Gottfried Wilhelm
Leibniz Universität Hannover
Faculty of Electrical Engineering and Computer Science
Institute of Practical Computer Science
Software Engineering Group

Enhancement of a Vulnerability Checker for Software Libraries with Similarity Metrics based on File-Hashes

Bachelor Thesis

in Computer Science

by

Huu Kim Nguyen

First Examiner: Prof. Dr. Kurt Schneider
Second Examiner: Dr. Jil Klünder
Supervisor: M.Sc. Fabien Patrick Viertel

Hannover, March 19, 2020

Declaration of Independence

I hereby certify that I have written the present bachelor thesis independently and without outside help and that I have not used any sources and aids other than those specified in the work. The work has not yet been submitted to any other examination office in the same or similar form.

Hannover, March 19, 2020

_____

Huu Kim Nguyen

Abstract

This bachelor thesis presents a software which checks a software project for libraries that have security vulnerabilities so the user is notified to update them.

External software libraries and frameworks are used in websites and other software to provide new functionality. Using outdated and vulnerable libraries poses a large risk to developers and users. Finding vulnerabilities should be part of the software development. Manually finding vulnerable libraries is a time consuming process.

The solution presented in this thesis is a vulnerability checker which scans the project library for any libraries that contain security vulnerabilities provided that the software project is written in Java or in JavaScript. It uses hash signatures obtained from these libraries to check against a database that has hash signatures of libraries that are known to have security vulnerabilities. The developer will receive warnings if the software detects vulnerable libraries. For every library in Maven and in the Node Package Manager registry, its metadata is used to find any matching entries in the National Vulnerability Database which indicate that the library has vulnerabilities. If a match is found, the hash signature and the metadata is then saved to the library hash signature database.

The hash checker can be integrated as a library into other software, which can then fully automate this process. Once the lengthy process of creating the hash signature database is finished, the hash-based vulnerability checker performs has a slightly better retrieval performance than any existing library vulnerability checker.

Table of contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

CCD	Code Clone Detector
CPE	Common Platform Enumeration
CVE	Commander Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
GUI	Graphical user interface
JS	JavaScript
JSON	JavaScript Object Notation
npm	Node Package Manager
NVD	National Vulnerability Database
OWASP	Open Web Application Security Project

Chapter 1

Introduction

The Internet is part of the daily lives for a large number of people. The amount of internet users in 2019 were estimated to be 4.1 billion users [1]. This is 53% of the world population at that time [2]. As of March 16 2020, Google had to process over 81,000 searches per second or over 7 billion searches per day [3]. To enhance the user experience on websites JavaScript libraries are used. These libraries, especially older versions, may have security vulnerabilities.

jQuery is a JavaScript framework which is used in over 69 million websites [4] and 809,844 websites out of the top million websites by traffic [5]. All versions of jQuery before 3.4.0 have security vulnerabilities with medium severity [6]. While none of then are critical vulnerabilities, vulnerable versions of jQuery are frequently used in websites. 89.7% of those that use jQuery still use any version before 3.0.0 which have security vulnerabilities [7][8]. The next popular library, bootstrap which is used in over 4.4 million websites [9] and 176,103 websites [5] in the top million websites also has medium-level vulnerabilities in the older versions of bootstrap before 3.4.0 [10].

The Node Package Manager alone has over one million packages [11], which are libraries for Node.js. It is a runtime environment that allows to run JavaScript in a standalone environment outside of any web browser [12]. The packages use a large number of dependencies which also have their own dependencies. The average npm package requires a total of 86.55 dependencies, including the nested ones [13]. Two high-profile incidents occurred in which a specific, supposedly newer version of a popular package was actually a malware. In the case of event-stream which currently has 1,741,003[1] weekly downloads [14], a new malicious contributor to the event-stream package added a new malicious dependency flatmap-stream which attempted to steal bitcoins from the victim before the malicious package was removed from npm [15].

A similar incident occurred in the eslint-scope package[16] (p. 3). It has 16,229,279[1] weekly downloads as of February 25 [17]. An account of a maintainer was compromised by an attacker and uploaded a malicious package under the same name. The malware tried to steal user credentials required for npm[18] (p. 1). This version was removed from npm [19].

These incidents do highlight the need to detect vulnerable libraries during the software development before the software goes to production. Even if the software developer ensures that their own code is safe, their used libraries can still be vulnerable, and a library that is safe to use at that time might become vulnerable in the near future if an exploit is found, usually after the software is finished.

The goal of this thesis is to create a software that allows any developer to scan their projects for any vulnerable libraries during the software development so that newer safer versions or alternatives are used instead.

---

[1] npm counts every single download as download, even if its automated[20]

1.1 Problem

Manually searching any vulnerability database like the National Vulnerability Database if any of the used libraries are vulnerable, using the library name, is time consuming and is outside of the expertise of a regular programmer. The option to hire security experts to check for security vulnerabilities does not always exist, especially if the budget or time of the project does not allow that, given the large number of software projects that had budget or schedule overruns [21].

Checking their used libraries repeatably for security vulnerabilities is not feasible with tight project deadlines that can cause schedule overruns [22]. Some programmers will try to ignore any security aspects in order to save time or only look at any possible security vulnerabilities at the end of the development when everything else is done.

Another issue is reusing existing libraries, not just to save time but also to prevent any incompatibilities caused by newer, safer versions of the library. This has the risk of using vulnerable libraries. There is also the possibility that during the protracted development of the software that a dependency which was secure at the start of the development becomes insecure to use when a vulnerability of that dependency is discovered and disclosed during the development of the software.

1.2 Proposed Solution

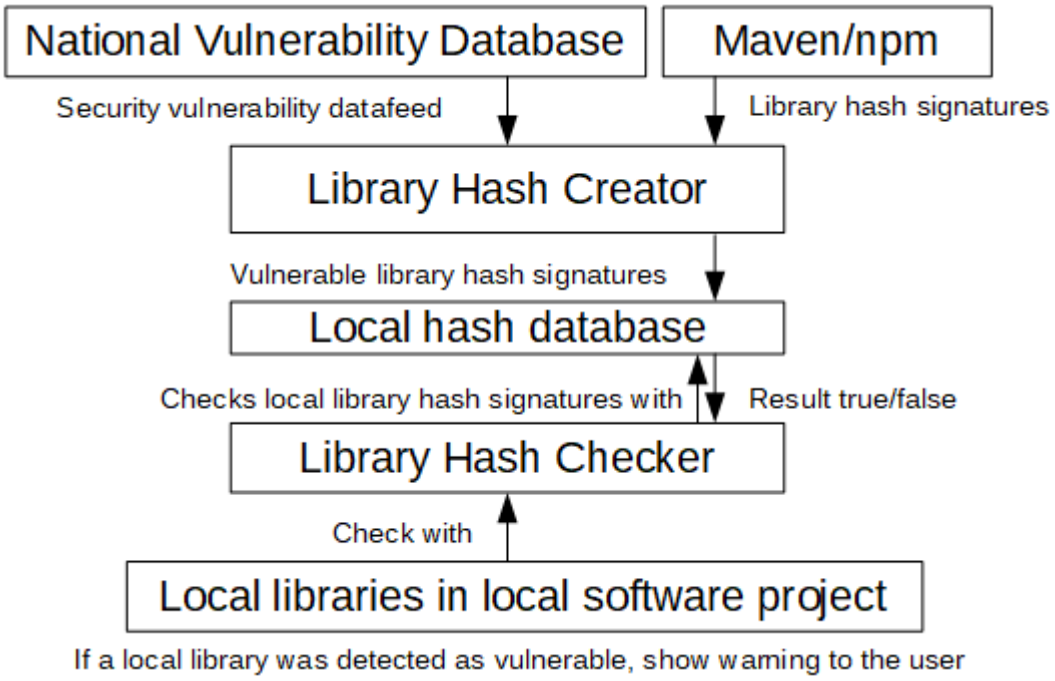


Fig. 1.1: Overall schema of the proposed solution

A tool as depicted in Fig. 1.1 is created which automates the process of searching the National Vulnerability Database for JavaScript and Java libraries. This approach is split into two programs. The server application analyses the National Vulnerability Database and checks in central repositories for vulnerable libraries. Any vulnerable Java and JavaScript libraries have its hash signature stored in a local database which can be accessed by a client.

The leaner client only contains any code required to check their used Java and JavaScript libraries with the local database. Only a connection to the local database is required. If the used libraries are vulnerable, the client detects them as such and warns the user right away so that they can either update to a newer version of the library or find an alternative without any wasted time on the old vulnerable library. The only user input required is the location of the software project. The lean client or its code can be easily integrated into other programs.

