8-5-2016

Identifying the Presence of Known Vulnerabilities in the Versions of a Software Project

Craig Cabrey
cac2573@rit.edu

ROCHESTER INSTITUTE OF TECHNOLOGY

MASTERS THESIS

Identifying the Presence of Known Vulnerabilities in the Versions of a Software Project

Author:

Craig Cabrey

Adviser:

Dr. Meiyappan Nagappan

A thesis submitted in partial fulfillment of the requirements

for the degree of Master of Science in Software Engineering

from the

Department of Software Engineering

B. Thomas Golisano College of Computing and Information Sciences

August 5, 2016

The thesis "Identifying the Presence of Known Vulnerabilities in the Versions of a Software Project" by Craig Cabrey has been examined and approved by the following Examination Committee:

Dr. Meiyappan Nagappan
Thesis Committee Chair
Assistant Professor

Dr. Mehdi Mirakhorli
Assistant Professor

Dr. Scott Hawker
Graduate Program Director
Associate Professor

Acknowledgements

I would like to thank my committee members, Meiyappan Nagappan and Mehdi Mirakhorli. In particular, I would like to thank my adviser, Meiyappan Nagappan, who provided me with the opportunity and necessary guidance to pursue this research.

I would also like to thank the independent security researcher, Bushra Aloraini, whose efforts to produce the evaluation data set was crucial to the success of this project.

I would like to thank the Golisano College of Computing and Information Sciences, which provided funding for this work.

Finally, I would like to thank all my family and friends, without whom I would not be where I am today.

ROCHESTER INSTITUTE OF TECHNOLOGY

Abstract

Department of Software Engineering

Master of Science in Software Engineering

Identifying the Presence of Known Vulnerabilities in the Versions of a Software Project

by Craig Cabrey

As the world continues to embrace a completely digital society in all aspects of life, the ever present threat of a security flaw in a software system looms. Especially with a stream of high profile security flaws and breaches, the public is more aware of the risk now than ever before.

However, the realities of any software project is that there are engineering concerns of the utmost importance that all demand simultaneous attention. To balance and manage these challenges, software engineering has developed patterns of industry activities and best practices. Yet even as engineers rely on these practices to stay afloat, managing security can become elusive in a tangled mess of complex relationships between systems. Modern software projects rely upon other software to do its job; only the most niche and specialized software lives in isolation in today's industry.

In this work, we present an approach to help alleviate one of the aspects of actively managing security in a software project. The objectives of this approach are 1. to establish the presence of a known vulnerability in a software project version and 2. to develop a set of versions of a software project which identify vulnerability status. We tested the approach on three Apache Software Foundation projects, for a total of eleven vulnerabilities tested. In the analysis of

the results, we find that the approach is conservative in marking a particular version $not\ vulnerable$, but when it does so, it is completely consistent with the evaluation results. This conservative nature is a beneficial characteristic of the approach when considering the context of software security in which it operates.

Contents

List of Figures

List of Tables

For my parents, Tom and Denise Cabrey.

Chapter 1

Introduction

1.1 Background

Over time, software projects evolve to be enormously complex, both in terms of structure and interactions with external systems. There are a number of concerns of which engineers must be aware during the system's design phase and its evolution. Issues such as compatibility, deployment, quality control, and security all demand time and resources while a project is being developed or maintained. Modern practices to address these concerns can be traced all the way back to when Belady et. al proposed Lehman's Law of Software Evolution (Belady and Lehman, 1976).

Among those mentioned, software security has become an issue of paramount importance. With the global transition to online systems, the smallest mistake could have far reaching consequences (Anderson, 2001; Telang and Wattal, 2007). Software security, then, is a fundamental quality of any system. To address this, industry standardized practices have been developed surrounding security issues, such as responsible disclosure[1]. Responsible disclosure requires the cooperation of all involved parties, from the software maintainers to the software users. Without this cooperation, the practice falls apart and has the potential to leave software open for exploit.

[1] http://www.cert.org/vulnerability-analysis/vul-disclosure.cfm

Within the software community, a flaw that has security ramifications is known as a vulnerability. A vulnerability is a software defect that exposes an attack vector for a malicious actor to exploit. Not every vulnerability within a system is necessarily exploitable, however it can be difficult to understand when this is the case (Shirey, 2000).

The discovery and disclosure of new vulnerabilities occurs on a regular basis and is an area of research that is fairly well understood. For example, Liu et. al (Liu et al., 2012) performed a survey on techniques of discovering vulnerabilities and outlined a number of techniques: static analysis, penetration testing, fuzzing, Vulnerability Discovery Models, and more. These techniques are already deployed in practice while research on new and existing techniques is continuing.

However, while much research has been done in the way of predicting vulnerabilities (Hovsepyan et al., 2012; Shin and Williams, 2013), there is little research on tracking existing vulnerabilities across the history of a software project. In an industry that is increasingly collaborative and decentralized (especially in the case of open source projects), the possibility of regressions being introduced remains ever present. At least two of Lehman's laws reinforce this possibility: *increasing complexity* and *declining quality*. Furthermore, the ever increasing trend of code reuse across and within projects makes possible the contamination of security problems from external pieces of software.

Utilizing the vast quantity of information on security vulnerabilities and high quality project metadata, known vulnerabilities can be tracked throughout a project's history. Projects that have a history of past vulnerabilities will be able to track the status of these defects moving forward, to the benefit of all involved parties.

The goal of this research is twofold: 1. develop a general algorithmic approach

for identifying the presence (or absence) of a known vulnerability in a software project, and 2. from this algorithm, extrapolate the set of versions of the software project that contain the vulnerability and those that do not.

Section 1.2 outlines the motivation for this work and Section 1.3 details the specific research objectives. In Section 2, the related work is identified. Section 3 gives an in-depth overview of the approach. Section 4 presents the results, evaluation, and analysis. Sections 4.5, 5.1, and 5.2 outline threats to validity, applications, and future work, respectively. Finally, Section 5 recaps with the conclusion.

1.2 Motivation

Since the exposure of high profile security vulnerabilities such as Heartbleed (Durumeric et al., 2014) and POODLE (Möller, Duong, and Kotowicz, 2014), security has become a forefront issue not only in the software engineering industry, but also on the minds of the general public.

As the number of high profile security vulnerabilities continues to grow, the question has been raised as to whether security is being managed as properly as it could be. The proliferation of advanced mobile computing platforms has placed an additional emphasis on this urgency, as secure systems are more critical now than ever before.

At the same time, software engineering techniques have advanced significantly since the introduction of Lehman's Laws. Developers have turned to the vast ecosystem of open source software to provide functionality built by the community that proved too difficult or too cumbersome to achieve on an individual basis. But this approach also has its downsides.

The quality of external code has a significant influence on the quality of the project. For example, Mojica et. al analyzed hundreds of thousands of Android

applications in search of code reuse (Mojica et al., 2014). The authors found that the quality of the apps and libraries that were being reused affected the quality of the consuming app. In the same respect, security has an impact as well: including vulnerable versions of an app or library might infect the user.

In addition to this, dependency management becomes another item on the maintenance checklist, further increasing the mental load of the maintainers (Klatt et al., 2012). For the projects that have the necessary infrastructure to perform these tasks, it is less of an issue. But for projects that don't have the resources or tooling necessary, dependencies may stagnate. In many cases, developers will simply bypass the proper channels and directly include copies of the libraries which they wish to use. This leads to a situation where there are many copies of the same software included as a dependency in such a way that precludes easily updating the software.

Keeping apprised on the latest developments in the security landscape and software engineering practices is a difficult challenge. Millions of projects are engaging in software reuse in an unprecedented manner. The downside is that keeping these projects up to date, especially as vulnerabilities are disclosed, patched, and released, can be a daunting challenge. This research attempts to alleviate part of that burden by introducing a method to automatically track the status of a vulnerability across different versions of a software project.

1.3 Research Objectives

There are two objectives that this work contributes. The first objective is to develop a general algorithmic approach to identifying the presence of a known vulnerability in a software project. The second is to determine the set of versions of the software project that contain the resolution of the vulnerability and the set of versions which do not.

