shift of at least the full size of the variable is undefined behaviour and also a potential "*time bomb*" [24].

Pieczul and Foley [27] demonstrated the "Dark side of the code" phenomena through a theoretical example showing how the Java method **WebUtils.snapshot()** hides functionality making it possible to exploit the method to access resources on the local network [27]. In the study of the evolution of security controls in the Apache Struts OSS package, Pieczul and Foley [7] observed the "Dark side" phenomena in practice. The study showed how low-level details of components used in the Java application was inaccessible to the developers causing the "Dark side" phenomena [7]. In our project we have observed how low-level system details like undefined or implementation defined C behaviour or integer overflows caused by 32/64-bit platform issues can be "Dark side of the code", areas of the code and functionality not fully considered or understood by the developer. These are issues requiring in-depth knowledge of programming language application platform(s) specifics and have the potential to cause a security gap between actual and expected behaviour when they are not understood or considered by the developer.

4.5.2 Blind spots

In our data set of Libarchive vulnerabilities we see that many of the security errors relates to special crafted or malformed archive files. These archive files give unexpected input or causes unexpected behaviour in the code, which we also see through the types of errors in the memory safety taxonomy (Table 4.4)

where we find "Unexpected input" as the category with most vulnerabilities. The study by Oliveira *et al.* [28] showed that vulnerabilities are blind spots in developer's heuristic-based decision-making process. In their day-to-day operations developers focus on the problem at hand which normally involves functional and performance requirements, and they usually assume common cases for the inputs and states the piece of code can reach. The vulnerabilities, on the other hand, lies in the uncommon cases often overlooked by the developer and exploited by the attacker. To find these cases a significant cognitive effort is required through complexity of fault analysis, whereas people normally prefer to use as little effort as possible when solving a problem [28].

The blind spot instances we find in our data set relates to illogical or illegal values in the archive files, to example wrong size or compression ratio values or empty archive entry file names. One instance is CVE-2015-8930 where directory in one ISO9660 archive entry is a member of itself. This uncommon case results in an infinite loop in a path-builder method used when parsing the archive entries. The self-owned directory results in the pointer not moving and the path-builder adding the same directory to the path indefinitely. The fix adds a sanity check rejecting a self-owned directory before processing of the entry, and also ads a depth check in the path builder method rejecting an archive entry if directory depth of 1000 is reached. One observation from these fixes is that a directory depth of 1000 violates the ISO9660 specifications, which restricts the directory hierarchy depth to eight [44].

Another instance is CVE-2016-4302, related to RAR archive files. An illegal dictionary size of 0 was not rejected in the archive processing. This resulted in a zero-sized memory allocation, and a subsequent heap-based buffer overflow and possible arbitrary code execution. Adding to the security issue was also an assumption in the Ppmd7 decompression routine of a dictionary size of at least 12 bytes. This assumption was not enforced, allowing the overflow of the previous heap buffer chunk. The applied fix rejects zero sized dictionaries and enforce the 12 byte assumptions in the Ppmd7 decoder.

We also see a blind spot scenario in CVE-2016-1541, related to ZIP metadata entries in macOS. On an uncompressed file, the compressed and uncompressed sizes are used interchangeably when allocating and writing to a memory buffer. The uncompressed size was used when allocating the buffer and then the compressed size was used in the input check when writing to the buffer. In a normal situation with an uncompressed archive file these sizes would be equal, but the fields are user controlled making it possible for an attacker to manipulate the values to create a buffer overflow.

As a link between the "Dark Side" phenomena and blind spots we find the integer overflow in CVE-2016-4300. In a crafted 7Zip archive file it is possible to cause a **size_t** variable to overflow given the sufficient number of **numFolders** and the sufficient **numUnpackStream** values. The overflowed value is used in memory allocations and could be exploited to cause a heap buffer overflow. In the code we find an input check on each **numUnpackStreams** being less than the

defined variable **UMAX_ENTRY**, but each **numUnpackStreams** is added to the **size_t** variable **unpack_streams** without any check making an integer overflow possible. To exploit this vulnerability an abnormal large number of sub-streams are needed. That we find input checks on each **numUnpackStreams**, but not on in the calculation of **unpack_streams** can be explained by this scenario being a blind spot for the developer.

Another link to "Dark Side" is the illegal left shift described in Section 4.5.1. The root cause for this vulnerability is a compression code with an illegal size larger than 16 bits. Some validations existed of the compression code but not on the size, showing that this was an unconsidered scenario.

Of the other vulnerabilities caused by developer blind spots we find archive files with zero or negative file size values, negative compression size values, or archive entry file names that are either NULL, empty or containing illegal characters values. Though we find input value checks, sanity checks and return value checks in the code, these vulnerabilities are exploited through uncommon cases or conditions not considered by the developers. We also see that this happens repeatedly, to example with the introduction of RAR version 5 support in Libarchive version 3.4.0. A missing check on illegal RAR headers of size 0 resulted in segmentation faults when parsing the corrupt RAR archive. Legal RAR 5 header size is 7 bytes and illegal values should have been rejected. Overall, the exploitation of vulnerabilities through crafted or malformed archive files is blind spot to Libarchive developers.

4.5.3 Opportunistic fixes and solutions

Another phenomenon we see in our data set is opportunistic fixes and solutions. When fixing a security issue, the developer might prefer the solution that fits the existing code and not the more extensive and complete solution relating to the root cause of the problem. Such solutions are often more convenient to implement and does not interfere with the existing code structure [7]. This can also relate to implementation of new functionality. The easiest solutions that fit and not interfere with the existing code is preferred by the developer.

We find one clear example of this phenomena in CVE-2015-8916. The security error is reported as a NULL pointer dereference in the *bsdtar* implementation caused by an empty archive entry file name returned by the RAR reader. Further processing of the returned file name results in the NULL pointer dereference error. The root cause of the vulnerability is identified by the developer as the header in the malformed RAR archive file being wrongly interpreted by the RAR reader as being a multi-volume RAR file, causing the empty file name to be returned. Though the root cause is identified, the implemented fix only adds a return value check in *bsdtar* rejecting empty file names returned from the reader and preventing the NULL pointer dereference error. The fix of the root cause is deferred to later versions, but we find no follow up of this until the same multi-volume RAR problem occur in CVE-2018-1000878. In this error a malformed RAR archive file is

also interpreted as a multi-volume archive file as in CVE-2015-8916, but this time the error causes an early release of a Ppmd7 buffer resulting in a Use-After-Free vulnerability. The implemented fix applies extended checks to verify if the file is a multi-volume file or not, preventing the premature release of the buffer. Such an extensive check to fix CVE-2015-8916 could have prevented the vulnerability in CVE-2018-1000878.

In relation to CVE-2015-8916 we also find CVE-2015-8917. This is the same NULL pointer dereference in *bsdtar* as in CVE-2015-8916, but this time caused by a malformed CAB archive file returning empty archive entry file names. The same return value check in *bsdtar* is applied as fix to both vulnerabilities, but as for the RAR vulnerability the CAB vulnerability has a root cause in the malformed archive file. The root cause is identified as a combination of missing CAB header checks and illegal characters in the entry file name. No follow up of these issues are found, and no later CAB related vulnerabilities are caused by these issues.

Of opportunistic or easy solutions, we see the error handling issue related to CVE-2011-1777 and CVE-2011-1778. The use of **exit** in error handling made it possible for an attacker to trigger an application termination through a malformed archive file. The two reported vulnerabilities relate to memory allocation errors in TAR and ISO9660 archives, but the error handling was applied across the whole library. By Libarchive version 3.0.0a the error handling system was mainly rewritten, and the library moved away from process termination to status codes and either graceful termination or skipping or bailing on the current processing while continue with the next item in line. The process termination with **exit** is only used in special cases, and the new error handling routine is advised in the "getting started" guidelines for Libarchive.

The original **exit**-solution is an opportunistic easy solution to error handling where there is no need to clean-up or other measures to make the application continue to functioning after an error, but Libarchive being a programming library such solution has implication beyond the library or the *bsdtar* and *bsdcpio* implementations. Given that this issue was fixed and the current advice around error handling in the guidelines, the issue was understood by the developers. But we also see in discussion forums that this did not have full priority and that it took some time to implement.

Of other instances related to opportunistic fixes and solutions we again see the illegal left shift vulnerability described above, caused by an illegal sized compression code. In the original code we found some validation of the compression parameters, but also the comment `"TODO: verify more"`. These verifications were added in the fix to the vulnerability, but the validation of the compression code size was a separate verification issue. Though the root cause would not have been caught by the complete compression parameter validation, the initial solution was incomplete but still deemed good enough to be put into production.

We also see the directory traversal vulnerability in *bsdcpio* in CVE-2015-2304 in relation to opportunistic fixes. In the discussion of a potential fix to this vulnerability it was decided to implement a non-Windows solution as the first step.

We find no trace of a follow-up of the Windows part of this, but in the Windows related code we find a *cleanup_path_name()* method where absolute paths seems to be handle correctly. There is therefore unclear how much of a problem absolute paths were on the Windows-side in the beginning, but the discussion leading up to the fix shows the willingness to limit the solution in the first place.

4.5.4 Report biases

In their study, Pieczul and Foley [7] found how a "Dark Side" phenomena also could exists in the documentation and interpreting of vulnerability reports. Vulnerabilities are often identified by security researchers external to the development project, and though these reports give detailed descriptions of the security problem and attack vectors they can lack a broader and detailed understanding of the application they are testing. The result can be that the developers reviewing the security report limit their scope to the problem as described in the report instead of looking at the problem in a broader scope and trying to identify similar problems or other attack vectors [7]. In our analysis we observe how the report bias is influenced by the use of automated testing tools like fuzzing and in addition to create a dark side scenario also can influence opportunistic fixes.

Fuzzing is Black-Box testing technique used to identify implementation bugs using malformed data injection in an automated fashion. Fuzzing can also be used for automated file format testing, where the fuzzer generates malformed file samples and uses these in the automated test [45]. In our data set of Libarchive vulnerabilities we see that fuzzing is used to identify several of the vulnerabilities. With Libarchive being a low-level file centered application, fuzzing is a good and necessary testing tool and many of the vulnerabilities would not have been identified without the use of fuzzers. But we also observe some phenomena related to these vulnerabilities that is worth noting when studying how vulnerabilities emerge and evolve.