1.3 Structure of the Thesis

In Chapter 1 the motivation of this bachelor thesis, the problem and the solution are described. At the end of this chapter the structure of this thesis is described.

In Chapter 2 the functional and nonfunctional requirements of the software created for this thesis are stated and how they are prioritized for this software project. Each requirement is assigned to a package because some requirements depend on other requirements. The stakeholders which may have interest in this software and in this thesis are also mentioned in this chapter.

In Chapter 3 some basics and definitions are described in detail to understand the concepts used in the next chapter. Most described information in this chapter are associated with the National Vulnerability Database (NVD), which includes the Common Platform Enumeration(CPE), the Common Vulnerability Scoring System(CVSS) and the Common Weakness Enumeration(CWE).
The Maven Central Repository and the Node Package Manager(npm) are also described in this chapter, alongside with how hash functions work.

In Chapter 4 all concepts for the software are presented and how they are used in the software. This chapter exactly describes how both the library hash creator and the library hash checker work and why they are separated.

In Chapter 5 the exact details of the software and its architecture are shown. The main concern here are some of the technical details that are required. The structure of the database is described and how the applications access it. There are instructions on how to use the graphical user interface of both the library hash creator and the library hash checker and how to use them if the software if run in an environment without a graphical user interface. This chapter also describes how the library hash checker can be integrated into other software. Some difficulties, drawbacks and removed features are stated in this chapter.

In Chapter 6 relevant related works are mentioned and how the software created for the thesis is different from these related works as well as how some of its ideas were inspired from the related works. The related software are the OWASP Vulnerability Checker and snyk.io. A browser plugin named retire.js also gets a mention here but is not used anymore. Two theses also get mentioned here and how these were the basis for this thesis.

In Chapter 7 the accuracy and precision performance of the created software is measured and a comparison against other similar software to evaluated if the newly created application for this thesis is better than the compared applications. The time performance is also measured and its implications caused by the time performance are described. There is also a short evaluation to compare different hash functions regarding

The final chapter is the overall conclusion and outlook of this thesis. There is a summary of this thesis and a post mortem analysis of this thesis. The outlook also describes how the created software can be used and integrated into other projects and what issues might arise in the near future.

Chapter 2

Requirements

In this chapter the stakeholders and requirements as required for this thesis are presented. The relevant stakeholders are mentioned in section 2.1, followed by the list of every functional requirement in section 2.2 and the list of every nonfunctional requirement in section 2.3. Every requirement is then prioritized in section 2.4 and is put in a package in section 2.5

2.1 Stakeholder

The primary stakeholders are software developers, whose responsibility is to develop secure software. By detecting vulnerable libraries as soon as they are imported to the project, the programmers are informed right away and can either update the libraries to a newer safer version or find a different safer library. This will help to prevent the software being shipped with vulnerabilities. Using this software also allows for developers with fewer skills in security to find and remove vulnerable libraries. This will save man-hours and therefore costs, and with tight budget and time limitations [21][22], it is valuable. The software must not be intrusive in one way or another, it must not be unoptimized and it must be easy to use.

The next set of stakeholders are any workers and students from the Software Engineering Group from the Leibniz University Hanover. Once this software is completed, the ownership will be passed to these stakeholders and as a result, the code must be clean and documented, so that maintenance is easy and it can be easily expanded as they need it. They also want to integrate at least parts of the software to their existing software.

The last set of stakeholders are security experts, they can have a deeper look at the vulnerable libraries and analyze why they are marked as vulnerable. Knowing that the warnings produced by the software may not be absolutely true, they can assess the risks of the project with it, as every project should have a dedicated security expert as in this article [23] by Rohr. They can also contribute improvements to the library hash checker.

2.2 Functional requirements

Here is a list of requirements. Functional requirements are denoted by [FR] while nonfunctional requirements are denoted by [NR]

[FR1] A server maintains a SQL database with hashes from vulnerable libraries from Maven listed in the NVD

A server contains and maintains a SQL database which contains entries consisting of the library name, vendor identifier version number and corresponding hash.

The hash is based of the libraries binary data,, either already in a compressed form in case of Java libraries or by all of its individual files if it is not compressed, such as in case of any npm libraries.

[FR2] A Server maintains a SQL database with hashes from vulnerable libraries from npm listed in the NVD

A similar database must be maintained for npm libraries. Unlike Maven however, npm does not have a proper vendor identification and users do not used packed libraries. Instead, the individual files will be hashed. npm only uses names, so additional checks will be required to find and sort any false positives out.

[FR3] Regular updates of the databases.

This database must update all of its contents on a regular basis, so it always contains the latest known vulnerabilities. The user should be able to configure the update frequency and the total amount of updates required.

[FR4] Client can fetch the database and use it to check their own Maven libraries for vulnerable libraries.

A client must be written to check their used Maven libraries with the vulnerability database. Their own libraries are hashed first before being checked against the database. Their used libraries which are stored in the local repository may have their own recursive dependencies which are also stored locally, which have to be taken into account as well.

[FR5] Client can fetch the database and use it to check their own npm libraries for vulnerable libraries.

Like in Maven, the client must check the libraries for npm which are stored in the node_modules folder in the NodeJS project. Note that this folder also stores any dependencies of the dependencies, so they will be implicitly scanned too in that process.

[FR6] Integration of client hash checker for npm into existing software [kbariazirani2019]

The hash checker for npm must be integrated into [kbariazirani2019], which is a standalone software that uses the Spring Framework. Like above, it should be optional for the user.

[FR7] Cross-Platform compatibility (Optional, low priority)

It should at least run on Windows and on most Linux distributions. If a Mac based computer is available, also try to aim for macOS compatibility. Note that mobile operating systems will not be supported because most IDEs do require a desktop operating system as programming is usually not done on smartphones.

[FR8] Very minimal to no user interaction required (Optional, low priority)

The software developer should use this software without any user interaction or at least only require the initial setup of this software and after that it can run on its own.

2.3 Nonfunctional requirements

[NR1] Software is written in Java

All software must be written in Java, using one single programming language allows to maintain consistency. External libraries via Maven are still allowed.

[NR2] Optimal performance for the client hash checker

The user should not notice that the hash checker is running in the background and at the same time the user should get the results from the hash checker as soon as possible. General usability must not be hampered by this software.

[NR3] Small footprint of the software and the database

A smaller size of the resulting database allows for an easier transport of the database and lower search times when querying the database, as the search time is proportional to the size.

[NR4] Low update times for the database.

To maintain a high uptime, the server must perform its updates as fast as possible. Partial updates are an option, as long as they contain the additional or modified contents, but partial updates must be done daily. It is recommended to perform the updates at night when no one uses the database.

[NR5] Clean code and documentation

In order to keep the software serviceable, the code must be written cleanly and uphold coding conventions. Next to the code is an extensive documentation so that any future software developers can get started right away to maintain

2.4 Priority

Here is the priority list, from the most important to the least important requirements. First the all essential functionality must be done.

- [NR1] sets the basis for all the written software in this thesis and is therefore very essential.
- [FR1],[FR2], the server applications must be done first since the clients will not work without a functioning database. These are critical for the functionality.
- [FR4],[FR5], the standalone clients, are also very important as they allow to detect the vulnerable libraries in the first place, otherwise it just would not work.

The next set of requirements are not critical from a pure functionality standpoint but are either part of the thesis requirements, help quality of life or are performance related.

- [FR6] is part of the actual requirement of this thesis and therefore have a very high priority, after the standalone server and client software are done since these plugins depend of those standalone software. Writing the plugins are still important as using the plugins will not require user interaction afterwards.

- [FR3] is implemented externally. This requires a server side task to perform the automation. A scheduled task that runs this server application every 24 hours should suffice and it only needs to add the newest or modified additions of the NVD.
- [NR5] is actually important since this code will be maintained and updated by other people and they must be able to do that without requiring the help of the original programmer.
- [NR2],[NR3],[NR4] are optimization requirements and depending how the software performs, these can be ignored if the software ends up performing good enough or need to be addressed if the initial versions of the software are running too slowly.

The last set of requirements are optional and therefore have the lowest priorities

- [FR7],[FR8] are optional requirements which are done after all essential and important requirements are fulfilled. Note that the cross platform compatibility might end up being more important if explicit Linux compatibility is required.

2.5 Packages

The requirements must be done in a specific order, since some requirements depend on other requirements. As a guideline, the requirements with the highest priority are done first and the ones with the lowest priority are done last.

Server applications
To get a functional database, [FR1] and [FR2] must be done first.

Standalone client applications
Once the database is up and running [FR4] and [FR5] can be done.