The end result of the research is to present the versions of a given software project that resolve a given vulnerability. This is intended to be used by a software developer to track when a system dependency may be exposing a vulnerability in the system and a path forward to resolve the potential vulnerability.

1.3.1 Vulnerability Identification

The goal of this part of the research is to devise a methodology by which to identify whether the given software version exhibits a given vulnerability. It is important to note that the purpose of this research is *not* to identify additional, unknown vulnerabilities.

1.3.2 Software Version Sets

The result of this part of the research is to utilize the methodology of identifying whether a software version exhibits a particular vulnerability discussed in the previous section and apply it to all versions of a software project.

It is important to note here that just because some version of software contains the resolution for a vulnerability, it does not preclude future versions from having the same vulnerability present again. There are many factors that go into software development that may cause a vulnerability to reappear, such as a significant refactoring. Therefore, it is important to consider all versions of a software project and not simply stop at the version of the software that initially incorporated the defect resolution.

Chapter 2

Related Work

There are two general areas of research that are relevant to this work.

The first is security research. Security is a complex and ever evolving topic which is unique in that it includes the presence of bad actors. In regards to security, the software engineering industry has grown tremendously in its understanding of and ability to proactively seek out the best practices in software security.

The second is software evolution over a time period. While this is a particularly large field, we are interested in observing the activity of code over time in how it is modified, used, and re-used. This is a heavily researched area of software engineering and there are a variety of techniques and practices developed over the past several decades.

2.1 Change Detection

Fluri et. al presented an approach, *change distilling*, that is capable of tracking changes across software versions (Fluri et al., 2007). Rather than relying on text difference algorithms, the approach uses abstract syntax trees and tree-based differencing algorithms. The authors evaluated the algorithm with a benchmark of 1,064 manually classified changes from three open source projects.

Kim and Notkin surveyed matching techniques for revisions across a software project (M. Kim and Notkin, 2006). The evaluated matching techniques were:

1. Entity Name,
2. String,
3. Syntax Tree,
4. Control Flow Graph,
5. Program Dependence Graph,
6. Binary Graph,
7. Clone Detection, and
8. Origin Analysis

The authors found that the optimal matching technique varies on the specific context. However, the best technique involved a combinations of techniques. This hybrid approach consists of running all techniques and finding a consensus from the results.

For the purposes of this research, we will be focusing on the second technique, matching by string. While this technique is not the best in terms of its ability to accurately match across different revisions of a project, it is the best option for the context of our work.

Techniques such as Entity Name, Control Flow Graph, and Program Dependence graph are too coarse in their approach. The core cause of a vulnerability can manifest itself in the smallest of details (for example, an off by one error that causes a buffer overflow). Thus, these approachs are unable to provide the fine grain detail this research requires. The Syntax Tree technique requires specialized tooling to be able to parse the source artifacts and build up the abstract syntax trees for comparison purposes. Since one of the goals of the work is to be a general algorithmic approach, depending on support of specific languages

is not acceptable. Binary matching would require transforming the patch into its binary equivalent, which would be even more inaccurate (differences in binaries of different platforms, lack of reproducible builds, and so on). Finally, the clone detection and origin analysis techniques are promising approachs, but unfortunately neither one would be accept a snippet of code as an input, such as what a patch would provide.

2.2 Vulnerability Discovery

In another work, Kim explored different vulnerability discovery models and presented another that takes advantage of multiple versions within a software project (J. Kim, Malaiya, and Ray, 2007). The proposed model analyzes two successive versions. Kim finds that two successful versions are likely to share vulnerabilities because of the shared codebase.

Spacco et. al utilized a static analysis tool, *FindBugs*, to track defect warnings across different versions of a software project (Spacco, Hovemeyer, and Pugh, 2006). The authors identify two approaches: pairing warnings from different versions and warning signatures. The former identifies the pair based on a hash and equivalence predicate. The latter generates a string that includes class, method, and field names, then compares the value of the MD5 hash of the string.

Zhang et. al performed an empirical study with the National Vulnerability Database in an attempt to predict software vulnerabilities (Zhang, Caragea, and Ou, 2011). Though the prediction model failed to produce satisfactory results, part of the author's approach includes using the difference between two versions of a software project as an input.

2.3 Code Reuse

Hanna et. al (Hanna et al., 2013) built and executed a scalable system that performed code similarity analysis on applications from the Google Play Store called Juxtapp. The researchers evaluated their system on a sample of 58,000 Android applications to demonstrate the system's scalability. They found 463 applications with buggy code reuse, 34 applications with known malware, and applications that are pirated versions of legitimate applications available on the store.

Adams et. al conducted an empirical study regarding the various activities required to make code reuse effective. (Adams et al., 2016). The process of code reuse is not free: it requires specific quality assurance and other software engineering activities which can cost signficant time and effort. The empirical study analyizes three highly successful open source distributions (Debian, Ubuntu, and FreeBSD). One of the more relevant aspects of this study was how these organizations go about including updates from upstream vendors. There was no clear answer to this question: changing anything is an inherent risk, but not including critical updates (such as security patches) is even more of a risk. This already complicated question is even further compounded in organizations that make use of dozens of upstream software components.

2.4 Code Clones

Roy et. al performed an analysis of code clones on more than fifteen open source C and Java systems (Roy and Cordy, 2008). To perform the analysis, they used the clone detection tool NICAD, which is effective for identical and near-identical code clones. The results gathered were then validated manually by visual inspection of the identified clone pairs. The researchers found that

the clone detection tool was effective in identifying duplicated code in several high profile C and Java open source software projects.

Jang et. al (Jang, Agrawal, and Brumley, 2012) presented a tool called ReDeBug for finding unpatched code clones in operating system distributions such as Debian GNU/Linux. ReDeBug is a tool designed to scale to the operating system level and as such focuses on being fast and reducing the false detection rate of determining code clones. To evaluate the tool, the researchers ran it across all packages in Debian Lenny and Squeeze, Ubuntu Maverick and Oneiric, all SourceForge C/C++ projects, and the Linux kernel. It was able to identify 15,546 unpatched vulnerabilities from performing this analysis. The researchers were able to demonstrate the real world capabilities of ReDeBug by confirming 145 bugs in Debian Squeeze packages.

Chapter 3

Approach

In order to determine whether a vulnerability is present in a particular version of software, we must be able to detect if the version exhibits the characteristics of that vulnerability. We refer to this outcome as a $vulnerability\ and\ version$ pair.

The characteristics of a vulnerability are a result of the flaw in the software itself. That is, the logical error(s) that occurred when the software was written or changed which gave birth to the vulnerability. The resolution of a vulnerability is also introduced by a change to the software source.

Therefore, by searching for the correction to the error in the software source, we can deduce whether that vulnerability is exhibited in that version of the software. This correction comes in the form of a software patch, which forms the fundamental element of the approach.

Note that this approach is useful only when developers do not specify the affected versions of a software project. When the versions are called out, it is best to rely upon the developer's knowledge and expertise of the project which they represent.

3.1 Anatomy of a Patch

The approach centers around the use of a software patch. To fully understand why this is useful and appropriate, it is crucial to understand what kind of information a patch contains and how it is encoded.

A software patch is a special document that describes how to make one or more changes to a codebase[1]. The GNU `diff`[2] and `patch`[3] utilities are examples of programs used to create and apply patches, respectively. Source control management systems such as Git use these tools to describe changes between different revisions.

Figure 3.1 illustrates an example of a software patch from the Apache HTTP Server project.

The first section of the patch is known as the *header*. It contains information such as source control metadata, authorship information, time stamps, and the subject matter of the patch.

The *body* of the patch is composed of any number of *diffs*. Each diff describes one or more changes to a single file within the source code tree. These changes are known as *hunks*.

Each hunk contains information that describes how lines within a source file are modified. A line can either be added or removed. An addition is denoted by a line that is prefixed with a plus sign (+) whereas a removal is denoted by a line that is prefixed with a minus sign (−). A hunk has no concept of a modified line; rather a modification will be treated as a removal and an addition.