As described above, Pieczul and Foley [7] found that the vulnerability reports could limit the scope of the developers to the error as described in the report. What we see in our analysis is that the vulnerabilities identified using fuzzing comes with the log or trace from the testing as the only documentation of the vulnerability. In some instances that makes dismissing or closing vulnerabilities as duplicates the easy solution even if the root cause is different. If we go back to CVE-2015-8916 and CVE-2015-8917 described in Section 4.5.3 we see that an opportunistic fix was implemented as a solution to both vulnerabilities. Both vulnerabilities reports the same NULL pointer dereference in the *strip_absolute_path()* method in *tar/util.c* using a fuzzer with malformed RAR and CAB archive files respectively. As described above, the root cause in the RAR case was identified as a single-/muliti-volume RAR archive issue, but a fix to this was deferred to later and the implemented fix focused on the NULL pointer dereference by adding a return value check in *bsdtar*. The RAR root cause was identified by the same developer implementing the fix. The same developer also closed the CAB vulnerability as a

duplicate with same fix and comments in the issue tracker, but without identifying any root causes to that vulnerability. The root causes in the CAB archive reader were identified some months later by another developer. Given the identified RAR root cause, the existence of a different CAB root cause must have been understood by the developer but the similarities between the vulnerabilities have caused the same opportunistic fix to be applied to both vulnerabilities.

We also find fuzzing related vulnerabilities where there are actual duplicated reports in our data set. One example of this is CVE-2016-8688 related to MTREE archive files. We find six different error reports in the Libarchive bug tracking system related to this vulnerability. All are reporting Out-of-Bound or Use-After-Free errors in the MTREE reader. Five of these issues were reported by the same researcher using a fuzzing tool, while the last was reported by another researcher with manual testing using a crafted archive file. All had the same root cause in a calculation error when reading ahead to the next line in the archive file. In addition to this example there are other similar issues, either with one CVE-ID connected to two or more error reports with the same root cause or two or more CVE-ID's where the error reports on each has the same root cause. In some of these cases the different error reports are reported by different researchers around the same time and the double reporting have a natural explanation. For other double report, for the example with CVE-2016-8688, the multiple reports from the same researcher can be explained by the automated testing tools and the lack of broader knowledge of the application by the researcher. The different tests are performed with different settings and malformed files and produces different traces and logs. Without further analysis of the result and knowledge of the code an external researcher cannot easily say if these issues are the same or not. Reporting all the issues are therefore correct handling of this, but it falls on the developers to analyse these further. With several double-reports like the ones in CVE-2016-8688 this can cause situations where it easy to dismiss issues as duplicates based on trace files and previous experience even if the root causes are different.

Of other issues we see related vulnerabilities reported using fuzzing is problems to reproduce the security errors and situations where the error is correctly handled in the code but still reported as a vulnerability. We find that the problems with reproducing the errors are due to lack of information around issues like platform and compiler settings, and that there are discussions around these issues in the in the issue tracking system. CVE-2016-4809 is caused by a **malloc** failure in the CPIO reader due to large symlinks. The failure is handled correctly in the code, but still reported as an error or after a fuzzer test. An input validation on symlink sizes is added as a fix. Together with the double reporting, the reproducing problem and the issues like CVE-2016-4809 risk creating a sense of noise and the issues being less important and we get comments like "*Do something sensible for empty strings to make fuzzers happy*", as we see in the commit comment in the fix to CVE-2017-14166.

4.6 Socio-Technical system analysis

From the phenomena described in the previous section we see how the low-level details of operating systems, platforms and compilers create "Dark sides" for the developers and how complexity of file systems and file formats supports can create developer blind spots. We also observe how developers opting for opportunistic fixes and solutions creates or influence vulnerable code, like the error handling solution causing abrupt process termination or the fix to the NULL pointer dereference in *bsdtar* where the underlying multi-volum RAR issue was left unfixed causing a new vulnerability later. Last, we have also seen how external security researchers and the use of automated testing tools creates report biases. We have seen examples of vulnerabilities with similarities in the trace or log files are treated as the same issues without attempts to find and fix any underlying root causes. We have also observed actual double reporting due to the use fuzzers, and also problems reproducing the vulnerabilities due lack of information in the vulnerability reports from the testing tools.

To gain insight into how these phenomena occur and what is influencing them we analyse the Socio-Technical System (STS) surrounding the vulnerability handling in Libarchive. In this analysis we adopt the STS model by Kowalski [40]. As discussed in Section 3.4, an OSS project consists of both social and technical aspects and the STS model help us better understand how the above described phenomena. The STS model is showed in Figure 3.2 and a summary of our findings is presented in Figure 4.1. The following sections presents these findings in detail.

Culture • Security culture	**Methods** • Secure coding guidelines/standards • Internal testing tools and standards • External testing tools and standards • Multi file format support
Structure • Contributing developers • Security researchers/testers • Software users • Vulnerability disclosure policy	**Machines** • Cross-platform environments

Figure 4.1: Socio-Technical analysis results

4.6.1 Culture

Sabbagh and Kowalski [46] define security culture as *"The way our minds are programmed that will create different patterns of thinking, feeling and actions for providing the security process"*. As we have described above, the "Dark side" phenomena is the security gap that occur between expected and actual behaviour of the code and the blind spots happens in the corner cases and unusual informa-

tion flows overlooked by the developers in their day-to-day operations. From the definition of the security culture, we see that this guides the thinking and actions around the security process in the OSS project, and from that we see that the security culture influence how and to which extent the "Dark side" and the blind spot phenomena are allowed to occur and influence the security in the project. The security culture will also guide the thinking around opportunistic fixes and solutions and the handling of reported vulnerabilities and how report biases will influence this process.

In our analysis of the Libarchive vulnerabilities we do not find a clear and defined security culture. There are no existing guidelines regarding secure programming or vulnerability awareness. In the Libarchive GitHub project page there is a Wiki page with a "getting started" guide including code examples, but none of these includes the topic of secure programming or common vulnerabilities. The one thing we find is an error-handling guideline which describes the correct use of error-codes with regards to the earlier described vulnerabilities caused by the unconditional use of **exit** on errors.

From the follow-up and implemented fixes of the reported vulnerabilities we see few if any discussions regarding broader scopes of the vulnerabilities, and we see that the focus mainly is on fixing the problem at hand without any attempts to find similar vulnerable code or look at the vulnerability in a broader context. We also see, as discussed in Section 4.5, that opportunistic fixes and solutions are selected when implementing fixes or adding new functionality. In connection with this there is no general broader focus on the common types of vulnerabilities found in the project. As discussed in Section 4.4 we find that Libarchive vulnerabilities center around memory safety, and we find no guidelines or discussions around these types of vulnerabilities with regards to how such vulnerabilities occur and how to prevent them.

From this we observe how a limited security culture with little focus on what types of vulnerabilities that can occur in the application and how to secure the code against these, allows for our observed phenomena of "Dark side", blind spots, opportunistic fixes and solutions, and report bias to occur.

4.6.2 Structure

OSS development is characterised by community driven development, and unlike traditional software development organisations they do have few if any formal structures regarding formal planning and schedules [47]. With the focus on the question of how vulnerabilities emerge and evolve in an OSS project we find three stakeholders of interest. These are the contributing developers, testers or security researchers testing the software, and the users of the software. In addition, the vulnerability disclosure structure is of interest in our analysis.

With the focus on vulnerability introducing commits and vulnerability fixing commits we find that only a few of the developers with contributing commits to Libarchive were involved. Libarchive has a total of 184 developers with contribut-

ing commits. Of those, the top 5 contributors have more than 100 commits, and the top 2 has more than 1500 commits. Analysing the vulnerability introducing commits, we find 18 commits after Libarchive version 2.8.5. Among these we find extensive commits introducing new archive format support or extending support for existing formats. Eight developers were involved in these commits, where 13 of these commits, and 2 pull request approvals were by three of the top 5 developers. Of the vulnerability fixing commits we find 41 of the 53 vulnerabilities were fixed by one of the top 5 developers, and in addition we find that two of the top 5 developers approved six pull requests with fixes of nine vulnerabilities. From this we see that the vulnerabilities are not caused by a broad community of developers with limited knowledge about the project. The fixing of vulnerabilities, including the use of opportunistic fixes, are also limited to a few developers. All with broad knowledge of the project.

Among the testers or security researchers we find three groups contributing to the testing of Libarchive. These are professional security researchers like Cisco Talos or professional tools like Google OSS-Fuzz, semi-professional researchers like "The fuzzing project" by Hanno Böck, or single users testing the library on their own. As discussed in Section 4.5.4, external testers often have limited knowledge of the internal structure and working of the software under testing. This together with the use of automated testing tools like fuzzers contribute to a report bias where the developers risk limited their scope to the security error at hand without any attempt to look at the vulnerability in a broader scope.

Libarchive is used across operating systems like FreeBSD, NetBSD, macOS, Windows and various Linux distributions. It is also used in individual software like package managers, archiving tools and file browsers. This creates a cross-platform environment increasing the complexity in the software. The cross-platform environment also influences the requirements of the archive formats with regards to platform and OS specific behaviour.

Last, there are no defined vulnerability disclosure policy for Libarchive. Many of the vulnerabilities found using fuzzing are reported as "normal" issues in the issue tracking system in GitHub. For other vulnerabilities the researcher asks for contact information either through issue tracking or the discussion forum. In some instances, the project owners are also contacted directly. The vulnerabilities reported through the issue tracker are fixed as they occur, and as described above the fixes are mainly done by a few of the top contributing developers. We also find examples of vulnerabilities that are not followed up until they are re-reported through a new or similar issue. This can be due to a lack of a structured vulnerability disclosure policy, and that the possible security impact of vulnerabilities reported through the issue tracking system are not fully understood by the developers.

4.6.3 Methods

In the methods section we analyse the methods used with regards to secure programming, the methods used in testing the software both internal and external, and the methods used to implement archive formats support in Libarchive.

As discussed under Culture in 4.6.1, we find no guidelines or coding standards regarding secure programming in Libarchive. There are also no specific guidelines or introduction to the most common vulnerability categories in the OSS project. What we find is a Libarchive test suite and in the testing guidelines we see: "*Any significant change to Libarchive, including most bug fixes, should be accompanied by new tests or changes to existing tests*" [2]. This is a reactive approach to testing, where you test for known issues whereas the vulnerabilities often are found in the uncommon cases and in unknown issues as we have seen in the previous discussions. We also find fuzz tester in the testing suite, and we also see that after 2016 Libarchive was added to Google OSS-Fuzz which is a free fuzzing platform for OSS [48]. As described in Section 4.5.4 fuzzing is an automated testing tool used to find software bugs through malformed data injection [45]. This is proactive testing approach where the goal is to uncover unknown security bugs and ideally fix these before the vulnerable code reach production.