Regular updates
[FR3] should be done next to ensure that the database is always up to date. [FR8] can be handled here with external task schedulers if necessary.

Optimization of the standalone software if required.
Assuming there is enough time left and depending on the current performance, try to improve the performance based on [NR2],[NR3],[NR4]. The software should not run very slowly.

Code cleanup and refactoring
[NR5] mandates a clean code for easy maintainability. This applies to all created code.

Cross platform compatibility
[FR7] requires cross platform compatibility. Using Java as required in [NR1], which already is cross-platform compatible, this should only require a few adjustments

Integration to previous work
Once all the programming is done on the standalone applications, the software will be integrated to the existing plugin which fulfills [FR6]. [FR8] will be implicitly done because once it is integrated to the plugins, this will be fulfilled because the plugins already do not require user interaction.

Chapter 3

Basics

In this chapter some basics are explained which are required to understand the concept of the hash-based security vulnerability checker.

Section 3.1 briefly defines the security vulnerability. A entry in the National Vulnerability Database and its attributes are explained in Section 3.2. In section 3.3 the basics of the hash function is explained with one hash function that is used in the vulnerability checker. The last two sections 3.4 and 3.5 describe Maven and npm repositories respectively.

3.1 Security vulnerability

By using the NVD the definition of a security vulnerability is cited from Common Vulnerabilities and Exposures (CVE), which feeds the NVD [24]. It defines a vulnerability as:

> “A "vulnerability" is a weakness in the computational logic (e.g., code) found in software and some hardware components (e.g., firmware) that, when exploited, results in a negative impact to confidentiality, integrity, OR availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety).“ [25]

If the software has a security vulnerability, which is caused by either design flaws or programming errors, then an attacker can exploit that vulnerability. To name some examples what an attacker can do after exploiting it, they can disclose sensitive information, alter or delete the contents of the database or deny others any service.

3.2 National Vulnerability Database (NVD)

The National Vulnerability Database(NVD) [26] is a security vulnerability database maintained by the National Institute of Standards and Technology(NIST). Created in 2005, it is fully synchronized with the Common Vulnerabilities and Exposures List (CVE List) launched by MITRE in 1999. Both are sponsored by the U.S. Department of Homeland Security (DHS), Cybersecurity and Infrastructure Security Agency. Both NVD and CVE are public and free to use [27].

Each CVE entry only contains a CVE ID number, any arbitrary description and references.

To quote the relationship between the NVD and the CVE List:

"The CVE List feeds the NVD, which then builds upon the information included in CVE Entries to provide enhanced information for each entry such as fix information, severity scores, and impact ratings." [24]

A NVD entry, which has all the information from the CVE entry, also contains three more aspects:

- A Common Vulnerability Scoring System (CVSS) score to quantify the severity of that vulnerability, currently supporting versions 3.1, 3.0 and version 2.0, this allows the developers to assess how important any vulnerability is. The base score vector which is used to calculate the CVSS Base score and that score itself are included.
- A list of Common Platform Enumerations (CPE). Software, hardware and operating systems are enumerated into CPEs. If an NVD entry has that CPE, this means that the software associated with that CPE is vulnerable.
- A Common Weakness Enumeration (CWE) to categorize the security vulnerability.

NVD entries can be updated. A preliminary NVD entry may not have any CPEs, CWEs and CVSS scores which are added later once more information of the security vulnerability is known. The vendor can dispute the vulnerability and the entry gets marked as such until the dispute is resolved. NVD entries cannot be deleted but they can be rejected [28]. Rejected entries do not have any CPEs, CWEs and CVSS scores and should be ignored.

3.2.1 Common Platform Enumeration (CPE)

Common Platform Enumeration (CPE) [29] is a machine-readable naming scheme which allows for a systematic enumeration of affected hardware, software or operating systems. Its syntax is based on the generic syntax for Uniform Resource Identifiers(URI).

All CPEs of version 2.3 must follow this format [30] (p. 3):
cpe:2.3:part:vendor:prodct:version:update:edition:lang:sw_edition:target_sw:target_hw:other

- "cpe:2.3" indicates that it is a CPE version 2.3.
- "part" (mandatory) identifies a hardware(h), operating systems(o) or application(a).
- "vendor" is the creator of the product, either the company name or a single person
- "product" is the standard name of the product.
- "version" is the exact version of the product minus an "Update" attribute.
- "update" is an additional attribute to the version. Examples: alpha, beta, rc1.
- "edition" is a legacy attribute from version 2.2 and is deprecated since version 2.3.
- "lang" is the user interface language, Examples: English, German.
- "sw_edition" is a specific edition of the product: Examples: Pro.
- "target_sw" is the software environment in which this product runs. Example: Node.js
- "target_hw" is the hardware environment in which this product runs.
- "other" is for other information.

Based from the information of another paper [31], only the first five attributes are relevant.

Most of the time, only the first four attributes are used. If the version attribute is missing, then there is a version range in the NVD entry most of the time. Any attribute that are not used use a placeholder(“*”) instead.

```
cpe:2.3:a:gitlab:gitlab:11.5.0:rc1:*:*:community:*:*:*
```

Listing 3.1: CPE Example using as many attributes as possible

In context of the NVD, a NVD entry contains a list CPEs. Any hardware or software that are associated with the listed CPEs that are in the NVD entry have that security vulnerability for all listed versions of that hardware or software. One example is seen in Listing 3.1.

3.2.2 Common Weakness Enumeration (CWE)

89,"Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')"

Fig. 3.2: CWE ID followed by a short description.

Common Weakness Enumeration (CWE) is a detailed, community-developed list of over 900 listed common software and hardware vulnerabilities as of March 06 2020 [32]. By categorizing common security vulnerabilities, it is easier to quantify security vulnerabilities.

The vulnerabilities and its consequences are explained in full detail and how these vulnerabilities can be mitigated, if possible. Note that in-code mitigation is not always feasible if external libraries are affected, in this case updating or changing the affected external library is sometimes the better solution.

The CWE list does not just cover the classical set of vulnerabilities like buffer overflows and injections but also more trivial weaknesses like obsolete GUI elements [33] or poor programming practices [34]. CWEs are also categorized inside a very large tree hierarchy which is too large to be visualized. The entry contains the relationship to other CWE entries [35]. A parent node covers a broader vulnerability while its child node covers a more specialized vulnerability. One example is seen below in Fig. 3.3, where CWE-943 has four child nodes, including CWE-89 which is briefly described in Fig. 3.2, and has the parent CWE-74.

The NVD uses two placeholder, CWEs “NVD-CWE-Other” and “NVD-CWE-noinfo” if no fitting CWE can be categorized and if the CWE attribute is supposed to be empty, respectively. The NVD only uses a subset of every available CWE [36].

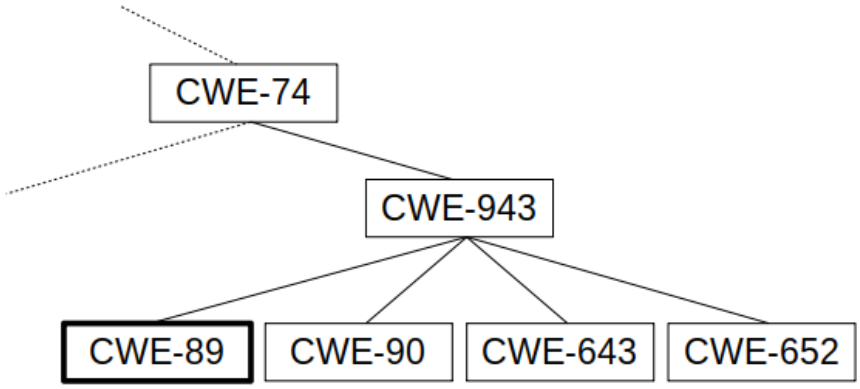


Fig. 3.3: Small portion of the large CWE tree.

3.2.3 Common Vulnerability Scoring System (CVSS)

The Common Vulnerability Scoring System (CVSS) [37] is an open framework that allows to quantify the severity of any vulnerability. By rating base exploitability metrics and base impact metrics the CVSS can provide a score from zero to ten. Further adjustment is possible with temporal metrics and environmental metrics. The CVSS is platform-independent, meaning that this can be applied to hardware and software while the scoring algorithm remains identical for every score.

A more severe security vulnerability gets a higher score while lesser vulnerabilities get lower scores. This allows for an easier prioritization for software developers, highlighting the severe vulnerability that should be fixed first and then the lesser vulnerabilities can be fixed later. To reconstruct the CVSS score, the NVD also provides the CVSS vector next to the score which describe the base metrics in a shortened format, as seen in Fig. 3.4.