The information encoded by a patch is rich with information that can be used to detect the resolution of a vulnerability within a software project. Relying on a patch as the foundation of the approach means that the approach remains

[1] http://oss-watch.ac.uk/resources/softwarepatch
[2] https://www.gnu.org/software/diffutils/
[3] http://savannah.gnu.org/projects/patch/

language independent and operates directly on the primary artifacts of the software project.

3.2 Data Conditioning

To ensure we compute the most accurate and consistent results, a number of steps are taken during or prior to execution of the detection process. These steps focus on conditioning the input data to minimize the number of false results.

1. Where possible, a patch that provides a resolution for the vulnerability is extracted from the version control system of the project. A source control specific patch (such as Figure 3.1) has additional metadata that assists with resolving file histories across different versions. For the purposes of this work, `git format-patch -1 SHA` was used to perform this step.

2. Source comments and other white space changes are pruned from the patch. Comments have no effect on the operational aspect of the code and therefore no implications with regards to the resolution of a vulnerability and can be safely removed. Removing comments from the detection process produces a more accurate ratio with respect to the code.

   White space also has no bearing on the operation of the code. Moreover, white space is prone to a high rate of churn within software projects, which further reinforces the view that it should not be considered.

3. Non-language files, identified by file extension by means of GitHub's Linguist mapping[4], and tests, identified by the case insensitive string 'test' or 'tests' in the file path, are ignored during processing and do not contribute to the ratios as explained in Subsection 3.3.2.

[4]https://github.com/github/linguist/blob/master/lib/linguist/languages.yml

4. For each file in the patch, an attempt is made to resolve the file's path between the time of the patch and the target version being tested. This process requires access to a Git repository and relies heavily on the quality of information within the repository.

In many cases, obtaining a patch from a vulnerability can be a difficult process. The quality of information varies on a case by case basis. Fortunately, a previous work focuses on linking a vulnerability to the change that resolves the defect (H.-M. Chen et al., 2016). As long as the procedure in that research works as intended, it can be used in conjunction with this work to build a pipeline starting with a vulnerability identifier and ending in version sets.

3.3 Detection Process

In this section, the procedure that is used to operationalize the approach is described.

Once a vulnerability is identified, a patch from the repository of the project in question is extracted and prepared, it can be used in the detection procedure.

For each file in the patch, the changes are split up into two distinct lists: one for additions and another for deletions. Next, the source of the file from the version being test is read into a separate list.

The collection of source lines is iterated over twice; first as the list of additions is iterated and again as the list of deletions is iterated. Through each iteration, a line of source is compared against a change.

The comparison step consists of stripping all white space from both lines, splitting up each line into two sets of tokens, and checking the sets for equality.

The detected additions and deletions are then filtered down to ensure that nothing is double counted. The filtering prevents lines that are simply moved

and unmodified from being counted as parts of the results. As an example, the following is an indicated change that would be ignored by the process:

```
- printf("");
+ printf("");
```

This procedure is repeated for all versions in a software project identified for testing and collected into a mapping from version identifier to a result set.

3.3.1 One Line Changes

In the case of a patch that includes a one line change to a file, there is an additional component to the procedure. Rather than searching for only the change to the file, it will also search for the context surrounding that change. Specifically, that means one line above and below the change in question, if applicable.

It is necessary to treat this scenario in a special manner because of the edge case it presents. Searching for the presence or absence of one line does not provide enough context to sufficiently determine if the specific change is actually exhibited in the file.

Because the one line has the potential to appear more than once in a file, it would cause an unacceptable rise in the false positive rate. Forcing the change alongside its context has the effect of considering only the area of the file that is relevant.

The next section elaborates on how the final result is calculated and used to determine a version's vulnerability status.

3.3.2 Ratios

The raw values used to make a final determination of the status of a vulnerability and version pair are two ratios: an $additions$ ratio and $deletions$ ratio. These

values are calculated at the end of the procedure, as defined by Equation 3.1 and 3.2.

$$r_a = \begin{cases} \frac{1}{|P_a|} \times \sum_{f \in P} \sum_{a \in P_{f_a}} (a \in S_f), & if\ |P_a| > 0 \\ null, & otherwise \end{cases} \tag{3.1}$$

$$r_d = \begin{cases} \frac{1}{|P_d|} \times \sum_{f \in P} \sum_{d \in P_{f_d}} (d \notin S_f), & if\ |P_d| > 0 \\ null, & otherwise \end{cases} \tag{3.2}$$

Where,

- r_a and r_d are the ratios of additions and deletions, respectively,
- P_a and P_d are the sets of all relevant additions and deletions in the patch, respectively,
- P is the set of relevant files in the patch,
- P_{f_a} is the set of relevant additions in the patch for f,
- P_{f_d} is the set of relevant deletions in the patch for f,
- S_f is the set of source lines in f,
- $null$ indicates that the result will not be considered when determining a final outcome

Both r_a and r_d have values that range from 0.0 to 1.0. An r_a value of 1.0 means that all additions are present in the source tree whereas a value of 0.0 means that none of the additions are present. Similarly, an r_d value of 1.0 means that all deletions are *not* present in the source tree whereas a value of 0.0 means that all are present.

In the case of a patch containing hunks that consist solely of either additions or deletions, only the relevant ratio is considered.

3.3.3 Ratio Thresholds

In order to determine the likelihood of a vulnerability being exhibited in a version of software, the raw results generated by the methodology must be translated into more relevant findings.

Thus, there are three possible outcomes for each vulnerability and software version pairing:

1. *Vulnerable (V)*
2. *Not Vulnerable (NV)*
3. *Indeterminate (I)*

Equation 3.3 describes the mapping from raw results to the possible outcomes:

$$R = \begin{cases} indeterminate, & \neg(\exists f \in P\ exists(f)) \\ vulnerable, & if\ r_a < T_a\ or\ r_d < T_d \\ not\ vulnerable, & otherwise \end{cases} \tag{3.3}$$

Where,

- R is the result or outcome,
- T_a and T_d are the thresholds for additions and deletions, respectively,
- $exists()$ is a function that determines if a file exists at a particular version

To develop a pair of thresholds for this work, the system was trained by manually inspecting the input patch and the source artifacts. The version of software that included the patch is treated as the *control* version. That is, the version that we are most certain includes the defect fix. This assertion is additionally backup up by project metadata such as issue trackers that identify the version in which the vulnerability is resolved. T_a and T_d were set to 0.5 and 0.25, respectively.

FIGURE 3.1: Example of a software patch from the Apache HTTP Server Project, created with `git format-patch -1`

```
From 2fe8ec85fa8ef1340a61e688f3bc43c799add78e Mon Sep 17 00:00:00 2001
From: "William A. Rowe Jr" <wrowe@apache.org>
Date: Tue, 2 Mar 2010 04:46:13 +0000
Subject: [PATCH] SECURITY: CVE-2010-0408 (cve.mitre.org)

mod_proxy_ajp: Respond with HTTP_BAD_REQUEST when the body is not sent after
request headers indicate a request body is incoming; this is not a case of
HTTP_INTERNAL_SERVER_ERROR.

Submitted by: Niku Toivola <niku.toivola sulake.com>

rpluem, jim, wrowe



git-svn-id: https://[...]/httpd/trunk@917875 13f79535-47bb-0310-9956-ffa450edef68
---
 modules/proxy/mod_proxy_ajp.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/modules/proxy/mod_proxy_ajp.c b/modules/proxy/mod_proxy_ajp.c
index 635ba32..0f5674c 100644
--- a/modules/proxy/mod_proxy_ajp.c
+++ b/modules/proxy/mod_proxy_ajp.c
@@ -257,7 +257,7 @@ static int ap_proxy_ajp_request(apr_pool_t *p, request_rec *r,
             ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
                          "proxy: ap_get_brigade failed");
             apr_brigade_destroy(input_brigade);
-            return HTTP_INTERNAL_SERVER_ERROR;
+            return HTTP_BAD_REQUEST;
         }

         /* have something */
--
2.8.0
```

It is important to acknowledge that developing the thresholds in this manner is not optimal and incorporates the bias of the author.