Fuzzers are also used by external security researchers testing Libarchive. As discussed in Section 4.5.4 we have seen how these testing tools and the security reports produced from these tools can contribute to a report bias. The security reports consisting of trace or log files from the fuzzers combined with external researchers with limited knowledge of the program under testing causes issues like double reporting of error, problems recreating the errors, and we also find examples of the developers closing errors as duplicates even if the errors have different root causes. The last issue can be contributed to the security reports consisting of trace file limiting the developers scope of the problem to the issues as showed in the trace file.

Another report issue related to the use of fuzzing as test method is a bias in reported read versus reported write related vulnerabilities. Libarchive can read 18 archive formats and write 8 archive formats. Of the 53 vulnerabilities in our data sat, only 4 relates to write functionality. One explanation for this is that testing of read functions is easier done with fuzzing tools, than the manual process needed to test write functionality. Some write-tests exists in the Libarchive test suite, but from the Libarchive test guidelines we see that these tests mainly verify bytes written to memory to test that an archive was created correctly [2]. We have not done any analysis or testing into undiscovered vulnerabilities in the write-part of Libarchive and cannot say that there is an actual bias in the testing. But given the difference in number of vulnerabilities and the extensive use of fuzzers in the testing, this is a possible bias worth noting.

Another method related issue is the implementation of archive format support. Multiple archive formats are supported by Libarchive and many of the vulnerabilities in our data set is found is edge and corner cases in crafted or malformed

archive files. Libarchive need to be able to read archive files created from other implementations, and in some cases this means that not every part of an archive format standard is followed to the letter. We see examples of these dynamic implementations of an archive format support leads to missing or insufficient input validations that could be exploited. One example of this is ISO9660 infinite loop described earlier caused by a self-owned directory. The ISO9660 standard limit the directory to a depth of eight [44], but no input check existed on this restriction and after the fix of the vulnerability Libarchive still allows up to 1000 deep directory hierarchies. Another example is the vulnerability in the RAR reader caused by an illegal zero sized dictionary variable. The RAR specification defines legal dictionary sizes to be between 64 KB and 4096 KB [49], and we also find these values defined as constants in the source code, but they are all unused. Given that not all archive files to be read by Libarchive follows the standard it is necessary to have a less strict implementation of the different archive formats, but as the two examples shows this can also cause missing input validations and possible vulnerable code.

4.6.4 Machines

As discussed above, Libarchive is cross-platform software used across different operating systems and in different individual software packages. This cross-platform context influences the vulnerabilities and the fixes to these. As we have seen in Section 4.5 this can be through 32/64-bit platform issues causing integer overflows or fixes that only applies to some of the platforms like the fix to the directory traversal in *bsdcpio*.

The cross-platform context also applies to the support of multiple archive formats, including support for platform specific implementations of these archive formats. This adds to the complexity of the application, and as discussed in the previous section the need to for multi-format support can cause missing input validations and possible vulnerabilities.

Chapter 5

Vulnerability evolution model and case studies

In this chapter we present our model of how vulnerability emerge and evolve in an OSS package. The model is developed from the results of our analysis into patterns and phenomena behind the Libarchive vulnerabilities in our data set, and from the analysis into the STS surrounding the vulnerability handling in Libarchive. We will also present the results from a case study where we apply the model on two other Open-Source Software (OSS) packages, and also do a comparison between our model and a similar vulnerability model by Pieczul and Foley [7].

5.1 Vulnerability evolution model

From our analysis of the Libarchive vulnerabilities, we develop the model depicted in Figure 5.1. The model describes four vulnerability causing phenomenon and three input or influencing elements into these phenomena. The four phenomena are "Dark side", blind spots, opportunistic fixes, and opportunistic solutions. The three inputs elements are the security culture and practises in the OSS project, the application context and report biases in the vulnerability reports.

In our analysis we have observed how "Dark side" phenomena can occur in low-level details application platforms and compiler details, and how blind spots occur in application complexity caused by cross-platform support and multi-archive format support. We have also observed how opportunistic fixes to one vulnerability can cause new vulnerabilities, and also how opportunistic solutions to initial problems causes vulnerable code. We have also observed how the vulnerability causing phenomena can influence or be connected to each other. One example of this is how integer overflows can be caused both by low-level details not considered by the developer and also through unexpected input values. We also see that opportunistic fixes and opportunistic solutions causes new "Dark side" and blind spot scenarios leading to new vulnerabilities, and we see that the "Dark side" and blind spot phenomena leads to opportunistic fixes of a vulnerability in-

fluenced by report biases in the vulnerability reports.

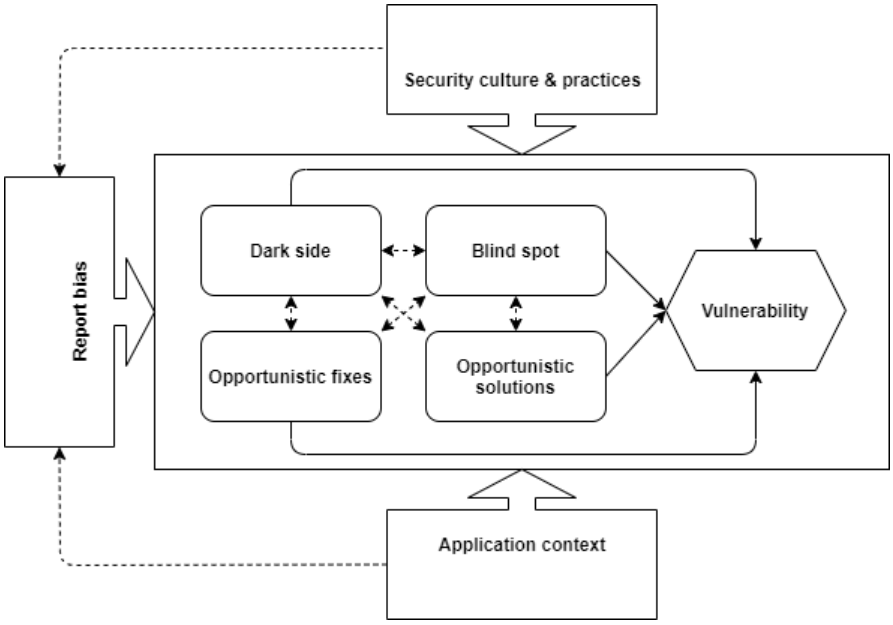

Figure 5.1: Vulnerability evolution model

Through our STS analysis we see how the OSS project security culture, and the structures and methods around this culture, influence the four vulnerability causing phenomena. As defined earlier, the security culture is the way our minds are programmed that will create different patterns of thinking, feeling and actions for providing the security process [46], and as we have seen in our STS analysis a mature security culture should include methods like secure coding guidelines and structures like structures like vulnerability disclosure policies. Together, a mature security culture with these methods and structures in place will decrease the room for the vulnerability causing phenomena to occur. To example, secure coding guidelines and focus on common vulnerabilities in the project will increase the focus on possible vulnerable code and make the "Dark side" and blind spots phenomena less likely. Such guidelines together with a defined testing policy can also make opportunistic fixes and solutions less likely choices when implementing fixes or new solutions. A security disclosure policy can ensure a structured and defined handling of reported vulnerabilities, and to example ensure broader scope when fixing a security bug and decrease the chance for opportunistic fixes of these types of bugs. On the other hand, a less mature security culture with fewer methods and structures in place to support the vulnerability handling in OSS project will increase the room for the vulnerability causing phenomena to occur. From this we see that security culture in the OSS project determines the methods and structures in place around the security handling, and together the security culture and practices are an influencing element into the vulnerability causing phenomena in our model.

In the application context we find elements from the structure, methods, and machine in our STS analysis, and it is defined by the cross-platform support, the user base and used testing tools. As showed in the STS analysis, the cross-platform support covers both support for multiple operating systems, 32/64-bit platforms, and multiple file systems. The analysis also showed a connection between the user base and the cross-platform support in terms of user request for support of multiple operating systems, file systems, file formats, etc. In the case of Libarchive we saw this also included the support of vendor specific implementations of file format support, not always compliant with the file format specifications. Together with the used testing tools this forms the application context. As we have seen, the "Dark side" phenomenon is the security gap that occur between expected and actual behaviour, and in our analysis we have observed how this can happen in the low-level operations of cross-platform environments and compiler operations determined by the application context. The blind spot phenomenon is caused by unexpected input and system states not considered by the developer, and again we have observed how this phenomena is influenced by the application context and a cross-platform environment where the system complexity is increased by multi-format input causing more areas for blind spots to occur.

Last, we have also observed how report biases influence the phenomena causing vulnerabilities. In our analysis we first observed report bias as a separate phenomena in the emergence and evolvement of vulnerabilities, and through the STS analysis we observed how report bias was caused by the structure, methods and machines through external security researchers, the use of automated testing tools and narrow security reports. We therefore put report bias as an influencing element in our model. The report bias can in itself cause a "Dark side" or blind spot. The limited scope of a security report can hide broader details and consequences of the vulnerability and cause the developer to limit the scope of the fix to the error as described at hand. We have also observed how the use of automated testing tools like fuzzers can contribute to this phenomenon through security reports mainly consisting of trace and log files, and we have also seen how such security reports contributes to opportunistic fixes where the root cause is left unsolved and only the problem described in the trace file is solved. We also observe report biases in testing coverage of the code caused by the used testing tools. We find more vulnerabilities in code that can be automated tested than in code requiring manual testing. Last, we also observed that automated testing caused double reporting, and this combined by the security bugs being reported as normal bugs in the issue tracking system caused the developer to look at these issues as any other issue where the goal was to close the issue.

As showed in our model, the report bias is also influenced by the security culture and practices and by the application context. The security culture and practices determine the vulnerability disclosure practice and the practices in use for fixing vulnerabilities. The application context determines the types of testing tools that can be applied to the application.

5.2 Case study

To test our model, we perform a case study by applying the model to two additional OSS packages. The case study is performed by first identifying and analysing the input elements of security culture and practices, application context, and report biases, before analysing the reported vulnerabilities in the OSS packages against our vulnerability causing phenomena. The goal with these case studies is to identify how well the model describe how vulnerabilities emerge and evolve in an OSS package, and possibly identify other elements not included in our model.

The two OSS packages used in the case studies are Libsndfile [50] and DokuWiki [51]. These projects are both on our list of possible OSS projects, showed in Table 4.1, and fits our criteria of sufficient number of vulnerabilities over a 10 year period and a sufficient distribution of vulnerabilities in that period.Libsndfile is C library for reading and writing sound files [50], and has similarities to Libarchive through being a low-level C library with cross-platform support. DokuWiki is an open-source PHP wiki application [51]. Applying our model on these two OSS projects, we are able to test how the models applies to different project types.