Version 3.0 added two more exploitability metrics, User Interaction and Scope while renaming Authentication to Privileges required. The scoring algorithm was adjusted with CVSS 3.0, but remains identical from v3.0 to v3.1, most changes in v3.1 were in the documentation of the CVSS [38]. The full exploitability and impact metrics are described in Table 3.5 and 3.6 respectively. The scoring scale is written in Table 3.7.

AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
Fig. 3.4: Example CVSS vector [37]

Exploitablity metrics describe the overall difficulty of an exploit.

CVSS v2.0	CVSS v3.x	Description (CVSS v3.1)
Attack Vector (AV)	Attack Vector (AV)	This metric describes how “close” the attackers location has to be to the vulnerable component. • “Network”: The attack can be executed from the Internet. • “Adjacent”: The attack occurs from the local network. • “Local”: The attacker requires direct access to the machine • “Physical”: The attacker must be in the same physical location as the device that is to be attacked.
Access Complexity (AC)	Attack Complexity (AC)	The complexity of an attack is described and if it is deterministic. • “Low”: The attack can be consistently executed every time. • “High”: The attack is very difficult to execute or cannot be executed consistently, relying on external factors or luck.
Authentication (Au)	Privileges Required (PR)	The metric describes the privileges the attacker must have before performing an attack: • “None”: Attacker has no privileges • “Low”: Attacker has basic user privileges • “High”: Attacker requires admin privileges
N/A/	User Interaction (UI)	New in v3.0. Describes the required interaction from the victim. • “None”: No interaction from the victim. • “Required”: Victim must perform a specific set of actions in order for the attack to succeed.
N/A	Scope (S)	New in v3.0. If the “Changed” value is used, then the vulnerability can affect software beyond any security boundaries.

Table 3.5: CVSS Exploitablity metrics [37]

Impact metrics describe the damage that is done after a successful attack.

CVSS v2.0	CVSS v3.x	Description (v3.0)
Confidentiality Impact (C)	Confidentiality Impact (C)	This metric describes the amount of confidential information that is disclosed if an attack occurs. • “None”: No information is disclosed • “Low”: Some information is disclosed. • “High”: All information is or some critical information is disclosed. An example for critical information are master keys.
Integrity Impact (I)	Integrity Impact (A)	This metric describes the impact on the integrity or whenever any information after the attack is still the same information that can be trusted. • “None”: No data can be modified. • “Low”: Some data can be modified. • “High”: All data can or some critical data can be modified by the attacker.
Availability Impact (A)	Availability Impact (A)	This metric describes the impact on the integrity or whenever any information after the attack is still the same information that can be trusted. • “None”: Availability is unaffected. • “Low”: Minor performance degradation of the vulnerable system, no full denial is possible. • “High”: The attack can fully deny the service of the vulnerable system or cause permanent damage to the system that affects the availability. An example is an ideal Denial of Service attack.

Table 3.6: CVSS Impact metrics [37]

CVSS v2.0	CVSS v3.x	Rating
N/A	0.0	None
0.0-3.9	0.1 – 3.9	Low
4.0-6.9	4.0 – 6.9	Medium
7.0-10.0	7.0 – 8.9	High
N/A	9.0 – 10.0	Critical

Table 3.7: CVSS scoring scale [37]

There are temporal metrics, which can lower the base score based on three factors,

- If the exploit only exists as proof of concept rather than being a functional exploit
- If the exploit is not widespread
- If there is a fix for the vulnerable software.

Optional Environmental metrics use the same fields as the base metrics, but can be rated differently from the base metrics based on the local context of the user. The NVD does not use temporal and environmental metrics. NVD still continues to use both CVSS v2.0 and CVSS v3.x [39]. CVSS v3.0 did inflate the scores compared to v2.0. In a large sample, 40.23% of all CVSS v2.0 scores were rated “High” while in v3.0 62.53% of all scores are either rated “High” or “Critical” [40].

3.3 Hash function

Hash functions take an input bit string of any length and compute a fixed-length output bit string. These are a deterministic one-way function which means that it is computationally easy to create a hash out of any given input, all the time [41].

Additional properties are required for secure cryptographic hash functions:

- It is practically impossible to calculate the input bit string from the hash.
- It is computationally infeasible to find an input bit string that results the same hash.
- Collision resistance, no two inputs must result the same output bit string.
- Avalanche effect, the slightest change results a completely different bit string. It should not be possible to derive one hash by another hash.
- The output hash bit string must have a length of at least 256 bits.

The main problem is the birthday paradox [42] (p. 125). Using an analogy, if 365 birthdays exist, there is already a 50% chance in a group of 23 people that two of them share a birthday.

$$p_M(n) = 1 - \frac{M!}{(M-n)!M^n}$$

Fig. 3.8: Probability of hash collision [43].

Using the formula from Fig. 3.8, the probabilities of a collision can be calculated. M is the total number of available hashes given the hash size, which is $3.40 * 10^{38}$ when a hash size of 128 bits is used. The number of used hashes are in n. The hash size in bits is k. The average probability of a collision is calculated in Table 3.9 when the hash size is at least 64 bits.
For Table 3.9 the total number of available hashes is given from the hash size in k bits.

Number of used hashes	Probability of collision (rounded)
$2^{(k/2)-8}$	~ 0%
$2^{(k/2)-7}$	~ 0%
$2^{(k/2)-6}$	0.001%
$2^{(k/2)-5}$	0.005%
$2^{(k/2)-4}$	0.02%
$2^{(k/2)-3}$	0.78%
$2^{(k/2)-2}$	3.08%
$2^{(k/2)-1}$	11.75%
$2^{(k/2)-0}$	39.34%

Table 3.9: Probability of a hash collision for a hash size $k \geq 64$ bits

The popular SHA1 [44] and MD5 [45] hash functions are no longer considered to be secure because it is feasible to perform chosen-prefix collision attacks on these hash functions. A normal collision attack requires to find another input m2 that produces the same hash as the original input m1. The chosen-prefix collision attack prepends a different prefix p1 and p2 respectively to each message m1 and m2 so that h(p1 || m1) = h(p2 || m2). As long as the prefixes can be found, they can be easily prepended to the original message. In this context it is theoretically possible to create a malware and with the right amount of byte padding, it can have the same MD5 signature as a legit software library [45].

There are also non-cryptographic functions like MurmurHash3 [46] but regardless of their performance they describe themselves as not secure. This function can only produce a 32-bit, 64-bit or 128-bit hash, which is below the recommendation of 256 bits.

File verification is one main application for hash functions. Due to the avalanche effect and the inability to calculate the original file bytes from the resulting hash ideally only two identical files can produce the same hash signature. It is also faster than a byte-to-byte check. For file verification the main concern is time performance, as in how many hashes can be produces per seconds for any given hash function as a large number of files will be handled. Maven currently uses SHA-1 for file verification, while npm uses SHA-512.
At first glance a hash size of 128 bits seems to be sufficient but the hash size must be large enough to for a large future database. The total number of available hashes for the given hash size can be seen in Table 3.10 below.

Hash size (k)	Total number of available hashes (M)	Recommended maximum number of available hashes ($2^{(k/2)-8}$, probability of collision $7.62 * 10^{-6}$) (n)
64 bits	$1.84 * 10^{19}$	$1.68 * 10^{7}$
128 bits	$3.40 * 10^{38}$	$7.21 * 10^{16}$
160 bits	$1.46 * 10^{48}$	$4.72 * 10^{21}$
256 bits	$1.16 * 10^{77}$	$1.33 * 10^{36}$
384 bits	$3.94 * 10^{115}$	$2.45 * 10^{55}$
512 bits	$1.34 * 10^{154}$	$4.52 * 10^{74}$

Table 3.10: Number of available hashes before a probable collision occurs.

3.3.1 BLAKE2

BLAKE2 is a cryptographic hash function [47]. It is based on BLAKE which is one of the five SHA-3 finalists [48]. It lost to Keccak which was chosen to be the new SHA-3 hash function [49]. The main goal of BLAKE2 is to replace MD5 and SHA-1 in applications that require high performance. BLAKE2 claims to perform faster than SHA-1 and MD5 while not having any of the security flaws. Four different variants are available, the default variant BLAKE2B with a hash size of 512 bits which is optimized for 64-bit systems, the smaller BLAKE2S variant that has a hash size of 256 bits is optimized for 32-bit systems. Two more parallel variants exist that take advantage of multi-core CPUs, the 4-way BLAKE2bp and the 8-way BLAKE2sp variants.