3.4 Data Sources

In this section, we describe the data sources used to generate results for the research. There are two primary data sources that will be utilized either in part or standalone. For the sake of this research, only open source projects were considered. However, with enough access to information for a proprietary project,

the approach would work just as well.

3.4.1 National Vulnerability Database

The National Vulnerability Database (NVDB) is a public resource maintained by the United States government that collects and makes available information on disclosed vulnerabilities. This system provides historical feeds that can be utilized for mining purposes.

The NVDB provides a rich data set that can be used to extract known vulnerabilities for open source projects and useful metadata related to each vulnerability.

3.4.2 Project Repository

For the approach to operate, it is critical to have complete access to the source artifacts of the software project. The most effective way to get these artifacts is by using the project repository itself. Additionally, repositories also typically contain the state of the source at each version, which is necessary for the objectives of the approach.

3.4.3 Project Metadata

Open source projects are much more than the source that makes up the end result. It is a collaboration of effort among individuals that requires constant and clear communication.

To that end, there is a wealth of information stored in a project's infrastructure, such as mailing lists, issue trackers, and version control logs. Using this data provides context with respect to a particular vulnerability; namely how it was resolved and the time line of the defect.

3.5 Implementation

A Python 3.5 application using the approach was developed to consistently and efficiently generate results using the approach defined in Section 3.

The Python library `GitPython`[5] was used to automate the manipulation of repositories. Git tags are most commonly used to denote release points[6]. Thus, tags were the primary mechanism used in evaluating each version of a software project. However, because tags are used for more than just software releases, the set of tags being used was manually verified to ensure it matched an actual release. Before checking out each tag for testing, `git reset –hard` and `git clean –df` was executed in the repository to ensure there was no cross contamination between versions.

The Python library `whatthepatch`[7] was used to parse the contents of a patch. Figure 3.4 is an example of how the patch in Figure 3.1 is parsed and represented. For each line in the patch, there is a tuple composed of three elements: the original line number, the final line number, and the line contents. With this in mind, there are three cases to consider in the parsed representation:

1. `(1, None, '...')`: A line was removed
2. `(None, 1, '...')`: A line was inserted
3. `(1, 2, '...')`: The line remains unchanged

Along with access to the repository, the parsed representation of the patch became the input for the detection process.

[5]https://github.com/gitpython-developers/GitPython, Version 1.0.2
[6]https://git-scm.com/book/en/v2/Git-Basics-Tagging
[7]https://github.com/cscorley/whatthepatch, Version 0.0.4

3.6 File History Resolution

As noted in Subsection 3.2, one of the steps taken before the detection process is the resolution of the paths of each file in the patch with respect to the target version.

There are a number of scenarios that are possible, but not all can be handled in a manner that leads to meaningful results. The ability to track a file through history depends heavily on the project's version control activities and practices.

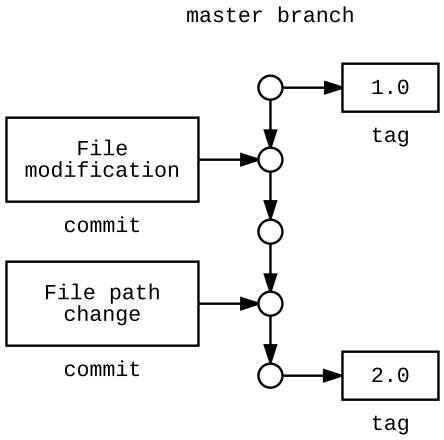
Consider Figure 3.2. The figure represents a relatively simple history of a software project. Each circle represents a change (or commit) to the tree, with forward progression downward. The model being illustrated shows that the project utilizes a single `master` branch off of which each version is tagged. While there may be feature branching happening for development purposes, everything is merged back into `master` before a release is tagged. Thus, for the purposes of tracking, the history is linear and simple to follow. In this scenario, the file's path was changed after the patch was introduced. Catching the change in path is straight forward as it simply a matter of analyzing each change between the introduction of the patch and the tag `2.0`.

Now consider Figure 3.3. In this case, the project is utilizing a more complex model which includes release branches. For each major release (a major release being the `x` in an `x.y.z` versioning scheme), a branch is created off of `master` and used to sequentially tag the minor versions (typically starting at `major.0`). With this model, it is much more difficult to track the history of a file. In this specific scenario, a file path change and subsequent modification occurred after a branch was created for the `1.x` series of releases. While not an extreme situation by itself, the modification to the file was then backported[8] to the `1.x` series by means of a cherry-pick[9]. Because the backporting process likely did

[8]Backporting is the process of retroactively applying newer changes to legacy versions of software.

[9]https://git-scm.com/docs/git-cherry-pick

FIGURE 3.2: A simple Git history involving a single master branch



not include the change in the file path, it is necessary to backtrack to the *ancestor* of both branches before moving forward to the `1.1` tag. Within our implementation, this is achieved by using `git rev-list –ancestry-path`.

These are only two scenarios of many. In some cases, it is impossible to find out what has happened to a file between two points in history. For example, at some point in between, a developer may have accidentally removed a file. Realizing their mistake, they would have most likely added it back in at a later point. By the time the tracking mechanism has reached the point of deletion, however, the commit trail ends and the file would be marked deleted. It is important to note that the history tracking method cannot make up for these issues and will thus lead to $indetermine$ outcomes as a fallback mechanism.

FIGURE 3.3: A Git history involving master and release branches

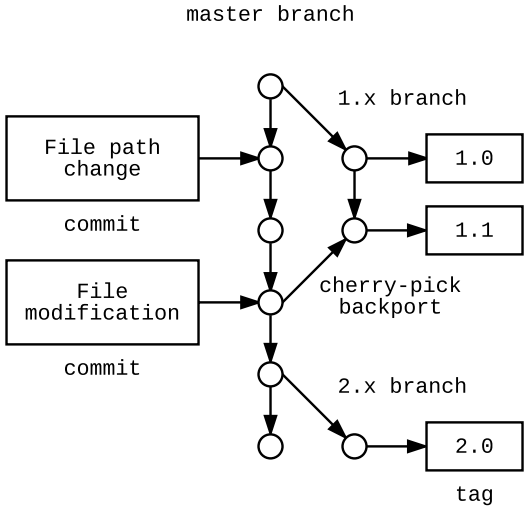


FIGURE 3.4: Example of a parsed patch

```
(257, 257, '            ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,'),
(258, 258, '                         "proxy: ap_get_brigade failed");'),
(259, 259, '            apr_brigade_destroy(input_brigade);'),
(260, None, '            return HTTP_INTERNAL_SERVER_ERROR;'),
(None, 260, '            return HTTP_BAD_REQUEST;'),
(261, 261, '        }'),
(262, 262, ''),
(263, 263, '        /* have something */')
```

Chapter 4

Results

In this section, the results of the approach are presented. The approach was executed across three separate Apache Software Foundation[1] projects. The major versions of each project were tested against a number of vulnerabilities found from either the NVDB, the project's issue tracker, or by mining the project's revision history. Because the evaluation described in Subsection 4.2 is an intensive, time consuming process, we were forced to limit the number of versions and vulnerabilities for each project.

Tables 4.1, 4.2, and 4.3 present the combined results from the approach and the evaluation on the Apache HTTP Server Project[2], Apache Hadoop[3], and Apache CloudStack[4], respectively. These projects were chosen for their rich project metadata, their relative importance in the software community, and the set of historical vulnerabilities associated with each project.