5.2.1 Libsndfile

With the crawler described in Section 3.3 we found 18 Libsndfile vulnerabilities from the period between 2009 and 2020. These are all vulnerabilities with references to the Libsndfile GitHub project. After a manual review of these vulnerabilities and a control of the result against CVE Details [37] we included an additional 12 vulnerabilities in our case study. These are all Libsndfile vulnerabilities without any GitHub references. The list of the included Libsndfile vulnerabilities is found in Table C.1 in Appendix C. The results from the case study are summarised in Figure 5.2, and described in detail below.

Security culture & practices

Analysing the security culture and practices, we find no defined security culture in Libsndfile. There are no secure coding guidelines or listings of common vulnerability types, and no information sharing or learning attempts from previous vulnerabilities in the project. Also, we find no vulnerability disclosure policy in the project. From the 30 vulnerabilities in our list, we find that these are either reported directly to the lead developer or through the GitHub issue tracker. Last, we do find that Libsndfile has a test suite, and the project is included in the Google OSS-Fuzz project.

Application context

Studying the application context, we find that Libsndfile is a cross-platform application supporting Linux, macOS and Windows. From the documentation we see that Libsndfile use features specified by the 1999 ISO C standard, and that the

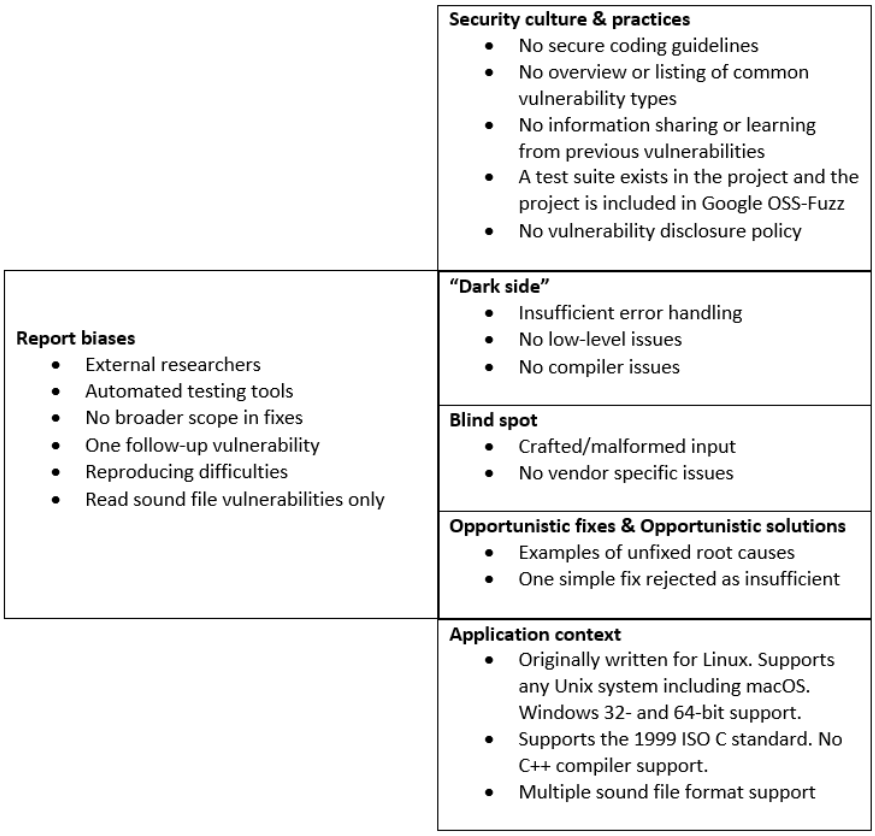

Figure 5.2: Case study Libsndfile

project actively does not support C++ compilers. The library support 26 different sound file formats, with several sub-formats [50].

Report biases

From the 30 reported Libsndfile vulnerabilities we find that these are reported by external security researchers or users of the library, and they are discovered using automated testing tools. In connection with this we find examples of difficulties reproducing the errors, and one vulnerability is marked as disputed due to such difficulties. There is also one follow-up vulnerability caused by an insufficient fix to a previous vulnerability. As we also observed in the security culture and practice's we see no broader scope or attempts in looking for similar security issues when fixing the vulnerabilities.

Vulnerability causing phenomena

Analysing the Libsndfile vulnerabilities using our model of vulnerability causing phenomena, we find no "Dark side" scenarios caused by low-level platform issues

like 32/64-bit issues and no compiler or C specification issues. The vulnerabilities are mainly blind spot scenarios caused by crafted or malformed sound files, to example sound files changing the number of channels in the middle of the file. There are vulnerabilities where an insufficient error handling was exploited. This falls into the "Dark side" phenomena caused by a difference between expected and actual behaviour. Of opportunistic fixes and solutions, we find one unfixed root cause but also on rejected fix due to being insufficient and completely fixing the security error. There are no opportunistic solutions causing vulnerabilities.

Summary

Overall, we see that the emergence and evolvement of vulnerabilities in Libsndfile cab be described using our model. We find that the security culture allows for the vulnerability causing phenomena to occur, we observe how these phenomena are influenced by the application context, we find report biases, and all vulnerabilities can be traced to one of the vulnerability causing phenomena. But though we find an application context influence through the multi-format support for sound files and blind spots caused by malformed archive files, we find less influence through low-level cross-platform issues or compiler issues. The latter can be attributed to the well-defined support for the 1999 ISO C standard. On the former we see that though we have 30 vulnerabilities in our list, and we find the usages of automated testing in these issues, Libsndfile is less thoroughly tested than Libarchive. We also see that even if Libsndfile is tested through Google OSS-Fuzz, none of the security errors found through this tool are assigned an CVE ID, and thus not part of our case study. An analysis including all security related issues in the issue tracker could therefore uncover such issues, and also uncover issues not covered by our model.

5.2.2 DokuWiki

Our crawler fro Section 3.3 returned 14 DokuWiki vulnerabilities in the time period 2009 to 2020. As for Libsndfile we performed a manual review of these vulnerabilities and controlled the result against CVE Details [37]. This returned an additional six DokuWiki vulnerabilities without any reference to the GitHub project. These were included in our study and the list of the vulnerabilities are found in Table C.1 in Appendix C. The results from the DokuWiki case study are summarised in Figure 5.3 and described in details below.

Security culture & practices

From our case study we find a mature security culture in DokuWiki. We find security guidelines both with regards to coding on installation and configuration of the application. The guidelines contain an overview of the most common vulnerability types and we find a general openness around security errors with published descriptions of the issue and release of security patches. In connection with this

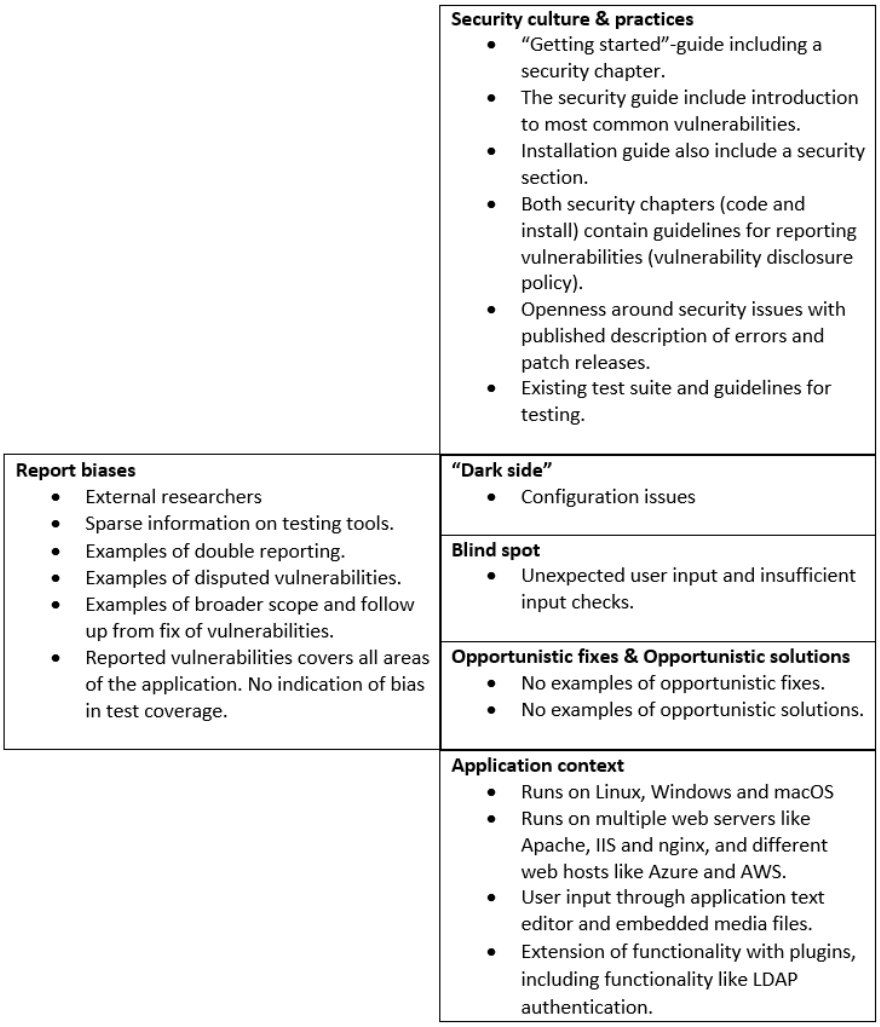

Figure 5.3: Case study DokuWiki

we also find an example of follow-up on a vulnerability where a vulnerability in the ACL plugin caused the developers to do a broader review of the plugin finding two additional vulnerabilities. There is also a DokuWiki test suite and guidelines around testing.

Application context

The application context analysis shows that DokuWiki is a cross-platform application both in terms of OS, web server and web host support. The main input to the application is user input through application text editor including embedded media files. There is also a plugin functionality where users can write their own DokuWiki plugins to extend functionality. These plugins also cover areas like LDAP authentication.

Report biases

Analysing possible report biases we find mainly external researchers doing the testing, but little information around test methods and tools. There are examples of double reporting and disputed vulnerabilities, but we find discussions and explanations around these issues and the vulnerabilities are not just closed as duplicates or "no error". We find vulnerabilities in several areas of the application, including configuration issues and there are no traces of bias with regards to test coverage.