This hash function is notably used in Argon2, a hash function used for passwords that won the Password Hashing Competition [50]. A security analysis on BLAKE2 was done in an independent paper [51] and it was proven to be secure.

3.4 Maven repositories

Maven [52] is a software project management tool primarily for the Java programming language, which provides a uniform build system for every Maven project, with the goal to simplify and speed up software development.

To achieve this, Maven also has multiple software repositories [53] that host software libraries. It uses the groupId and the artifactId to uniquely identify software libraries.

The groupId [54] denotes the owner of this project and the location of the software library within the hierarchy of the libraries of the owner. It has to follow the Java package naming convention [55]. It contains a reversed domain name from which the owner can be derived. The library might also be part of a package hierarchy, any packages are noted. The owner can be derived from this groupId once the top-level domain(TLD) is removed.
The artifactId is the exact name of the library (here *jackson-databind).*

> Example: com.fasterxml.jackson.core
> There is a reversed domain com.fasterxml. The presumed vendor is fasterxml, because com is a TLD. The library is part of the “core” package which is part of the “jackson” package.

3.5 Node Package Manager (npm)

Node Package Manager (npm) [56] is a package manager created in 2010, which allows to manage Node.js libraries and resolve its dependencies automatically. npm is the default package manager for Node.js which is a runtime environment that allows to run JavaScript outside of the web browser [12]. Its registry hosts over a million JavaScript packages [11].

This is not a general repository for standalone JavaScript libraries. Some npm packages do contain these standalone libraries but this is not required. There was an attempt to create a repository for JavaScript libraries [57] but it currently has no activity since 2009.

Chapter 4

Concept

The concept of the library hash creator and the library hash checker are described in this chapter.

In section 4.4 the full functionality of the library hash creator is described for creating hash signatures from Maven libraries, from obtaining a list of every Maven library to the insertion to the database. Section 4.5 describes the different business logic used when the library hash creator has to create hash signatures from npm packages, especially during the actual hash signature creation. Standalone JavaScript libraries that are found in a npm package are separately handled, which is explained in section 4.6. The concept of the library hash checker is then described in section 4.7 which applies to libraries found in both Maven and npm. In section 4.8 there is some discussion about fundamental flaws of the hash-based vulnerability checker.

4.1 Definitions

The two applications and the database are defined here.
- The Library hash creator is an application that creates the hash signatures from the vulnerable libraries that it processes.
- These newly created hash signatures are then stored in the hash signature database.
- The Library hash checker calculates the local hash signatures from local software libraries and checks these hashes against the hash signature database
- The hash-based vulnerability checker refers to the full software of the above three.

4.2 Generic tasks

These are the broadly defined steps for the first two tasks that are performed in the library hash creator and for the third task which is performed in the library hash checker which is depicted in Fig. 4.1.

Importantly, this general process is platform-independent, meaning as long as the programming language in question has a repository of libraries which ideally also has a list of its libraries, it is possible to create a new library hash creator and library hash checker for that programming language that works on the same principle. However, the some of the exact details can be still platform-dependent. For an example, Maven and npm are completely different programs and the exact process of downloading libraries from each different repository can be different.

While it is not required, it is recommended to run Task A before Task B to ensure that the internal index is up to date. Both tasks take place in the library hash creator.

In both Task A and Task B the NVD datafeeds have to be downloaded first.

Task A: Creation of an internal index

1. Create a whitelist containing every created library for that programming language.
2. If a vendor and product name is used, create a tree structure.
   Only in this shortened format: *cpe:2.3:part:vendor:prodct*
   Either a large map is used, with the partial CPE as key and the library location as key
   OR a tree structure from the root: vendor → product → library location
   2a. Optional: Use additional metadata from the library to create more suitable CPEs,
   this will require downloading more metadata which in return takes more time
3. Save the tree. If only a single name is used, the whitelist is sufficient.

Task B: Inserting hash signatures of vulnerable libraries to the hash signature database

1. User provides a maximum age of the NVD entry before before it is ignored.
2. For every yearly NVD datafeed plus the “recent” and “modified” datafeed:
   2.1. For every NVD entry in the datafeed:
   2.1.1 Check if that NVD entry is not too old, otherwise skip the NVD entry.
   2.1.2 Check if the CPE indicates an application, otherwise skip the NVD entry.
   2.1.3 If the CPE indicates an application, check it against the tree structure.
   2.1.4 Check if the provided CPE matches with any of the CPEs from the internal index.
   2.1.5 Obtain the version history of the library.
   2.1.6 NVD entry has the exact version in the CPE or it has a specified version range.
   2.1.7a. Download all affected versions of the library if no hash signature is available.
   2.1.7b. Calculate the hash signature from the downloaded library.
   2.1.7c. If a secure hash signature exists, then download the hash signatures instead.
   2.1.8. Insert or update the CVE ID, any CVSS scores and the hash to the hash database.
   2.1.8a. For research purposes, the name, the vendor, the CWE and the version is inserted.
   2.1.9. Repeat steps 2 to 11 until every datafeed has been processed.

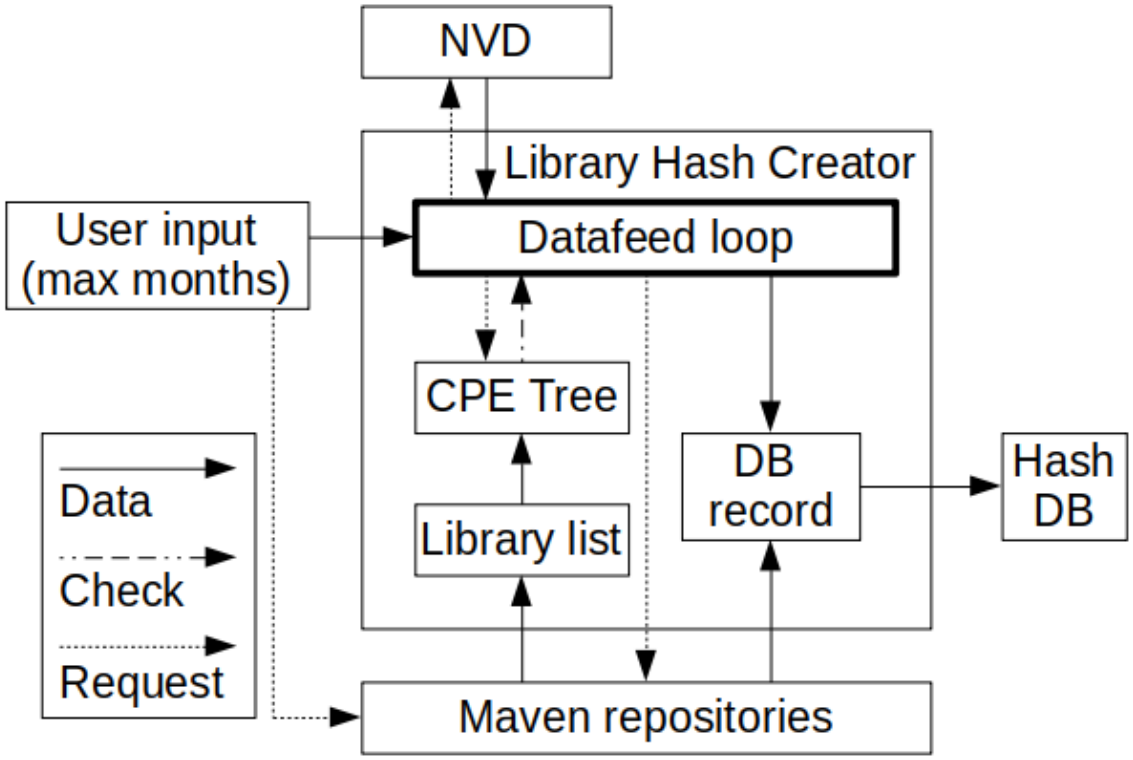


Fig. 4.1: Schematic of the library hash creator.

Task C: Check local libraries with the library hash checker.
1. Specify the directory of the software project to be scanned for vulnerable libraries
2. Detect any software libraries used in the software project
3. For every detected library,
   3.1. Create its hash signature.
3.2. Check the hash database if the hash exists. If true, warn the user.

4.3 NVD JSON datafeed download

The JSON datafeeds need to be downloaded every time the server library hash creator is used from the official NVD website [58]. These datafeeds are updated daily. They are downloaded in a compressed format and must be decompressed at the local computer.

For decompression the external program 7-zip[59] is used because it performs faster than the native zip libraries in Java and it avoids the zip slip vulnerability[60] that is caused by improper implementations of the ZIP decompression. The ZIP archives containing the JSON datafeeds are then decompressed so that the library hash creator can use these datafeeds.