Table 4.4 shows results for the versions that the project developers have deemed to be $not\ vulnerable$. These versions were obtained from either the vulnerability description page, the project's respective issue tracking the defect, or the project change log. In each instance, the version that contains the fix is explicitly identified by the project. For each of these versions, we also expect that the

[1] https://www.apache.org/
[2] https://httpd.apache.org/
[3] https://hadoop.apache.org/
[4] https://cloudstack.apache.org/

TABLE 4.1: Combined result set and evaluation set for the Apache HTTP Server Project

	CVE-2004-0174[a]		**CVE-2006-3747**[b]		**CVE-2007-1862**[c]	
Version	Result	Evaluation	Result	Evaluation	Result	Evaluation
1.2.0	*I*	*I*	*I*	*NV*	*I*	*NV*
1.3.0	*I*	*I*	*I*	*NV*	*I*	*NV*
2.0.1	*V*	*V*	*V*	*NV*	*I*	*NV*
2.1.1	*NV*	*NV*	*V*	*V*	*I*	*NV*
2.2.0	*NV*	*NV*	*V*	*V*	*V*	*NV*
2.3.0	*NV*	*NV*	*NV*	*NV*	*I*	*NV*
2.4.0	*NV*	*NV*	*NV*	*NV*	*I*	*NV*

[a]Resolved in versions 1.3.30 and 2.0.49
[b]Resolved in version 2.2.3
[c]Resolved in version 2.2.6

	CVE-2014-3583[a]		**CVE-2014-8109**[a]		**CVE-2015-3183**[b]	
Version	Result	Evaluation	Result	Evaluation	Result	Evaluation
1.2.0	*I*	*NV*	*I*	*NV*	*I*	*NV*
1.3.0	*I*	*NV*	*I*	*NV*	*I*	*NV*
2.0.1	*I*	*NV*	*I*	*NV*	*V*	*NV*
2.1.1	*I*	*NV*	*I*	*NV*	*V*	*NV*
2.2.0	*I*	*NV*	*I*	*NV*	*V*	*I*
2.3.0	*V*	*NV*	*I*	*NV*	*V*	*I*
2.4.0	*V*	*V*	*V*	*NV*	*V*	*I*

[a]Resolved in version 2.4.12
[b]Resolved in version 2.4.16

approach will yield $not\ vulnerable$ outcomes. With these results, we establish baseline validity of the approach by showing that the outcomes of the approach agree with what the project maintainers state about each vulnerability in relation to the version in which it was fixed.

The results of the approach will herein be referred to as the *result set*. The results of the evaluation will be referred to as the *evaluation set*.

4.1 Understanding the Results

To make sense of the results, we must distinguish between the implications of each outcome for each vulnerability and version pair.

TABLE 4.2: Combined result set and evaluation set for Apache Hadoop

	YARN-1993[a]		**HADOOP-12964**[a]		**CVE-2012-3376**[b]	
Version	Results	Evaluation	Results	Evaluation	Results	Evaluation
0.1.0	*I*	*NV*	*I*	*NV*	*I*	*NV*
0.10.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.11.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.12.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.13.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.14.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.15.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.16.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.17.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.18.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.19.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.2.0	*I*	*NV*	*I*	*NV*	*I*	*NV*
0.20.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.21.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.22.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.23.0	*I*	*V*	*I*	*V*	*I*	*NV*
0.3.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.4.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.5.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.6.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.7.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.8.0	*I*	*NV*	*I*	*V*	*I*	*NV*
0.9.0	*I*	*NV*	*I*	*V*	*I*	*NV*
1.0.0	*I*	*NV*	*I*	*V*	*I*	*NV*
1.1.0	*I*	*NV*	*I*	*V*	*I*	*NV*
1.2.0	*I*	*NV*	*I*	*V*	*I*	*NV*
2.2.0	*V*	*V*	*I*	*V*	*NV*	*NV*
2.3.0	*V*	*V*	*V*	*V*	*NV*	*NV*
2.4.0	*V*	*V*	*V*	*V*	*NV*	*NV*
2.5.0	*V*	*V*	*V*	*V*	*NV*	*NV*
2.6.0	*V*	*V*	*V*	*V*	*NV*	*NV*
2.7.0	*V*	*V*	*V*	*V*	*NV*	*NV*

[a] Resolved in version 2.8.0; not released at the time of testing
[b] Resolved in version 2.2.0

TABLE 4.3: Combined result set and evaluation set for Apache CloudStack

	CVE-2015-3251[a]		**CVE-2015-3252**[a]	
Version	Result	Evaluation	Result	Evaluation
2.1.3	*I*	*I*	*I*	*I*
2.2.0	*I*	*I*	*I*	*I*
3.0.1	*I*	*I*	*I*	*I*
4.0.2	*I*	*I*	*V*	*I*
4.1.0	*V*	*I*	*V*	*I*
4.2.0	*V*	*I*	*V*	*I*
4.3.0	*V*	*V*	*V*	*V*
4.4.0	*V*	*V*	*V*	*V*
4.5.1	*V*	*V*	*V*	*V*
4.6.0	*NV*	*NV*	*NV*	*NV*
4.7.0	*NV*	*NV*	*NV*	*NV*
4.8.0	*NV*	*NV*	*NV*	*NV*

[a]Resolved in version 4.5.2

TABLE 4.4: Results that establish baseline validity by showing *not vulnerable* for resolved versions

Vulnerability	**Version Fixed**	**Result**
Apache HTTP Server		
CVE-2004-0174	1.3.30, 2.0.49	*NV*
CVE-2006-3747	2.2.3	*NV*
CVE-2007-1862	2.2.6	*NV*
CVE-2014-3583	2.4.12	*NV*
CVE-2014-8109	2.4.12	*NV*
CVE-2015-3183	2.4.16	*NV*
Apache Hadoop		
YARN-1993	2.8.0[a]	*N/A*
HADOOP-12964	2.8.0[a]	*N/A*
CVE-2012-3376	2.2.0	*NV*
Apache CloudStack		
CVE-2015-3251	4.5.2	*NV*
CVE-2015-3252	4.5.2	*NV*

[a]Not available at time of testing

A $vulnerable$ outcome implies that the version of software exhibits the characteristics of the vulnerability being tested. The software itself may or may not be exploitable, depending on the circumstance.

A $not\ vulnerable$ outcome implies that the version of software does not exhibit the characteristics of the vulnerability being tested.

An $indeterminate$ outcome means that there was insufficient data available to make a confident determination. In the result set, a vulnerability and version pair is marked as $indeterminate$ when none of the files that comprise the body of the patch are present in the source code tree of that version. In the evaluation set, a vulnerability and version pair is marked $indeterminate$ at the discretion of the researcher along with a rationale.

4.2 Evaluation

In order to effectively establish the validity of the approach, a separate and unbiased analysis of the same data set was performed by an independent security researcher.

The security researcher was instructed to perform a security review of all vulnerabilities across all listed versions of each project. The researcher was permitted to use any methodology (static analysis, runtime exploitation, manual investigation, etc.) of their choosing to determinate vulnerability status. Additional, they were instructed to use the same result types ($vulnerable$, $not\ vulnerable$, and $indeterminate$). However, the formal definition of $indeterminate$ was not established at the time of the evaluation.

The author of this work had no access to the results of the analysis or the independent researcher before results from the approach had been generated.

Tables 4.1, 4.2, and 4.3 present the results from the independent evaluation alongside the results from the approach.

Furthermore, it was important to establish that the approach indicated $not\ vulnerable$ for the versions that the developer indicated as such. The result tables note the versions for which each vulnerability was considered resolved.

4.3 Example

CVE-2004-0174 is an identifier for a security vulnerability within the Apache HTTP Server. The essence of the vulnerability is that the server's connection handler, `server/listen.c`, does not handle short-lived connections in an appropriate manner which may lead to a race condition and denial of service attacks. This issue was resolved in version 1.3.30 of the 1.3.x series and version 2.0.49 of the 2.0.x series.

Table 4.1 shows that for version 2.0.1 of the Apache HTTP Server, both the result set and evaluation set indicate that this version was subject to the vulnerability. However, versions 1.2.0 and 1.3.0 were both marked as $indeterminate$ by the result set and evaluation set. The security researcher that produced the evaluation set gave the following rationale: "Connection handler is in http_main.c." The result set indicates $indeterminate$ because the relevant source files were not present at these points in the project's history. From version 2.1.1 and forward, both the result set and evaluation set indicate that the software is no longer subject to the vulnerability.