Vulnerability causing phenomena

Analysing the 20 vulnerabilities against our model and the vulnerability causing phenomena we find no direct opportunistic fixes or solutions causing vulnerabilities. There is the release of the ACL plugin resulting in three vulnerabilities. From our data collection and analysis, we find this as result of insufficient testing more than a result of an opportunistic solution. The vulnerabilities mainly fall into the blind spot phenomena where unexpected input and missing or insufficient input checks (missing sanitation) causes injections. There are also some configuration issues in the vulnerabilities that can be categorised as "Dark side" phenomena where we find unexpected behaviour due to the unconsidered configuration issues.

Summary

These results shows that our model describes the DokuWiki vulnerabilities to some extent, but there are also areas less covered by our model. The model shows how a mature security policy influence the vulnerabilities. One example of this is the follow up we see from the developers on the ACL plugin vulnerability, resulting in the discovery of two related vulnerabilities in the same plugin. We also see this in the security guidelines and the general openness around vulnerabilities. We find less influence of report biases and the application context into the vulnerabilities. We find some double reporting and disputed vulnerabilities but not to an extent that causes security errors to be overlooked or taken less seriously. There are configuration related vulnerabilities, but none of these are due to cross-platform issues or other low-level issues.

When developing our model from the analysis of Libarchive vulnerabilities we defined the application context element in the model in terms OS support, file system support, 32/64-bit issues, etc. What we see from the DokuWiki case study is that this definition can be too narrow to fit a PHP web application, and that these types of issues to little extent applies to this type of application. The application type can in itself be a context issue, determining the types of vulnerabilities we are likely to find the application. As we have described above, we can also put the plugin functionality into the application context element. This is a way to extend

functionality without interfering with the core application, and as we have seen it can introduce vulnerable code as in the ACL plugin.

Another issue to consider is the vulnerabilities due to configuration issues. In the discussion above we categorised this a "Dark side" phenomena, due to these issues being details not considered by the developer and affecting differences between expected and actual behaviour. On the other hand, these are not programming related vulnerabilities and an argument can be made for these issues to be treated separately from our defined phenomena. To some extent these configuration issues are part of the application context and is an influencing or input item into the vulnerability causing phenomena. In a web application like DokuWiki configuration and setup is more of an influence into application vulnerabilities than in low-level applications like Libarchive and Libsndfile. If we also should threat the configuration issues as separate vulnerability causing phenomena is a question for further work and analysis.

As for Libsndfile, an analysis including all security related issues in the issue tracker and not only vulnerabilities assigned an CVE ID could also have uncovered more report bias related issues or other elements or phenomena not covered by our model.

5.3 Comparison to Pieczul & Foley

A similar study as ours was performed by Pieczul and Foley [7]. The study analysed the evolution of a security control in the Apache Struts OSS package and presented a model describing how vulnerabilities emerge and evolved based on their results. We have referred to the work in [7] in our analysis of the vulnerability history in Libarchive, and for completeness we include a comparison between our model and the model from [7] depicted in Figure 5.4.

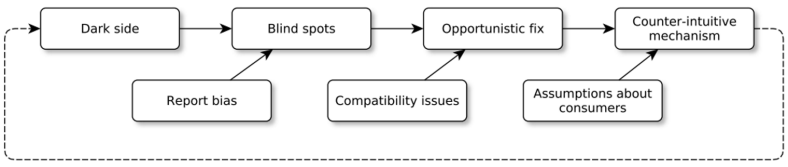

Figure 5.4: Vulnerability model by Pieczul and Foley [7]

First, there are differences in the methods used in developing the two models. In our model we have identified phenomena causing vulnerabilities and role of the STS into these phenomena. This is the result of the analysis of collected artefacts related to or data set of reported Libarchive security vulnerabilities. The model by Pieczul and Foley [7] was developed from the result of analysing the evolution of one security control in the Apache Struts OSS package. This security control had several reported security vulnerabilities, and the analysis was done on collected artefacts related to the evolution of the security control component [7]. As in our study the artefacts were security reports, code changes, discussions in issue

trackers, etc. but with focus on the whole evolution of the security component and not only on the security vulnerabilities.

Comparing the two models we find the "Dark side", blind spot and opportunistic fix phenomena in both our model and in [7]. Under the opportunistic fix phenomena, Pieczul and Foley [7] also observed how compatibility issues played a part in causing the opportunistic fix through sub-optimal solutions to make the fix compatible with older versions [7]. We have not observed this in our study. What we have seen in our analysis is that the Libarchive vulnerabilities exists in the lower levels of the library, whereas the API has remained unchanged through the versions included in our analysis. This makes compatibility issues less likely to occur. In addition to this, our model also includes opportunistic solutions as a separate phenomena. We have observed how choosing easy solutions or solutions fitting the existing code also applies to implementation of new functionality and not only fixes in old code. These types of opportunistic solutions can also cause vulnerable code.

Another phenomena found in [7] but not in our model is "Counter-intuitive mechanisms" and in connection with this, "Assumptions about consumers". This is a phenomenon where the code in itself is not vulnerable in itself, but the solution and correct usage is difficult to understand. Thus, incorrect usage might cause their own vulnerabilities. Contributing to this an incorrect assumption about consumers understanding about security mechanisms [7]. As for the compatibility issues, the low-level existence of the vulnerabilities and the well-defined API causes this phenomena less likely to occur in our analysis. We have not found issues where wrong usage of the library caused the vulnerability.

The last element in the model by Pieczul and Foley [7] is the Report bias. In their model the report bias is observed as a "Dark side" when documenting or interpreting a vulnerability report. The researcher might not fully understand all implications of the vulnerable code and the developer might limit the scope to the vulnerability report when implementing the fix [7]. We also observed report biases in our analysis, and in addition to the limited scope in writing and interpreting the vulnerability reports we did also observe how the usage of automated testing causing report biases. In our analysis we have seen how fuzzing causes double reporting of vulnerabilities, how limited vulnerability reports mostly containing trace or log output causes difficulties in reproducing the errors, how the trace and log output from the testing tools can make it easier to dismiss vulnerabilities with different root causes as duplicates, and we have also seen a possible bias in test coverage due to the use of tools like fuzzers. Through the analysis of the STS, we did also find how the report bias was caused by the structure and methods in the STS model, with external researchers and the usage of automated testing tools. We therefore put the report bias as an input into the vulnerability causing phenomena in our model.

In addition to the report bias, our STS analysis found how the security culture and practices, and the application context influence the vulnerability causing phenomena of "Dark side", blind spot and opportunistic fixes and solutions. These

are elements not found in the model by Pieczul and Foley [7]. The STS analysis is a new element added to the vulnerability model in our study, and together with the differences in methodology and the differences in the OSS packages analysed we see how these differences effect the results analysis and the resulting models.

Chapter 6

Conclusion and further work

6.1 Conclusion

In this thesis project we have presented a model describing how vulnerabilities emerge and evolve in an OSS package. This model is developed from analysis of artefacts related to vulnerabilities in the Libarchive OSS project in the period between 2009 and 2020. The model answers our research question of how vulnerabilities emerge and evolve and what insight can be gained into this question from the related artefacts.

In developing the model, we have studied the patterns and phenomena behind the vulnerabilities, and the STS surrounding the vulnerability handling in the OSS package. Our model shows how the security culture and practices, the application context and report biases in security test coverage or in writing or interpreting vulnerability reports serves as an input or influence into vulnerability causing phenomena of "Dark side", blind spots and opportunistic fixes and opportunistic solutions. In addition to our vulnerability model, we also presented a memory safety taxonomy from our analysis of patterns behind the vulnerabilities. This taxonomy builds on the buffer overflow taxonomy by Schuckert *et al.* [1] by expanding this to include other memory related vulnerabilities.

Together, the model and taxonomy serve as tools to understand how vulnerabilities emerge and evolve. The taxonomy give insight into the specifics of how and where memory vulnerabilities occur and how to fix these, whereas the model give a more general understanding about what causes the vulnerabilities to occur in the development process and the influence of the STS into these phenomena. Both artefacts can be used to broaden the understanding around the topic of vulnerable code and improve the development process to increase security.

6.2 Further work

In this project we have used an iterative approach in the analysis of the vulnerabilities in our collected data set. We first analysed patterns behind the vulnerabilities,

then performed an analysis to find broader phenomena describing how vulnerabilities emerge and evolve in the OSS package, and last an analysis into the STS around the vulnerability handling in the OSS package. Each iteration increased our detail level into the knowledge of how vulnerability emerge and evolve in code, and resulted in the memory safety taxonomy and the vulnerability model. To test the model, we performed two case studies applying the model to two other OSS packages. Though we did find that our model described the vulnerabilities history in these two projects, our results also showed a better fit between our model and Libsndfile than with DokuWiki. This can be explained by Libsndfile bearing similarities to Libarchive both being C programs related to file handling, whereas DokuWiki being a PHP web application. The DokuWiki case study showed a possible extension to the application context element in our model, and we also found a set of vulnerabilities related to configuration issues. Though these could be defined as vulnerabilities caused by the "Dark side" phenomena, it can also be argued the threat these as caused as a separate element in the model. We also performed a comparison between our model and another vulnerability model by Pieczul and Foley [7], which showed differences between observed patterns and phenomena behind vulnerabilities. From our iterative approach in the analysis, the result from the case studies, and the comparison between similar models we see that a next step in this work is to preform additional analysis of vulnerability history in other OSS packages to expend and increase the detail level in our proposed model.

Other further work is a detailed study into how the proposed model can be used to improve software security. We see that the model can be used to gain knowledge into the different factors causing vulnerabilities, but such a study should focus on how the model could improve the Software Development Life Cycle (SDLC) to prevent vulnerabilities to occur in the first place.

Bibliography

[1] F. Schuckert, M. Hildner, B. Katt and H. Langweg, 'Source code patterns of buffer overflow vulnerabilities in firefox', in *SICHERHEIT 2018*, H. Langweg, M. Meier, B. C. Witt and D. Reinhardt, Eds., Bonn: Gesellschaft für Informatik e.V., 2018, pp. 107–118. DOI: 10.18420/sicherheit2018_08.

[2] *Libarchive*, https://github.com/libarchive/libarchive, Last accessed on 2021-01-06, 2021.

[3] Microsoft, *Security development lifecycle*, https://www.microsoft.com/en-us/securityengineering/sdl/, Last accessed on 2020-03-08, 2019.

[4] OWASP, *Owasp samm*, https://owasp.org/www-project-samm/, Last accessed on 2020-03-08, 2019.

[5] *Owasp top ten*, https://owasp.org/www-project-top-ten/, Last accessed on 2020-03-07, 2017.

[6] *Cwe top 25*, https://cwe.mitre.org/data/definitions/1200.html/, Last accessed on 2020-03-07, 2019.