4.4 Library hash creator for Maven libraries

Maven as described in section 3.4 uses a system that involves a groupId and an artifactId to identify libraries in Maven. Therefore it is required to match the CPEs with the groupId and artifactId. The exact process of downloaded the required hash signatures is also described in this section.

4.4.1 Library whitelist for Maven

There is an index file for the Maven Central repository, either as a single large file or as multiple incremental files [61]. All of them cannot be read by normal beans because these are binary files. The binary files do contain the full groupId and artifactId as a readable string.

```
1 \u0001u\u0000\u0000\u0000@com.fasterxml.jackson.core|jackson-databind|2.9.0
```

Listing 4.2: Excerpt from the binary file[1]

The pattern in Listing 4.2 is consistent. This allows the groupId with its associated artifactId to be extracted. From Listing 4.2, the groupId is com.fasterxml.jackson.core and the artifactId is jackson-databind. Right next to the artifactId is the associated version number which is not required for now.
The full explanation for the groupId and artifactId are in section 3.4.

Both are used to identify the project. Using both groupId and artifactId, the full string com.fasterxml.jackson.core/jackson-databind can be formed. A set datatype is used to store every extracted string, which eliminates duplicates. The set is then saved to a local text file which acts as the whitelist

[1]Binary characters are represented in Unicode characters.

4.4.2 Internal CPE tree structure

Every NVD entry has at least one CPE, which always has these two strings, one for the vendor and one for the product name.

No POM file, which contains the metadata for any Maven library, has any stored CPEs. This requires the creation of suitable CPEs by other means.

Every CPE entry contains the vendor and the product name. To allow a faster search, a tree structure will be created. The searching process searches the vendor first, then the product name.

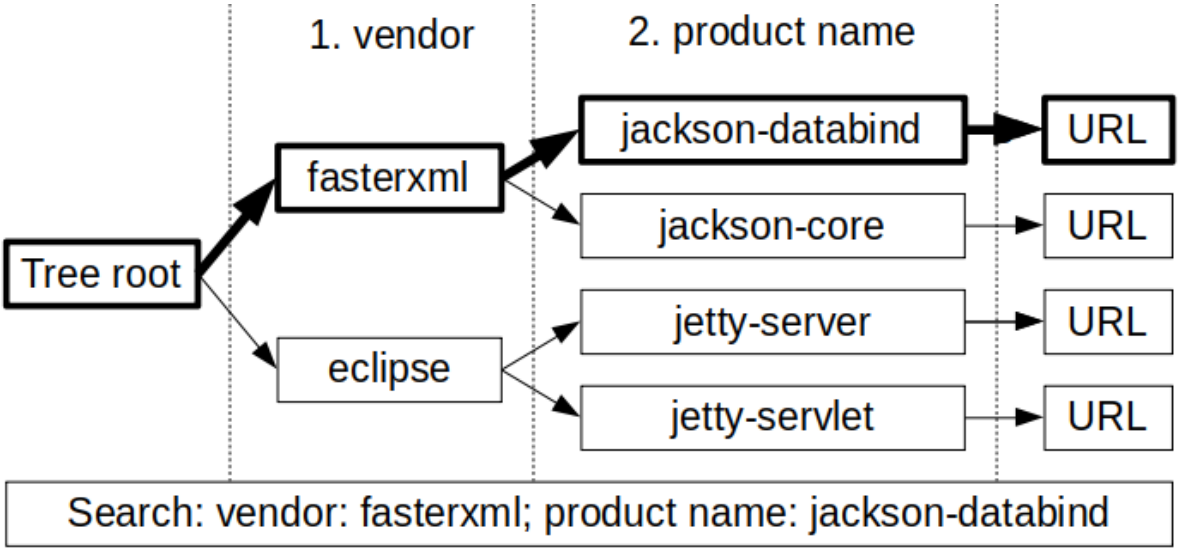


Fig. 4.3 Tree data structure with three levels.

As seen in the example in Fig. 4.3, the first level after the root node only contain nodes with the vendor as key string. Its child nodes are nodes that contain the product name as the key string. The groupId and artifactId are concatenated to a relative URL which is then stored in the leaf node after the product name node.

Duplicate vendor nodes are not allowed. Instead the product name of that duplicate vendor node is assigned to the existing vendor node. Duplicate product nodes names are allowed, as long as they do not share the same parent node. If they do share the same vendor, then the relative URL is assigned to the existing product name node instead.

The tree data structure is implemented using multiple generic data structures. There is a HashMap which uses the vendor string as the key and contains a nested HashMap as value. The nested HashMap is acts as the child node.

That nested HashMap uses the product name string as the key and uses a HashSet that contains URL strings. Most of the time the HashSet only contains one string but it can store two or more URL strings if it is ever required.

4.4.2.1 CPE tree insertion

Relevant metadata	Example value (used value)	Can be used as
groupId	com.fasterxml.jackson.core	Vendor (fasterxml)
artifactId	jackson-databind	Product name (jackson-databind)
Home page (URL)	github.com/FasterXML/jackson	Vendor (fasterxml)
Organization URL	(not used in this example)	Vendor (similar to home page)

Table 4.4: Relevant metadata from the POM file.

To obtain the vendor from the groupId, any top-level domains in the groupId have to be removed first. The resulting substring is still separated into more substrings by dots. The first substring after any TLDs is used as vendor. (Example: com.fasterxml.jackson.core)

As product name the exact artifactId is used.

First search for the node “fasterxml” from the root. If it does not exist, create that node, otherwise use the existing node. From that vendor node, create the product name node “jackson-databind” if it does not exist, otherwise use the existing product name node.

Concatenate the groupId and the artifactId, using a forward slash(“/”) as divider. Put the created relative URL in the leaf node after that product name node.

This only contains the relative location in Maven, not actual library hash signatures. Both vendor and product name do not use uppercase characters, any uppercase characters are converted to their lowercase equivalent.

The POM file of any Maven library optionally contains an URL to the homepage of the library, which is usually part of the vendors homepage and an URL to the homepage of the organization behind that library. These URLs contain domains which can be used as vendors in any CPE.
That file also contains the groupId, artifactId which are already obtained from the library whitelist and any additional metadata which are not relevant for this process.

4.4.2.2 CPE tree search

Two strings are provided to perform a search in the internal Maven index, the vendor and the product name, in that order. These two are obtained from any CPE.

First the search checks if there is any child node from the root that has the exact vendor name as the provided vendor string used for the search.

If no such node exists, stop the search and return a null string. Otherwise, traverse to the matching vendor node.

From that vendor node, search for every child node with a name that starts with the provided product name string used for the search.

The reason for this looser search is because CPEs do not specify the exact subpackage that is vulnerable. The CPE only contains the product name of the broad library, which is divided into multiple subpackages. In some cases multiple subpackages are vulnerable. Due the lack of detailed information, assume that all subpackages are vulnerable.

The product name node contains the full URL which is the location of the library in Maven. The URLs are added to a temporary set which is used to obtain the library hash signatures. Repeat from the vendor node until every suitable child node was searched

Using this example from Table 4.4, the search first looks for a vendor node named fasterxml. This exists, so traverse to that node and then look up for a node named jackson-databind. That product name node exists to which contains an URL. The full URL containing both the associated groupId and artifactId are added to a temporary set.

Look up for any other product name node that starts with jackson-databind. This is not the case, so only the URL from jackson-databind is used for further processing.

4.4.2.3 Edit distance

Currently the maximum allowed edit distance is zero, except for removing any underscores and hyphens if they are present. If there is a product name that uses hyphens, then two product names are inserted, the original name and one without any hyphens. This design decision is to keep the number of false positives low, but it also may increase the chance of false negatives.

4.4.3 CPE validation

```
1 cpe:2.3:a:fasterxml:jackson-databind:2.9.0:*:*:*:*:*:*:*
```

Listing 4.5: Example of a valid CPE using a single version

Every yearly NVD JSON datafeed plus the datafeed with the most recent additions of the NVD and the datafeed containing the most recently modified CVE entries are processed.

In every datafeed there is a list of NVD entries. Most CVE entries have associated CPEs. Every CPE will be processed by the server library hash creator individually.

If the second substring is not equal to the character “a” then the CPE is ignored. The “a” means application, the only other two options are “o” for operating system or “h” for hardware. All software libraries fall under the application category.

The CVE entry also contains a timestamp regarding its last modified date. Depending on the user settings, older CVE entries can be ignored.

The algorithm described in section 4.3.2.2 is then applied, using the vendor string and the product name string from the CPE as input. This should return a list of URLs that require further processing. One example of a valid CPE is seen in Listing 4.5.