CVE-2015-3251 is an identifier for a security vulnerability within Apache CloudStack. The issue allows remotely authenticated administrators to obtain password information for the root accounts of virtual machines under a CloudStack deployment's control. This issue was resolved in version 4.5.2.

Table 4.3 shows that versions 4.1.0 and 4.2.0 were indicated to be $vulnerable$ by the result set. However, the evaluation set indicates that these versions were considered to be $indeterminate$. The rationale provided for this conclusion was that these versions have a different code structure as compared with the vulnerable versions of the software (versions 4.3.0, 4.4.0, and 4.5.1).

4.4 Discussion

The nature of the three types of outcomes means we cannot easily use binary classification techniques to assess the performance of the approach and result set.

Instead, the performance of the results set can be assessed by analyzing its level of agreement with the evaluation set. Tables 4.6, 4.7, and 4.8 contain breakdowns of each outcome type from the result set as the ratios of their counterparts in the evaluation set. For example, for all of the $vulnerable$ CloudStack results in Table 4.6, 54.54% of them were also found to be $vulnerable$ in the evaluation set, 0% of them were found to be $not\ vulnerable$, and 45.45% of them were found to be $indeterminate$.

Interpreting the performance in this manner reveals that the approach as measured by the evaluation set is extremely conservative in marking a vulnerability and version pair as $not\ vulnerable$. For $vulnerable$ outcomes in the result set, only Hadoop had equivalent outcomes in the evaluation set. However, for all $not\ vulnerable$ outcomes in the result set, the evaluation set also had equivalent $not\ vulnerable$ outcomes.

When $indeterminate$ vulnerability and version pairs are broken down into ratios of their corresponding pairs, we see (with the exception of CloudStack) that the majority of outcomes in the evaluation set are $not\ vulnerable$. While the number of $indetermine$ outcomes may be alarming at first, the comparison

TABLE 4.5: Performance without $indeterminate$ outcomes

Accuracy	**Precision**	**Recall**	**F-measure**
$86.\overline{6}\%$	$77.\overline{7}\%$	100%	87.5%

TABLE 4.6: Breakdown of $vulnerable$ vulnerability and version pair outcomes in the result set for each project as ratios of their respective counterparts in the evaluation set

		Result Set		
		HTTP	Hadoop	CloudStack
Evaluation Set	Vulnerable	30.76%	100%	54.54%
	Not Vulnerable	46.15%	0%	0%
	Indeterminate	23.07%	0%	45.45%

to the evaluation shows that we can lend further credibility to the approach and its conservative nature.

Recall that the only context the approach has is a single patch containing the changes that introduce a fix for the vulnerability. The researcher, meanwhile, is able to use whatever resources they deem necessary to determine the vulnerability status. In many of the cases, even the researcher was not able to confidently mark an outcome, as the CloudStack evaluation set shows. When considering the context and the nature of the results, it is appropriate and expected that the $indeterminate$ outcome appears as frequently as it has in the result set.

If we consider the results without $indeterminate$ outcomes, as defined by:

$$\{\, x \neq indeterminate \mid x \in (Results \cap Evaluation) \,\}$$

then a binary classification of the results can be performed. In this context, a $not\ vulnerable$ result would be considered negative and a $vulnerable$ result would be considered positive. The performance of the approach after filtering the results on this criteria is shown in Table 4.5. The precision is calculated to be 77.7% and the recall is calculated to be 100% for an F-measure of 87.5%.

The approach favors erring on the side of caution by indicating a pair is either $vulnerable$ or $indeterminate$. Put another way, when the approach must bias

TABLE 4.7: Breakdown of *not vulnerable* vulnerability and version pair outcomes in the result set for each project as ratios of their respective counterparts in the evaluation set

		Result Set		
		HTTP	Hadoop	CloudStack
Evaluation Set	Vulnerable	0%	0%	0%
	Not Vulnerable	100%	100%	100%
	Indeterminate	0%	0%	0%

TABLE 4.8: Breakdown of *indeterminate* vulnerability and version pair outcomes in the result set for each project as ratios of their respective counterparts in the evaluation set

		Result Set		
		HTTP	Hadoop	CloudStack
Evaluation Set	Vulnerable	0%	32.91%	0%
	Not Vulnerable	91.30%	67.08%	0%
	Indeterminate	8.69%	0%	100%

itself towards either false negatives or false positives, favoring false positives is decidedly superior. When framed in the context of security, this is fitting and advantageous. Security is a critical and particularly detail oriented activity. Misplaced faith in tooling or the smallest, overlooked details have the potential to lead to disastrous results.

4.5 Threats to Validity

There are several threats that pose a risk to the approach and corresponding implementation.

The author used their own experiences and expertise to define the thresholds that map raw values to an outcome. Therefore, personal bias is inherent in these thresholds and results. Including additional experts as well as expanding the quantity and diversity of projects tested would correct this bias. The 5.2 section touches on this by suggesting the use of machine learning as a mechanism to resolve this issue.

As Bird et. al note, version control metadata can provide a rich source of information but the quality varies widely from project to project (Bird et al., 2009). The approach relies heavily on high quality version control metadata being available. Specifically, insufficient or poor file history may cause invalid or indeterminate results. Refactoring or restructuring efforts within a project could lead to similar outcomes. Further efforts to more intelligently handle version control efforts may mitigate this risk.

Finally, we are relying upon the security researcher's expertise to provide high quality, reliable data against which to test the result set for validity. While this is not an unusual approach in and of itself, there is a problem with calling a version of software $not\ vulnerable$. Identifying a version as $vulnerable$ (or even $indeterminate$) is acceptable because it rests upon a provable premise. $Not\ vulnerable$, however, is not as concrete. It is the difference between proving that something exists and proving that something does not exist. The former is concrete in nature while the latter is not. To account for this threat, we have included the rationale for each outcome in the evaluation set. These rationales can be found in Appendix A of this work.

Chapter 5

Conclusions

5.1 Applications

The primary contribution of this work is to identify sets of versions of software that exhibit a particular vulnerability. To that end, an end to end system could be integrated as part of a development work flow that utilizes this approach.

Such a system would have two uses: 1. warn developers of changes in areas that have been affected by past vulnerabilities and the likelihood that they may be re-introducing the vulnerability and 2. notify developers of potentially vulnerable dependencies within their software. A notification system would rely upon the ability to accurately identify the versions of dependencies included within a project.

The second noted use would need to be combined with work related to the identification of dependencies within software systems.

5.2 Future Work

In a future work, the approach can be expanded to address some of the issues identified in Section 4.5. The implementation can be improved to better handle

tracking files across history and through code changes to more accurately reflect the state of a project with respect to a vulnerability. Additionally, machine learning techniques as well as a larger and more diverse set of projects can be applied to refine the approach for accuracy and applicability.

The manual parts of the approach can be made automatic to better integrate the system into a development work flow. A more sophisticated tool can be devised that uses this research as a foundation in order to fully integrate the approach into a development work flow. In combination with other tools, such a system will be able to identify changes that affect previously vulnerable areas and alert developers of potentially vulnerable dependencies.

5.3 Conclusion

The objectives of this research were twofold: 1. to propose a general algorithmic approach to establishing the presence of a known vulnerability in a software project and 2. using this approach, generate a set of versions of a software project indicating which exhibit a specific vulnerability. In devising an approach, a software patch that corrects a defect was established as the ground truth.

Three Apache Foundation projects were chosen to establish the approach as an effective mechanism. A set of vulnerabilities were chosen from each project's respective history. A total of eleven vulnerabilities were tested across all projects. An independent security researcher performed a separate analysis on the same set of projects and vulnerabilities in order to evaluate the results of the approach.