[7] O. Pieczul and S. N. Foley, 'The evolution of a security control', in *Security Protocols XXIV*, 2016, pp. 67–84. DOI: 10.1007/978-3-319-62033-6_9.

[8] *The heartbleed bug*, https://heartbleed.com/, Last accessed on 2020-03-10, 2014.

[9] F. Massacci, S. Neuhaus and V. H. Nguyen, 'After-life vulnerabilities: A study on firefox evolution, its vulnerabilities, and fixes', in *ngineering Secure Software and Systems*, 2011, pp. 195–208.

[10] N. Munaiah, F. Camilo, W. Wigham, A. Meneely and M. Nagappan, 'Do bugs foreshadow vulnerabilities? an in-depth study of the chromium project', *Empirical Software Engineering*, vol. 22, pp. 1305–1347, 2017.

[11] C. Thompson and D. A. Wagner, 'A large-scale study of modern code review and security in open source projects', in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE, Toronto, Canada: Association for Computing Machinery, 2017, pp. 83–92, ISBN: 9781450353052. DOI: 10.1145/3127005.3127014. [Online]. Available: https://doi.org/10.1145/3127005.3127014.

[12] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary and B. Spates, 'When a patch goes bad: Exploring the properties of vulnerability-contributing commits', *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 65–74, 2013.

[13] A. Ozment and S. E. Schechter, 'Milk or wine: Does software security improve with age?', in *In USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, USENIX Association, 2006.

[14] Y. Shin, A. Meneely, L. A. Williams and J. A. Osborne, 'Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities', *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, Nov. 2011, ISSN: 2326-3881. DOI: 10.1109/TSE.2010.81.

[15] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios and D. Spinellis, 'Dismal code: Studying the evolution of security bugs', in *USENIX Association LASER 2013*, 2013.

[16] N. Edwards and L. Chen, 'An historical examination of open source releases and their vulnerabilities', in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 183–194, ISBN: 9781450316514. DOI: 10.1145/2382196.2382218. [Online]. Available: https://doi.org/10.1145/2382196.2382218.

[17] K. H. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy and A. K. Ghose, 'Automatic feature learning for vulnerability prediction', *ArXiv*, vol. abs/1708.02368, 2017.

[18] Y. Pang, X. Xue and A. S. Namin, 'Predicting vulnerable software components through n-gram analysis and statistical feature selection', *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pp. 543–548, 2015.

[19] M. Jimenez, M. Papadakis and Y. L. Traon, 'Vulnerability prediction models: A case study on the linux kernel', *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 1–10, 2016.

[20] P. Morrison, K. Herzig, B. Murphy and L. A. Williams, 'Challenges with applying vulnerability prediction models', in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, ser. HotSoS '15, Urbana, Illinois: Association for Computing Machinery, 2015, ISBN: 9781450333764. DOI: 10.1145/2746194.2746198. [Online]. Available: https://doi.org/10.1145/2746194.2746198.

[21] F. Schuckert, B. Katt and H. Langweg, 'Source code patterns of sql injection vulnerabilities', in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ser. ARES '17, Reggio Calabria, Italy: Association for Computing Machinery, 2017, ISBN: 9781450352574. DOI:

10.1145/3098954.3103173. [Online]. Available: https://doi.org/10.1145/3098954.3103173.

[22] F. Schuckert, M. Hildner, B. Katt and H. Langweg, 'Source code patterns of cross site scripting in php open source projects', *Norsk Informasjonssikkerhetskonferanse (NISK)*, 2018.

[23] C. Wressnegger, F. Yamaguchi, A. Maier and K. Rieck, 'Twice the bits, twice the trouble: Vulnerabilities induced by migrating to 64-bit platforms', in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 541–552, ISBN: 9781450341394. DOI: 10.1145/2976749.2978403. [Online]. Available: https://doi.org/10.1145/2976749.2978403.

[24] W. Dietz, P. Li, J. Regehr and V. S. Adve, 'Understanding integer overflow in c/c++', in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, Zurich, Switzerland: IEEE Press, 2012, pp. 760–770, ISBN: 9781467310673.

[25] Z. Gu, J. Wu, J. Liu, M. Zhou and M. Gu, 'An empirical study on api-misuse bugs in open-source c programs', *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 11–20, 2019. DOI: 10.1109/COMPSAC.2019.00012.

[26] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. DeLong, J. Cappos and Y. Brun, 'Api blindspots: Why experienced developers write vulnerable code', in *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 315–328, ISBN: 978-1-939133-10-6. [Online]. Available: https://www.usenix.org/conference/soups2018/presentation/oliveira.

[27] O. Pieczul and S. N. Foley, 'The dark side of the code', in *Security Protocols XXIII*, Cham: Springer International Publishing, 2015, pp. 1–11.

[28] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos and Y. Zhuang, '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*, ser. ACSAC '14, New Orleans, Louisiana, USA: Association for Computing Machinery, 2014, pp. 296–305, ISBN: 9781450330053. DOI: 10.1145/2664243.2664254. [Online]. Available: https://doi.org/10.1145/2664243.2664254.

[29] O. Pieczul, S. N. Foley and M. E. Zurko, 'Developer-centered security and the symmetry of ignorance', in *Proceedings of the 2017 New Security Paradigms Workshop*, ser. NSPW 2017, Santa Cruz, CA, USA: Association for Computing Machinery, 2017, pp. 46–56, ISBN: 9781450363846. DOI: 10.1145/3171533.3171539. [Online]. Available: https://doi.org/10.1145/3171533.3171539.

[30] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek and M. Hicks, 'Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it', in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, 2019.

[31] E. LeMay, K. Scarfone and P. M. Mell, 'The common misuse scoring system (cmss): Metrics for software feature misuse vulnerabilities', in *NIST Interagency Report 7864*, National Institute of Standards and Technology (NIST), 2012.

[32] FIRST, *Common vulnerability scoring system version 3.1 specification document*, `https://www.first.org/cvss/v3.1/specification-document`, Last accessed on 2021-01-06, 2019.

[33] P. Mell, K. Scarfone and S. Romanosky, *A complete guide to the common vulnerability scoring system version 2.0*, `https://www.first.org/cvss/v2/guide`, Last accessed on 2021-01-06, 2007.

[34] tenable, *Predictive prioritization: How to focus on the vulnerabilities that matter most*, `https://www.tenable.com/whitepapers/predictive-prioritization-how-to-focus-on-the-vulnerabilities-that-matter-most`, Last accessed on 2020-04-11, 2019.

[35] T. W. Edgar and D. O. Manz, 'Chapter 4 - exploratory study', in *Research Methods for Cyber Security*, T. W. Edgar and D. O. Manz, Eds., Syngress, 2017, pp. 95–130, ISBN: 978-0-12-805349-2. DOI: `https://doi.org/10.1016/B978-0-12-805349-2.00004-2`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/B9780128053492000042`.

[36] *Common vulnerabilities and exposures*, `https://cve.mitre.org/`, Last accessed on 2020-04-14, 2020.

[37] *Cve details*, `https://www.cvedetails.com/`, Last accessed on 2020-04-14, 2020.

[38] *Github*, `https://github.com/`, Last accessed on 2020-04-14, 2020.

[39] S.-F. Wen, M. Kianpour and B. Katt, 'Security knowledge management in open source software communities', in *Innovative Security Solutions for Information Technology and Communications*, Springer International Publishing, 2019, pp. 53–70, ISBN: 978-3-030-12942-2.

[40] S. Kowalski, *IT insecurity: A multi-discipline inquiry. Ph.D. thesis*. Sweden: Department of Computer, System Sciences, University of Stockholm and Royal Institute of Technology, 1994, ISBN: 91-7153-207-2.

[41] S.-F. Wen and S. Kowalski, 'A case study: Heartbleed vulnerability management and swedish municipalities', in *Human Aspects of Information Security, Privacy and Trust*, Springer International Publishing, 2017, pp. 414–431, ISBN: 978-3-319-58460-7.

[42] *Common weakness enumeration*, `https://cwe.mitre.org/`, Last accessed on 2021-01-30, 2021.

[43] *C programming language*, https://devdocs.io/c/, Last accessed on 2021-05-14, 2021.

[44] E. International, *Volume and file structure of cdrom for information interchange*, https://www.ecma-international.org/wp-content/uploads/ECMA-119_4th_edition_june_2019.pdf, Last accessed on 2021-04-12, 2019.

[45] OWASP, *Fuzzing*, https://owasp.org/www-community/Fuzzing, Last accessed on 2021-04-14, 2021.

[46] B. A. Sabbagh and S. Kowalski, 'Developing social metrics for security modeling the security culture of it workers individuals (case study)', *The 5th International Conference on Communications, Computers and Applications (MIC-CCA2012)*, pp. 112–118, 2012.

[47] S.-F. Wen, M. Kianpour and S. Kowalski, 'An empirical study of security culture in open source software communities', *2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 863–870, 2019. DOI: 10.1145/3341161.3343520. [Online]. Available: https://doi.org/10.1145/3341161.3343520.

[48] *OSS-Fuzz*, https://google.github.io/oss-fuzz/, Last accessed on 2021-05-05, 2021.

[49] *Rar version 3.00 - technical information*, http://www.iesleonardo.com/ele/gs/Utilidades/WinRAR/TechNote.txt, Last accessed on 2021-04-19, 2021.

[50] *Libsndfile*, https://libsndfile.github.io/libsndfile/, Last accessed on 2021-04-20, 2021.

[51] *Dokuwiki*, https://www.dokuwiki.org/dokuwiki, Last accessed on 2021-04-20, 2021.

[52] *National vulnerability database*, https://nvd.nist.gov/, Last accessed on 2021-01-14, 2020.

Appendix A

Libarchive vulnerabilities

Table A.1 contains the total list of the 53 Libarchive vulnerabilities analysed in this thesis project.