Optionally the CVE ID can be searched in the snyk.io vulnerability database [62]. If this database returns a result, that result also contains the exact location of the library in Maven and the exact subpackage that is vulnerable.

4.4.4 Discover of correct Maven repository

Not every Maven repository is stored in the Central Maven Repository [63] and there are a total of 1267 Maven repositories as of Mar 05 2020 [64].

Mvnrepository [64] is a web page that allows any user to search for any Maven software libraries. Every library has its own page in Mvnrepository which lists all every version of the library that is stored in its primarily used repository. Some libraries use multiple repositories. By default this website[1] [65] shows the primarily used repository, this is the one be used. This page has a link [66] to the repository which leads to another informational page about the repository that contains the actual URL [63] that is required.

Using that URL, the full URL can be concatenated[1] [67]. The repository URL is concatenated with the groupId with its dots replaced by forward slashes(“/”). This is concatenated again with the artifactId. At the end a forward slash is appended to the URL.

4.4.5 Version matching

There are two ways to record which versions of the software library are vulnerable. If only a single version is affected, then the version string in the CPE is used to indicate the vulnerable version.

If multiple versions are vulnerable, then either multiple CPEs with the version string are used or a single CPE is used with a placeholder version string(“*”). Instead, next to the CPE is additional information of the affected version range. Every version that falls within the version range is vulnerable. One example is depicted in Fig. 4.6.

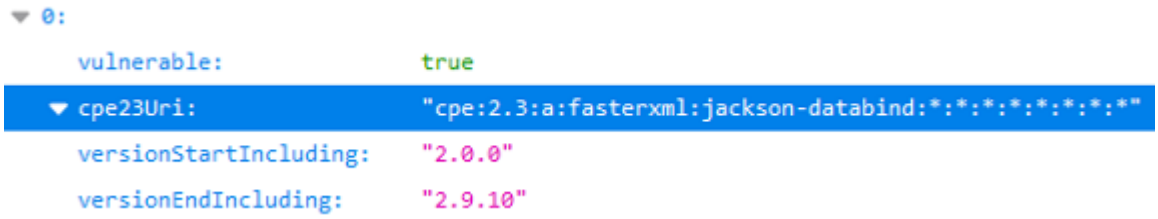


Fig. 4.6: Version ranges in the NVD entry from the JSON datafeed

To denote the last vulnerable version, the field “versionEndIncluding” is used, which marks every version up to the last vulnerable version as vulnerable, unless there another field named “versionStartIncluding”, which is the earliest version that is vulnerable. If that exists, earlier versions than the first affected version are not marked as vulnerable.

There is a variation for each field, for the former there is “versionEndExcluding” and for the latter there is “versionStartExcluding”. In both cases, the listed version is not vulnerable, but earlier versions in case of “versionEndExcluding” and later versions in case of “versionStartExcluding” are still vulnerable.

“versionEndIncluding” and “versionEndExcluding” are mutually exclusive, the same applies for “versionStartIncluding” and “versionStartExcluding”.

4.4.5.1 Semantic version normalization

Most but not all version numbers follow the semantic versioning convention [68] (p. 1), which uses three integers in the version number. To account for any versions that do not follow this convention these version numbers have to be normalized.

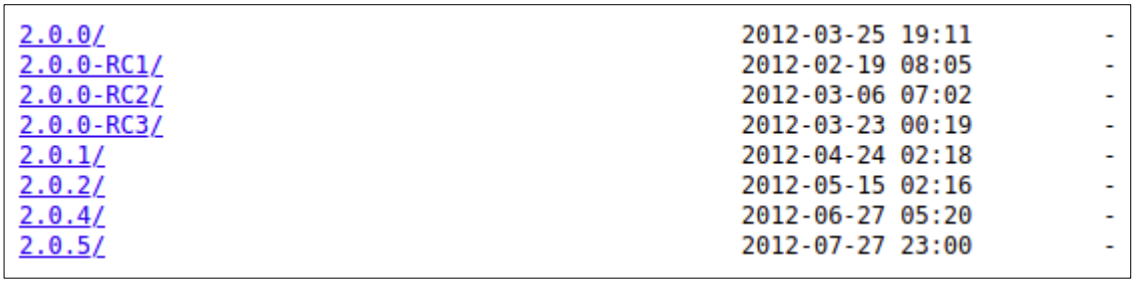
First every character that is not an integer or a separating dot is removed.
Then for every version number that use fewer than three integers additional zero integers are padded with a dot to separate every integer, until the normalized version number has three integers and two dots to separate each integer.

The same applies in reverse applies if the version number uses more than three integers. Every integer after the third one is removed with its separating dot. The resulting normalized version also contains three integers with the two separating dots.

For example, the version “2.9.0.1-alpha” gets normalized to version “2.9.0”. This allows for an easier comparison from version to version.

[1] https://repo1.maven.org/maven2/com/fasterxml/jackson/core/jackson-databind/

4.4.5.2 Library version information



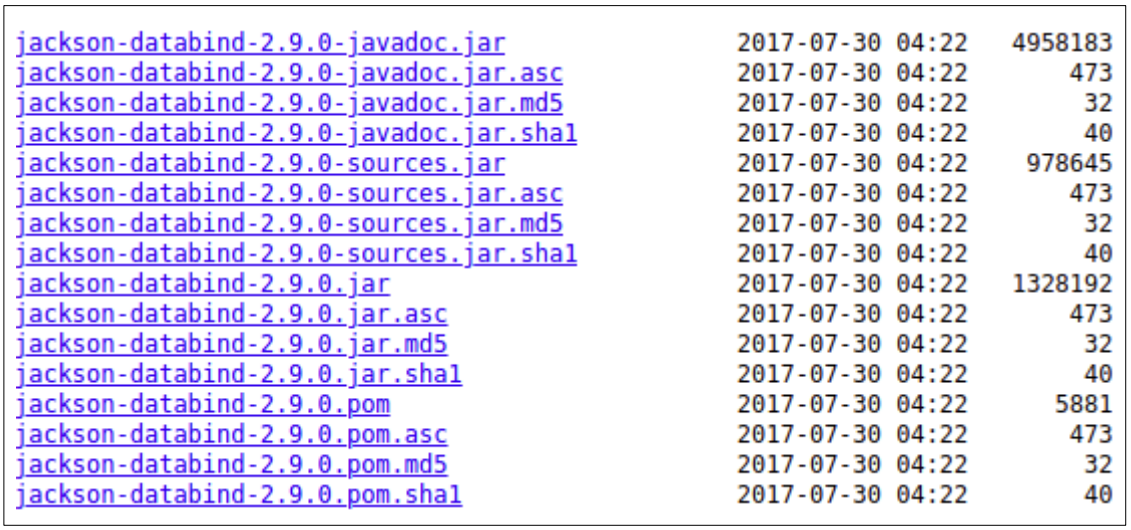
2.0.0/	2012-03-25 19:11	-
2.0.0-RC1/	2012-02-19 08:05	-
2.0.0-RC2/	2012-03-06 07:02	-
2.0.0-RC3/	2012-03-23 00:19	-
2.0.1/	2012-04-24 02:18	-
2.0.2/	2012-05-15 02:16	-
2.0.4/	2012-06-27 05:20	-
2.0.5/	2012-07-27 23:00	-

Fig. 4.7: Excerpt from the version information of jackson-databind. Source: [67]

To figure out which versions are to be processed first the full version page needs to be downloaded.
The version page lists every available version of that library. Using the information from section 4.5 only the single vulnerable version listed in the CPE or all versions that fall in the version range are looked into detail. One example is seen in Fig. 4.7.

If a version range is used, any version that have a lesser or equal version number than the version number listed at the field “versionEndIncluding” and if the “versionStartIncluding” field is used then the version number has to be greater than the one listed at that field.
Every directory of any vulnerable version is opened to extract the hash in section 4.4.6.