Comparing the results of the approach to the results of the independent evaluation shows that the approach has merit. An analysis of the comparison reveals the conservative nature of the approach: that the approach favors marking a

software version as $vulnerable$ over $non\ vulnerable$. We argue such a conservative approach is actually beneficial in this area of research. It is more appropriate to be cautious in deciding on the lack of a vulnerability as opposed to the alternative.

Appendix A

Evaluation Rationales

TABLE A.1: Evaluation set outcome rationales for the Apache HTTP Server Project

Version	CVE-2004-0174
1.2.0	Connection handler is in `http_main.c`
1.3.0	Connection handler is in `http_main.c`
2.0.1	Does not handle multiple connections in `listen.c`
2.1.1	Patched
2.2.0	Patched
2.3.0	Patched
2.4.0	Patched

Version	CVE-2006-3747
1.2.0	Problematic condition statement does not exist
1.3.0	Problematic condition statement does not exist
2.0.1	Problematic condition statement does not exist
2.1.1	Condition statement has an off-by-one error
2.2.0	Condition statement has an off-by-one error
2.3.0	Condition statement is implemented correctly
2.4.0	Condition statement is implemented correctly

Version	CVE-2007-1862
1.2.0	`mod_mem_cache` is not available
1.3.0	`mod_mem_cache` is not available
2.0.1	`mod_mem_cache` is not available
2.1.1	Does not use `apr_table_copy` to store/recall headers
2.2.0	Does not use `apr_table_copy` to store/recall headers
2.3.0	Code structure changed, different mechanism is used
2.4.0	Code structure changed, different mechanism is used

Version	CVE-2014-3583
1.2.0	`mod_proxy_fcgi` is not available
1.3.0	`mod_proxy_fcgi` is not available
2.0.1	`mod_proxy_fcgi` is not available
2.1.1	`mod_proxy_fcgi` is not available
2.2.0	`mod_proxy_fcgi` is not available
2.3.0	Buffers passed to `handle_headers()` are ensured to be properly `NULL` terminated
2.4.0	Cause of the vulnerability is present in this version

Version	CVE-2014-8109
1.2.0	`mod_lua` is not available
1.3.0	`mod_lua` is not available
2.0.1	`mod_lua` is not available
2.1.1	`mod_lua` is not available
2.2.0	`mod_lua` is not available
2.3.0	`mod_lua` is not available
2.4.0	`LuaAuthzProvider` prohibits users from supplying their own scripts to perform authorization

Version	CVE-2015-3183
1.2.0	`http_filters` is not available
1.3.0	`http_filters` is not available
2.0.1	`http_filters` is not available
2.1.1	`http_filters` is not available
2.2.0	Different code structure
2.3.0	Different code structure
2.4.0	Different code structure

TABLE A.2: Evaluation set outcome rationales for Apache Hadoop

Version	**YARN-1993**
0.1.0	YARN is not implemented
0.10.0	YARN is not implemented
0.11.0	YARN is not implemented
0.12.0	YARN is not implemented
0.13.0	YARN is not implemented
0.14.0	YARN is not implemented
0.15.0	YARN is not implemented
0.16.0	YARN is not implemented
0.17.0	YARN is not implemented
0.18.0	YARN is not implemented
0.19.0	YARN is not implemented
0.2.0	YARN is not implemented
0.20.0	YARN is not implemented
0.21.0	YARN is not implemented
0.22.0	YARN is not implemented
0.23.0	In `TextView.java`, `echo()` does not properly sanitize HTML attribute name
0.3.0	YARN is not implemented
0.4.0	YARN is not implemented
0.5.0	YARN is not implemented
0.6.0	YARN is not implemented
0.7.0	YARN is not implemented
0.8.0	YARN is not implemented
0.9.0	YARN is not implemented
1.0.0	YARN is not implemented
1.1.0	YARN is not implemented
1.2.0	YARN is not implemented
2.2.0	In `TextView.java`, `echo()` does not properly sanitize HTML attribute name
2.3.0	In `TextView.java`, `echo()` does not properly sanitize HTML attribute name
2.4.0	In `TextView.java`, `echo()` does not properly sanitize HTML attribute name
2.5.0	In `TextView.java`, `echo()` does not properly sanitize HTML attribute name
2.6.0	In `TextView.java`, `echo()` does not properly sanitize HTML attribute name
2.7.0	In `TextView.java`, `echo()` does not properly sanitize HTML attribute name

Version	HADOOP-12964
0.1.0	HTTP was not used extensively
0.10.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.11.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.12.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.13.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.14.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.15.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.16.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.17.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.18.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.19.0	`HttpServer.java` does not set an `X-Frame-Options` header
0.2.0	HTTP was not used extensively
0.20.0	`HttpServer.java` does not set an `X-Frame-Options` header
0.21.0	`HttpServer.java` does not set an `X-Frame-Options` header
0.22.0	`HttpServer.java` does not set an `X-Frame-Options` header
0.23.0	`HttpServer.java` does not set an `X-Frame-Options` header
0.3.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.4.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.5.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.6.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.7.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.8.0	`StatusHttpServer` does not set an `X-Frame-Options` header
0.9.0	`StatusHttpServer` does not set an `X-Frame-Options` header
1.0.0	`HttpServer.java` does not set an `X-Frame-Options` header
1.1.0	`HttpServer.java` does not set an `X-Frame-Options` header
1.2.0	`HttpServer.java` does not set an `X-Frame-Options` header
2.2.0	`HttpServer.java` does not set an `X-Frame-Options` header
2.3.0	`HttpServer2.java` does not set an `X-Frame-Options` header
2.4.0	`HttpServer2.java` does not set an `X-Frame-Options` header
2.5.0	`HttpServer2.java` does not set an `X-Frame-Options` header
2.6.0	`HttpServer2.java` does not set an `X-Frame-Options` header
2.7.0	`HttpServer2.java` does not set an `X-Frame-Options` header

Version	**CVE-2012-3376**
0.1.0	Does not contain federation features
0.10.0	Does not contain federation features
0.11.0	Does not contain federation features
0.12.0	Does not contain federation features
0.13.0	Does not contain federation features
0.14.0	Does not contain federation features
0.15.0	Does not contain federation features
0.16.0	Does not contain federation features
0.17.0	Does not contain federation features
0.18.0	Does not contain federation features
0.19.0	Does not contain federation features
0.2.0	Does not contain federation features
0.20.0	Does not contain federation features
0.21.0	Does not contain federation features
0.22.0	Does not contain federation features
0.23.0	Does not contain federation features
0.3.0	Does not contain federation features
0.4.0	Does not contain federation features
0.5.0	Does not contain federation features
0.6.0	Does not contain federation features
0.7.0	Does not contain federation features
0.8.0	Does not contain federation features
0.9.0	Does not contain federation features
1.0.0	Does not contain federation features
1.1.0	Does not contain federation features
1.2.0	Does not contain federation features
2.2.0	Patched
2.3.0	Patched
2.4.0	Patched
2.5.0	Patched
2.6.0	Patched
2.7.0	Patched

TABLE A.3: Evaluation set outcome rationales for Apache CloudStack

Version	**CVE-2015-3251**
2.1.3	Different code structure
2.2.0	Different code structure
3.0.1	Different code structure
4.0.2	Different code structure
4.1.0	Different code structure
4.2.0	Different code structure
4.3.0	In `LibvirtComputingResource.java`, when rebooting virtual machines, sensitive information is exposed since certain properties do not have the `VIR_DOMAIN_XML_SECURE` flag.
4.4.0	In `LibvirtComputingResource.java`, when rebooting virtual machines, sensitive information is exposed since certain properties do not have the `VIR_DOMAIN_XML_SECURE` flag.
4.5.1	In `LibvirtComputingResource.java`, when rebooting virtual machines, sensitive information is exposed since certain properties do not have the `VIR_DOMAIN_XML_SECURE` flag.
4.6.0	Patched
4.7.0	Patched
4.8.0	Patched

Version	**CVE-2015-3252**
2.1.3	Different code structure
2.2.0	Different code structure
3.0.1	Different code structure
4.0.2	Different code structure
4.1.0	Different code structure
4.2.0	Different code structure
4.3.0	In `LibvirtComputingResource.java`, when rebooting virtual machines, sensitive information is exposed since certain properties do not have the `VIR_DOMAIN_XML_MIGRATABLE` flag.
4.4.0	In `LibvirtComputingResource.java`, when rebooting virtual machines, sensitive information is exposed since certain properties do not have the `VIR_DOMAIN_XML_MIGRATABLE` flag.
4.5.1	In `LibvirtComputingResource.java`, when rebooting virtual machines, sensitive information is exposed since certain properties do not have the `VIR_DOMAIN_XML_MIGRATABLE` flag.
4.6.0	Patched
4.7.0	Patched
4.8.0	Patched

Bibliography

Adams, Bram et al. (2016). "An empirical study of integration activities in distributions of open source software". In: *Empirical Software Engineering* 21.3, pp. 960–1001.