Table A.1: Libarchive vulnerabilities [36, 37, 52]

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2011-1777	Multiple buffer overflows in the (1) heap_ add_ entry and (2) relocate_dir functions in archive_ read_ support_ format_ iso9660.c in libarchive through 2.8.5 allow remote attackers to cause a denial of service (application crash) or possibly execute arbitrary code via a crafted ISO9660 image.	Denial Of Service, Execute Code, Overflow	119	6.8 Medium AV:N/AC:M/Au:N/C:P/I:P/A:P	NA
CVE-2011-1778	Buffer overflow in libarchive through 2.8.5 allows remote attackers to cause a denial of service (application crash) or possibly execute arbitrary code via a crafted TAR archive.	Denial Of Service, Execute Code, Overflow	119	6.8 Medium AV:N/AC:M/Au:N/C:P/I:P/A:P	NA
CVE-2013-0211	Integer signedness error in the archive_ write_ zip_ data function in archive_ write_ set_ format_ zip.c in libarchive 3.1.2 and earlier, when running on 64-bit machines, allows context-dependent attackers to cause a denial of service (crash) via unspecified vectors, which triggers an improper conversion between unsigned and signed types, leading to a buffer overflow.	Denial Of Service, Overflow	189	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	NA
CVE-2015-2304	Absolute path traversal vulnerability in bsdcpio in libarchive 3.1.2 and earlier allows remote attackers to write to arbitrary files via a full pathname in an archive.	Directory traversal	22	6.4 Medi-umAV:N/AC:L/Au:N/C:N/I:P/A:P	NA
CVE-2015-8915	bsdcpio in libarchive before 3.2.0 allows remote attackers to cause a denial of service (invalid read and crash) via crafted cpio file.	Denial Of Service	125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H

Continued on next page

Table A.1 – continued from previous page

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2015-8916	bsdtar in libarchive before 3.2.0 returns a success code without filling the entry when the header is a split file in multivolume RAR which allows remote attackers to cause a denial of service (NULL pointer dereference and crash) via a crafted rar file.	Denial Of Service	476	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	6.5 Medium AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2015-8917	bsdtar in libarchive before 3.2.0 allows remote attackers to cause a denial of service (NULL pointer dereference and crash) via an invalid character in the name of a cab file.	Denial Of Service	476	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
CVE-2015-8918	The archive_ string_ append function in archive_ string.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (crash) via a crafted cab files, related to overlapping memcpy.	Denial Of Service, Overflow	119	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
CVE-2015-8919	The lha_ read_ file_ extended_ header function in archive_ read_ support_ format_ lha.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (out-of-bounds heap) via a crafted (1) lzh or (2) lha file.	Denial Of Service,Overflow	119	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
CVE-2015-8920	The _ ar_ read_ header function in archive_ read_ support_ format_ ar.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (out-of-bounds stack read) via a crafted ar file.	Denial Of Service	125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2015-8921	The ae_ strtofflags function in archive_ entry.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (out-of-bounds read) via a crafted mtree file.	Denial Of Service	125	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

Continued on next page

Table A.1 – continued from previous page

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2015-8922	The read_ CodersInfo function in archive_ read_ support_ format_ 7zip.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (NULL pointer dereference and crash) via a crafted 7z file, related to the _ 7z_ folder struct.	Denial Of Service	476	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2015-8923	The process_ extra function in libarchive before 3.2.0 uses the size field and a signed number in an offset, which allows remote attackers to cause a denial of service (crash) via a crafted zip file.	Denial Of Service	20	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	6.5 Medium AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2015-8924	The archive_ read_ format_ tar_ read_ header function in archive_ read_ support_ format_ tar.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (out-of-bounds read) via a crafted tar file.	Denial Of Service	125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2015-8925	The readline function in archive_ read_ support_ format_ mtree.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (invalid read) via a crafted mtree file, related to newline parsing.	Denial Of Service	125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2015-8926	The archive_ read_ format_ rar_ read_ data function in archive_ read_ support_ format_ rar.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (crash) via a crafted rar archive.	Denial Of Service	476	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H

Continued on next page

Table A.1 – continued from previous page

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2015-8927	The trad_ enc_ decrypt_ update function in archive_ read_ support_ format_ zip.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (out-of-bounds heap read and crash) via a crafted zip file, related to reading the password.	Denial Of Service	125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2015-8928	The process_ add_ entry function in archive_ read_ support_ format_ mtree.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (out-of-bounds read) via a crafted mtree file.	Denial Of Service	125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2015-8929	Memory leak in the _ _ archive_ read_ get_ extract function in archive_ read_ extract2.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service via a tar file.	Denial Of Service, Overflow	119	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2015-8930	bsdtar in libarchive before 3.2.0 allows remote attackers to cause a denial of service (infinite loop) via an ISO with a directory that is a member of itself.	Denial Of Service	20	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
CVE-2015-8931	Multiple integer overflows in the (1) get_ time_ t_ max and (2) get_ time_ t_ min functions in archive_ read_ support_ format_ mtree.c in libarchive before 3.2.0 allow remote attackers to have unspecified impact via a crafted mtree file, which triggers undefined behavior.	Overflow	190	6.8 Medium AV:N/AC:M/Au:N/C:P/I:P/A:P	7.8 High AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H
CVE-2015-8932	The compress_ bidder_ init function in archive_ read_ support_ filter_ compress.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (crash) via a crafted tar file, which triggers an invalid left shift.	Denial Of Service	20	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H

Continued on next page

Table A.1 – continued from previous page

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2015-8933	Integer overflow in the archive_ read_ format_ tar_ skip function in archive_ read_ support_ format_ tar.c in libarchive before 3.2.0 allows remote attackers to cause a denial of service (crash) via a crafted tar file.	Denial Of Service, Overflow	190	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2015-8934	The copy_ from_ lzss_ window function in archive_ read_ support_ format_ rar.c in libarchive 3.2.0 and earlier allows remote attackers to cause a denial of service (out-of-bounds heap read) via a crafted rar file.	Denial Of Service	125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2016-10209	The archive_ wstring_ append_ from_ mbs function in archive_ string.c in libarchive 3.2.2 allows remote attackers to cause a denial of service (NULL pointer dereference and application crash) via a crafted archive file.	Denial Of Service	476	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2016-10349	The archive_ le32dec function in archive_ endian.h in libarchive 3.2.2 allows remote attackers to cause a denial of service (heap-based buffer over-read and application crash) via a crafted file.	Denial Of Service, Overflow	119	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2016-10350	The archive_ read_ format_ cab_ read_ header function in archive_ read_ support_ format_ cab.c in libarchive 3.2.2 allows remote attackers to cause a denial of service (heap-based buffer over-read and application crash) via a crafted file.	Denial Of Service, Overflow	119	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2016-1541	Heap-based buffer overflow in the zip_ read_ mac_ metadata function in archive_ read_ support_ format_ zip.c in libarchive before 3.2.0 allows remote attackers to execute arbitrary code via crafted entry-size values in a ZIP archive.	Execute Code, Overflow	20	6.8 Medium AV:N/AC:M/Au:N/C:P/I:P/A:P	8.8 High AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

Continued on next page

Table A.1 – continued from previous page

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2016-4300	Integer overflow in the read_ SubStreamsInfo function in archive_ read_ support_ format_ 7zip.c in libarchive before 3.2.1 allows remote attackers to execute arbitrary code via a 7zip file with a large number of substreams, which triggers a heap-based buffer overflow.	Execute Code, Overflow	190	6.8 Medium AV:N/AC:M/Au:N/C:P/I:P/A:P	7.8 High AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H
CVE-2016-4301	Stack-based buffer overflow in the parse_ device function in archive_ read_ support_ format_ mtree.c in libarchive before 3.2.1 allows remote attackers to execute arbitrary code via a crafted mtree file.	Execute Code, Overflow	119	6.8 Medium AV:N/AC:M/Au:N/C:P/I:P/A:P	7.8 High AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H
CVE-2016-4302	Heap-based buffer overflow in the parse_ codes function in archive_ read_ support_ format_ rar.c in libarchive before 3.2.1 allows remote attackers to execute arbitrary code via a RAR file with a zero-sized dictionary.	Execute Code, Overflow	119	6.8 Medium AV:N/AC:M/Au:N/C:P/I:P/A:P	7.8 High AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H
CVE-2016-4809	The archive_ read_ format_ cpio_ read_ header function in archive_ read_ support_ format_ cpio.c in libarchive before 3.2.1 allows remote attackers to cause a denial of service (application crash) via a CPIO archive with a large symlink.	Denial Of Service	20	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
CVE-2016-5418	The sandboxing code in libarchive 3.2.0 and earlier mishandles hardlink archive entries of non-zero data size, which might allow remote attackers to write to arbitrary files via a crafted archive file.		20, 19	5.0 Medium AV:N/AC:L/Au:N/C:N/I:P/A:N	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:N
CVE-2016-5844	Integer overflow in the ISO parser in libarchive before 3.2.1 allows remote attackers to cause a denial of service (application crash) via a crafted ISO file.	Denial Of Service, Overflow	190	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	6.5 Medium AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H

Continued on next page

Table A.1 – continued from previous page

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2016-6250	Integer overflow in the ISO9660 writer in libarchive before 3.2.1 allows remote attackers to cause a denial of service (application crash) or execute arbitrary code via vectors related to verifying filename lengths when writing an ISO9660 archive, which trigger a buffer overflow.	Denial Of Service, Execute Code, Overflow	190	7.5 High AV:N/AC:L/Au:N/C:P/I:P/A:P	8.6 High AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:H
CVE-2016-7166	libarchive before 3.2.0 does not limit the number of recursive decompressions, which allows remote attackers to cause a denial of service (memory consumption and application crash) via a crafted gzip file.	Denial Of Service	399	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2016-8687	Stack-based buffer overflow in the safe_ fprintf function in tar/util.c in libarchive 3.2.1 allows remote attackers to cause a denial of service via a crafted non-printable multibyte character in a filename.	Denial Of Service, Overflow	119	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
CVE-2016-8688	The mtree bidder in libarchive 3.2.1 does not keep track of line sizes when extending the read-ahead, which allows remote attackers to cause a denial of service (crash) via a crafted file, which triggers an invalid read in the (1) detect_ form or (2) bid_ entry function in libarchive/archive_ read_ support_ format_ mtree.c.	Denial Of Service	125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2016-8689	The read_ Header function in archive_ read_ support_ format_ 7zip.c in libarchive 3.2.1 allows remote attackers to cause a denial of service (out-of-bounds read) via multiple EmptyStream attributes in a header in a 7zip archive.	Denial Of Service	125	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

Continued on next page

Table A.1 – continued from previous page

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2017-14166	libarchive 3.3.2 allows remote attackers to cause a denial of service (xml_ data heap-based buffer over-read and application crash) via a crafted xar archive, related to the mishandling of empty strings in the atol8 function in archive_ read_ support_ format_ xar.c.	Denial Of Service	125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	6.5 Medium AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2017-14501	An out-of-bounds read flaw exists in parse_ file_ info in archive_ read_ support_ format_ iso9660.c in libarchive 3.3.2 when extracting a specially crafted iso9660 iso file, related to archive_ read_ format_ iso9660$_{r}ead_{h}eader$.		125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	6.5 Medium AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2017-14502	read_ header in archive_ read_ support_ format_ rar.c in libarchive 3.3.2 suffers from an off-by-one error for UTF-16 names in RAR archives, leading to an out-of-bounds read in archive_ read_ format_ rar_ read_ header.		125, 193	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
CVE-2017-14503	libarchive 3.3.2 suffers from an out-of-bounds read within lha_ read_ data_ none() in archive_ read_ support_ format_ lha.c when extracting a specially crafted lha archive, related to lha_ crc16.		125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	6.5 Medium AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2017-5601	An error in the lha_ read_ file_ header_ 1() function (archive_ read_ support_ format_ lha.c) in libarchive 3.2.2 allows remote attackers to trigger an out-of-bounds read memory access and subsequently cause a crash via a specially crafted archive.		125	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

Continued on next page

Table A.1 – continued from previous page

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2018-1000877	libarchive version commit 416694915449219d5055 31b1096384f3237dd6cc onwards (release v3.1.0 onwards) contains a CWE-415: Double Free vulnerability in RAR decoder - libarchive/archive_ read_ support_ format_ rar.c, parse_ codes(), realloc(rar ->lzss.window, new_ size) with new_ size = 0 that can result in Crash/DoS. This attack appear to be exploitable via the victim must open a specially crafted RAR archive.		415	6.8 Medium AV:N/AC:M/Au:N/C:P/I:P/A:P	8.8 High AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H
CVE-2018-1000878	libarchive version commit 416694915449219d5055 31b1096384f3237dd6cc onwards (release v3.1.0 onwards) contains a CWE-416: Use After Free vulnerability in RAR decoder - libarchive/archive_ read_ support_ format_ rar.c that can result in Crash/DoS - it is unknown if RCE is possible. This attack appear to be exploitable via the victim must open a specially crafted RAR archive.		416	6.8 Medium AV:N/AC:M/Au:N/C:P/I:P/A:P	8.8 Hihg AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H
CVE-2018-1000879	libarchive version commit 379867ecb330b3a952fb 7bfa7bffb7bbd5547205 onwards (release v3.3.0 onwards) contains a CWE-476: NULL Pointer Dereference vulnerability in ACL parser - libarchive/archive_ acl.c, archive_ acl_ from_ text_ l() that can result in Crash/DoS. This attack appear to be exploitable via the victim must open a specially crafted archive file.		476	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	6.5 Medium AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H

Continued on next page

Table A.1 – continued from previous page

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2018-1000880	libarchive version commit 9693801580c0cf7c70e8 62d305270a16b52826a7 onwards (release v3.2.0 onwards) contains a CWE-20: Improper Input Validation vulnerability in WARC parser - libarchive/archive_ read_ support_ format_ warc.c, _ warc_ read() that can result in DoS - quasi-infinite run time and disk usage from tiny file. This attack appear to be exploitable via the victim must open a specially crafted WARC file.		119	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	6.5 Medium AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2019-1000019	libarchive version commit bf9aec176c6748f0ee7a 678c5f9f9555b9a757c1 onwards (release v3.0.2 onwards) contains a CWE-125: Out-of-bounds Read vulnerability in 7zip decompression, archive_ read_ support_ format_ 7zip.c, header_ bytes() that can result in a crash (denial of service). This attack appears to be exploitable via the victim opening a specially crafted 7zip file.	Denial Of Service	125	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	6.5 Medium AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2019-1000020	libarchive version commit 5a98dcf8a86364b3c2c4 69c85b93647dfb139961 onwards (version v2.8.0 onwards) contains a CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop') vulnerability in ISO9660 parser, archive_ read_ support_ format_ iso9660.c, read_ CE()/parse_ rockridge() that can result in DoS by infinite loop. This attack appears to be exploitable via the victim opening a specially crafted ISO9660 file.		835	4.3 Medium AV:N/AC:M/Au:N/C:N/I:N/A:P	6.5 Medium AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H

Continued on next page

Table A.1 – continued from previous page

CVE ID	Description	Category	CWE	CVSS2	CVSS3
CVE-2019-18408	archive_ read_ format_ rar_ read_ data in archive_ read_ support_ format_ rar.c in libarchive before 3.4.0 has a use-after-free in a certain ARCHIVE_ FAILED situation, related to Ppmd7_ DecodeSymbol.		416	5.0 Medium AV:N/AC:L/Au:N/C:N/I:N/A:P	7.5 High AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
CVE-2019-19221	In Libarchive 3.4.0, archive_ wstring_ append_ from_ mbs in archive_ string.c has an out-of-bounds read because of an incorrect mbrtowc or mbtowc call. For example, bsdtar crashes via a crafted archive.		125	2.1 Low AV:L/AC:L/Au:N/C:N/I:N/A:P	5.5 Medium AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H
CVE-2020-9308	archive_ read_ support_ format_ rar5.c in libarchive before 3.4.2 attempts to unpack a RAR5 file with an invalid or corrupted header (such as a header size of zero), leading to a SIGSEGV or possibly unspecified other impact.		20	6.8 Medium AV:N/AC:M/Au:N/C:P/I:P/A:P	8.8 High AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

Appendix B

Libarchive vulnerability timeline

Table B.1: Libarchive vulnerability timeline, full version

Version	Date	Introduced	Fixed	Comment
2.8.5	Sept. 2011	TAR: CVE-2011-1778, CVE-2015-8924, CVE-2015-8932 RAR: CVE-2015-8916 ISO9660: CVE-2011-1777, CVE-2015-8930, CVE-2016-5844, CVE-2017-14501, CVE-2019-1000020 ZIP: CVE-2013-0211 CPIO: CVE-2015-8915, CVE-2015-2304, CVE-2016-4809 CAB: CVE-2015-8917, CVE-2015-8918 MTREE: CVE-2015-8921, CVE-2015-8925 Other: CVE-2015-8920, CVE-2016-7166, CVE-2017-14166 General: CVE-2016-5418, CVE-2016-8687, CVE-2016-10209, CVE-2019-19221		Initial release in analysis

Continued on next page

Table B.1 – continued from previous page

Version	Date	Introduced	Fixed	Comment
3.0.0a	Nov. 2011	RAR: CVE-2015-8926, CVE-2015-8934, CVE-2016-4302, CVE-2017-14502, CVE-2018-1000877, CVE-2019-18408 ISO9660: CVE-2016-6250 ZIP: CVE-2015-8923 LHA: CVE-2015-8919, CVE-2017-5601, CVE-2017-14503 MTREE: CVE-2015-8931, CVE-2016-8688 CAB: CVE-2016-10349, CVE-2016-10350	TAR: CVE-2011-1778 ISO9660: CVE-2011-1777	Support for archive formats RAR, CAB and LHA added in this version.
3.0.1b	Nov. 2011	7Zip: CVE-2015-8922, CVE-2016-4300, CVE-2016-8689, CVE-2019-1000019		Support for achive format 7Zip added in this version
3.1.0	Jan. 2013	RAR: CVE-2018-1000878 ZIP: CVE-2016-1541 MTREE: CVE-2015-8928		

Continued on next page

Table B.1 – continued from previous page

Version	Date	Introduced	Fixed	Comment
3.1.900a	Feb. 2016	MTREE: CVE-2016-4301 Other: CVE-2018-1000880	TAR: CVE-2015-8924, CVE-2015-8929, CVE-2015-8932, CVE-2015-8933 RAR: CVE-2015-8926, CVE-2015-8916 ISO9660: CVE-2015-8930 ZIP: CVE-2013-0211, CVE-2015-8923, CVE-2015-8927 CPIO: CVE-2015-2304, CVE-2015-8915 CAB: CVE-2015-8917, CVE-2015-8918 LHA: CVE-2015-8919 METREE: CVE-2015-8921, CVE-2015-8925, CVE-2015-8928, CVE-2015-8931 7Zip: CVE-2015-8922 Other: CVE-2015-8920, CVE-2016-7166	Support for achive format WARC added in this version
3.2.0	April 2016		ZIP: CVE-2016-1541	
3.2.1	Jun 2016		RAR: CVE-2015-8934, CVE-2016-4302 ISO9660: CVE-2016-5844, CVE-2016-6250 CPIO: CVE-2016-4809 MTREE: CVE-2016-4301 7Zip: CVE-2016-4300	
3.2.2	Oct. 2016		MTREE: CVE-2016-8688 7Zip: CVE-2016-8689 General: CVE-2016-5418, CVE-2016-8687	

Continued on next page

Table B.1 – continued from previous page

Version	Date	Introduced	Fixed	Comment
3.3.0	Feb. 2017	General: CVE-2018-1000879	CAB: CVE-2016-10349, CVE-2016-10350 LHA: CVE-2017-5601 General: CVE-2016-10209	
3.3.3	April 2019		RAR: CVE-2017-14502 ISO9660: CVE-2017-14501 LHA: CVE-2017-14503 Other: CVE-2017-14166	
3.4.0	June 2019	RAR: CVE-2020-9308	RAR: CVE-2018-1000877, CVE-2018-1000878, CVE-2019-18408 ISO9660: CVE-2019-1000020 7Zip: CVE-2019-1000019 Other: CVE-2018-1000880 General: CVE-2018-1000879	Support for achive format RAR version 5 added in this version
3.4.1	Dec. 2019		General: CVE-2019-19221	
3.4.2	Feb. 2020		RAR: CVE-2020-9308	

Appendix C

Case study vulnerabilities

Table C.1: Libsndfile and DokuWiki vulnerabilities

Libsndfile	DokuWiki
CVE-2007-4974	CVE-2009-1960
CVE-2009-0186	CVE-2010-0287
CVE-2009-1788	CVE-2010-0288
CVE-2009-1791	CVE-2010-0289
CVE-2009-4835	CVE-2012-0283
CVE-2011-2696	CVE-2012-2128
CVE-2014-9496	CVE-2012-2129
CVE-2014-9756	CVE-2014-8761
CVE-2015-7805	CVE-2014-8762
CVE-2017-12562	CVE-2014-8763
CVE-2017-14245	CVE-2014-8764
CVE-2017-14246	CVE-2014-9253
CVE-2017-14634	CVE-2015-2172
CVE-2017-16942	CVE-2016-7964
CVE-2017-6892	CVE-2016-7965
CVE-2017-7585	CVE-2017-12583
CVE-2017-7586	CVE-2017-12979
CVE-2017-7741	CVE-2017-12980
CVE-2017-7742	CVE-2017-18123
CVE-2017-8361	CVE-2018-15474
CVE-2017-8362	
CVE-2017-8363	
CVE-2017-8365	
CVE-2018-13139	
CVE-2018-13419	
CVE-2018-19432	
CVE-2018-19661	
CVE-2018-19662	
CVE-2018-19758	
CVE-2019-3832	