Anderson, Ross (2001). "Why information security is hard-an economic perspective". In: *Computer security applications conference, 2001. acsac 2001. proceedings 17th annual*. IEEE, pp. 358–365.

[Apache-SVN] Contents of /httpd/httpd/branches/2.0.x/CHANGES (2016). URL: `http://svn.apache.org/viewvc/httpd/httpd/branches/2.0.x/CHANGES?view=markup` (visited on 08/08/2016).

[Apache-SVN] Contents of /httpd/httpd/branches/2.2.x/CHANGES (2016). URL: `http://svn.apache.org/viewvc/httpd/httpd/branches/2.2.x/CHANGES?view=markup` (visited on 08/08/2016).

[Apache-SVN] Contents of /httpd/httpd/branches/2.4.x/CHANGES (2016). URL: `http://svn.apache.org/viewvc/httpd/httpd/branches/2.4.x/CHANGES?view=markup` (visited on 08/08/2016).

Belady, Laszlo A. and Meir M Lehman (1976). "A model of large program development". In: *IBM Systems journal* 15.3, pp. 225–252.

Bird, Christian et al. (2009). "The promises and perils of mining git". In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, pp. 1–10.

Chen, Haibo (2016). *[HADOOP-12964] Http server vulnerable to clickjacking*. URL: `https://issues.apache.org/jira/browse/HADOOP-12964` (visited on 05/10/2016).

Chen, Hong-Mei et al. (2016). "Predicting and Fixing Vulnerabilities Before They Occur: A Big Data Approach". In: *Proceedings of the 2Nd International Workshop on BIG Data Software Engineering*. BIGDSE '16. Austin, Texas: ACM, pp. 72–75. ISBN: 978-1-4503-4152-3. DOI: 10.1145/2884781.2896829. URL: http://doi.acm.org/10.1145/2884781.2896829.

CVE-2004-0174. (2004). Available from MITRE, CVE-ID CVE-2004-0174. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-0174 (visited on 07/15/2016).

CVE-2006-3747. (2006). Available from MITRE, CVE-ID CVE-2006-3747. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3747 (visited on 07/15/2016).

CVE-2007-1862. (2007). Available from MITRE, CVE-ID CVE-2007-1862. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1862 (visited on 07/15/2016).

CVE-2012-3376. (2012). Available from MITRE, CVE-ID CVE-2012-3376. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3376 (visited on 07/15/2016).

CVE-2014-3583. (2014). Available from MITRE, CVE-ID CVE-2014-3583. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3583 (visited on 07/15/2016).

CVE-2014-8109. (2014). Available from MITRE, CVE-ID CVE-2014-8109. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8109 (visited on 07/15/2016).

CVE-2015-3183. (2015). Available from MITRE, CVE-ID CVE-2015-3183. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3183 (visited on 07/15/2016).

CVE-2015-3251. (2015). Available from MITRE, CVE-ID CVE-2015-3251. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3251 (visited on 07/15/2016).

CVE-2015-3252. (2015). Available from MITRE, CVE-ID CVE-2015-3252. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3252 (visited on 07/15/2016).

Durumeric, Zakir et al. (2014). "The matter of heartbleed". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, pp. 475–488.

Fluri, Beat et al. (2007). "Change distilling: Tree differencing for fine-grained source code change extraction". In: *IEEE Transactions on Software Engineering* 33.11, pp. 725–743.

Hanna, Steve et al. (2013). "Juxtapp: A scalable system for detecting code reuse among android applications". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, pp. 62–81.

Hovsepyan, Aram et al. (2012). "Software vulnerability prediction using text analysis techniques". In: *Proceedings of the 4th international workshop on Security measurements and metrics*. ACM, pp. 7–10.

Jang, Jiyong, Ankit Agrawal, and David Brumley (2012). "ReDeBug: finding unpatched code clones in entire os distributions". In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, pp. 48–62.

Kim, Jinyoo, Yashwant K Malaiya, and Indrakshi Ray (2007). "Vulnerability discovery in multi-version software systems". In: *High Assurance Systems Engineering Symposium, 2007. HASE'07. 10th IEEE*. IEEE, pp. 141–148.

Kim, Miryung and David Notkin (2006). "Program element matching for multi-version program analyses". In: *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, pp. 58–64.

Kinsella, John (2016a). *CVE-2015-3251: Apache CloudStack VM Credential Exposure*. URL: http://mail-archives.apache.org/mod_mbox/cloudstack-users/201602.mbox/%3C94DD4CB4-F718-4F79-A934-3D677E497114@gmail.com%3E (visited on 08/08/2016).

– (2016b). *CVE-2015-3252: Apache CloudStack VNC authentication issue*. URL: http://mail-archives.apache.org/mod_mbox/cloudstack-users/

201602.mbox/%3C7508580E-3D83-49FD-BE6E-B329B0503130@gmail.com%3E (visited on 08/08/2016).

Klatt, Benjamin et al. (2012). "Identify impacts of evolving third party components on long-living software systems". In: *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, pp. 461–464.

Liu, Bingchang et al. (2012). "Software vulnerability discovery techniques: A survey". In: *2012 Fourth International Conference on Multimedia Information Networking and Security*. IEEE, pp. 152–156.

Mojica, Israel J et al. (2014). "A large-scale empirical study on software reuse in mobile apps". In: *IEEE software* 31.2, pp. 78–86.

Möller, Bodo, Thai Duong, and Krzysztof Kotowicz (2014). "This POODLE bites: exploiting the SSL 3.0 fallback". In: *PDF online*.

Myers, Aaron T. (2012). *[CVE-2012-3376] Apache Hadoop HDFS information disclosure vulnerability*. URL: http://www.securityfocus.com/archive/1/523468 (visited on 08/08/2016).

Roy, Chanchal K and James R Cordy (2008). "An empirical study of function clones in open source software". In: *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*. IEEE, pp. 81–90.

Shin, Yonghee and Laurie Williams (2013). "Can traditional fault prediction models be used for vulnerability prediction?" In: *Empirical Software Engineering* 18.1, pp. 25–59.

Shirey, R. (2000). *Internet Security Glossary*. RFC 2828. GTE / BBN Technologies, pp. 189–189. URL: https://tools.ietf.org/html/rfc2828.

Spacco, Jaime, David Hovemeyer, and William Pugh (2006). "Tracking Defect Warnings Across Versions". In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. MSR '06. Shanghai, China: ACM, pp. 133–136. ISBN: 1-59593-397-2. DOI: 10.1145/1137983.1138014. URL: http://doi.acm.org/10.1145/1137983.1138014.

Telang, Rahul and Sunil Wattal (2007). "An empirical analysis of the impact of software vulnerability announcements on firm stock price". In: *IEEE Transactions on Software Engineering* 33.8, pp. 544–557.

Yu, Ted (2014). *[YARN-1993] Cross-site scripting vulnerability in TextView.java*. URL: `https://issues.apache.org/jira/browse/YARN-1993` (visited on 05/10/2016).

Zhang, Su, Doina Caragea, and Xinming Ou (2011). "An empirical study on using the national vulnerability database to predict software vulnerabilities". In: *International Conference on Database and Expert Systems Applications*. Springer, pp. 217–231.