4.4.6 Insertion of library hash signatures



jackson-databind-2.9.0-javadoc.jar	2017-07-30 04:22	4958183
jackson-databind-2.9.0-javadoc.jar.asc	2017-07-30 04:22	473
jackson-databind-2.9.0-javadoc.jar.md5	2017-07-30 04:22	32
jackson-databind-2.9.0-javadoc.jar.sha1	2017-07-30 04:22	40
jackson-databind-2.9.0-sources.jar	2017-07-30 04:22	978645
jackson-databind-2.9.0-sources.jar.asc	2017-07-30 04:22	473
jackson-databind-2.9.0-sources.jar.md5	2017-07-30 04:22	32
jackson-databind-2.9.0-sources.jar.sha1	2017-07-30 04:22	40
jackson-databind-2.9.0.jar	2017-07-30 04:22	1328192
jackson-databind-2.9.0.jar.asc	2017-07-30 04:22	473
jackson-databind-2.9.0.jar.md5	2017-07-30 04:22	32
jackson-databind-2.9.0.jar.sha1	2017-07-30 04:22	40
jackson-databind-2.9.0.pom	2017-07-30 04:22	5881
jackson-databind-2.9.0.pom.asc	2017-07-30 04:22	473
jackson-databind-2.9.0.pom.md5	2017-07-30 04:22	32
jackson-databind-2.9.0.pom.sha1	2017-07-30 04:22	40

Fig. 4.8: Raw list of files of jackson-databind 2.9.0. Source: [69]

The directory named after the version number contains multiple files, including the jar archive that contains the Javadoc documentation, the jar archive containing its source code and importantly, the jar archive of the actual library. Every file has a SHA1 hash file which stores the hash as a string in hexadecimal notation. Figure 4.8 shows the raw list of availabeOnly that file is downloaded to the local computer and is read. The hash is then extracted from the hash file.

Some additional information is required, the NVD entry contains a CVSS v2.0 score and in newer NVD entries there is a CVSS v3.x score.

To categorize the NVD entry most of them have a CWE field that contains the CWE in which the NVD entry is categorized.

Finally to enumerate the NVD entry within the vulnerability database every entry uses a CVE-ID, derived from the original CVE entry, from which the NVD entry build upon (see section 3.2 for more details)

Now that all information exists, the CVE ID, the name, the vendor, the hash signature, both CVSS scores and the CWE, these are inserted as a single record to the hash signature database. No duplicates are allowed, if this entry already exists then the CVSS scores are updated if the existing scores are different than the ones in the NVD entry.

These scores can be updated because NVD entries may not any CPEs and CVSS scores when the entry is initially created, over time as more information about the vulnerability is known the CPEs and CVSS scores are created and updated if necessary. There are also disputed and rejected NVD entries, if the NVD entry is disputed then the vulnerability might still be there. Rejected NVD entries do not have any CPEs and are not included [28].

4.5 Library hash creator for npm packages

The process of finding vulnerable libraries in the Node Package Manager is simpler because its packages only use a single name to identify packages. Once the name is taken by the author of that package, no one else can use it. This means that from the CPE, only the product name string is used and the vendor string is ignored.

This does carry a higher risk of false positives since only a single name is used, which is addressed in section 4.5.2.

Most of this process in section 4.5 is similar to section 4.4, but applied to npm instead to Maven. As such, only npm-specific details are described. The CPE tree required to identify Maven libraries from the CPE is not required here and a simple whitelist consisting of single names are used instead.

As in section 4.4.3, every available NVD datafeed is processed, which can be limited by the “maximum months passed since the last modification” number.

4.5.1 Package whitelist for npm

Using the npm package all-the-package-names [70] a list of every currently used package name in npm can be downloaded.

This acts as a whitelist, any other name not in the whitelist cannot be a npm package. If the name check passes, then it checks if the package with the exact version or the latest version in case of a version range exists by trying to download its packed package using “npm pack {name}@{version} --ignore-scripts”. If the specific version does not exist then it is not a npm package and that NVD entry is ignored. The same valid CPE check as in section 4.4.3 is applied here to check for a valid CPE.

4.5.2 False positive detection

To detect a potential false positive entry, the vendor is extracted from the CPE in the NVD entry.
The website of the npm package in npmjs is scanned for hyperlinks. If any of the hyperlinks, after removing the TLDs, contain the vendor then the NVD entry is a true positive entry.

A different example is chosen here, this time a vulnerability which affects the product name “xcode” from the vendor “apple” [71]. Because xcode is in the list of used package names in npm, the hash creator initially accepts the NVD entry for further processing. To determine if it is a false positive, the website of the npm package “xcode” [72] is scanned to find any hyperlinks that contain the word “apple”. No such hyperlink exists on that page, therefore it is a false positive.

However, because there is a chance that any false positives might actually be true positives, the hash signature database still includes these false positive entries. To differentiate them from the true positives, an additional value is used in a separate field. The “probablyTP” field uses 1 if the entry is probably a true positive, which means that it was not detected as a false positive, while 0 is used if the entry was detected as a false positive.

Note that in general that if there are any npm packages listed in the NVD, the listed vendor in the CPE usually matches the product name in the CPE or the vendor starts with the same name but the vendor string is concatenated with “_project”, “js” or “.js”. If this is the case, then it is presumed that it is a npm package and any false positive checking is skipped.

4.5.3 Obtaining npm packages

Instead of downloading the package using “npm install”, the packed tarball is downloaded using “npm pack”. Only the packaged tarball is downloaded without any dependencies. For downloading multiple versions this is faster than using “npm install” which is usually done when installing npm packagages. If “npm install” is used then npm would also try to resolve any dependencies and it takes up more hard drive space. The tarballs are compressed and only consist of a single file which are faster to download.

Using 100 random packages in one test run, using “npm pack” only required 5 minutes while using “npm install” required 30 minutes. There is also the risk of installing malicious packages when “npm install” is used, the latter option should not start any scripts which is the main mechanism of malicious packages but using “npm pack” reduces the risk further since the code is in a tarball archive and cannot run without extracting the files from the tarball first.

Finally, it is possible in the pack command to download multiple packages in a single command, for example, “npm pack lodash@4.2.0 lodash@4.1.0 lodash@4.0.0”, which allows for an even shorter download speed compared to install. This cannot be done with the install command since the node_modules directory can only contain one version of the package at any given time.

If the CPE from the NVD entry uses a single version then the exact packed tarball of the affected package version is downloaded using “npm pack [name]@[version] –ignore-scripts”. If that CPE uses a version range, then the information regarding all available versions of the package need to be downloaded first using “npm view [name] versions”. This must be done in every scan because the version information gets updated over time as newer versions are released to npm. The information regarding every available version of that packages is saved to a text file named “[name]-versions.txt”, replace [name] with the package name.

That text file is processed to find suitable version numbers. Only the listed versions greater or equal to the minimum specified version number in the version range and listed versions lesser or equal then the maximum specified version number in the version range are to be downloaded using the “npm pack [name]@[version] –ignore-scripts” command.

Regarding any downloads of npm packages, if there is already an entry in the hash signature database then the packed tarball will not be downloaded again.

4.5.4 Hash signature creation from the npm package

While the packed npm package tarball has a SHA-512 hash signature, the issue is that the contents of the tarball are not exactly identical to the installed package in any Node.js project. The installed package has a larger package.json file which is different to the package.json file found in the packed tarball, The installed library also has hidden files and sometimes an internal node_modules directory containing more dependencies which are not present in the contents of the packed tarball.

A different approach is used here which applies to the server library hash creator and to the client library hash checker.
The tarballs contents are extracted to a temporary directory. From the temporary directory, every file is hashed except files is named “package.json” or files starting with a dot which indicates a hidden file. This operation is applied recursively to every nested directory except the “node_modules” directory.

Every hash signature is saved to a temporary text file. The text files contents start with the full name of the tarball followed by all hash signatures of every valid file. After no more files are found in the temporary directory that were not already hashed, the temporary text file gets hashed. The newly created hash is then used as the hash signature of the npm package. The same database insertion process as in section 4.4.6 is used, except npm packages have no vendors so no vendor string is inserted to the hash signature database

This also allows to use any hash function. For this software, the BLAKE2B-512 hash function described in section 3.8.1 is used. All temporary files are deleted after this process.

4.6 Standalone JavaScript libraries

There is no actual repository that contains pure JavaScript libraries. OpenJSAN[57] was an attempt in 2007 to create one but there is no activity since 2009.

During the hashing process in section 4.5.3, certain JavaScript files can qualify as standalone JavaScript library. Using the existing information from the Code Clone Detector [31], most standalone JavaScript libraries start with their name in the file name.

Multiple versions of standalone libraries exist. Sometimes instead of a full JavaScript library a “minified” JavaScript file is used which removes unnecessary whitespace characters and some redundant elements like semicolons and comments. An minified JavaScript file is unreadable by humans.

Because online minification tools exist which may have minfied the JavaScript file in a different way, only official minified versions of the JavaScript library which have a minified version inside the npm package are supported in the library hash creator because the exact minification algorithm may be different for every different minification tool, which creates a slightly different file. Even the slightest differences will cause a different hash signature

Using the lodash example which does have a standalone library, if there is a JavaScript file that starts with “lodash” then said JavaScript file is very likely a standalone library.

The single file is hashed using BLAKE2B-512 described in section 3.3.1.

The hash is then inserted alongside with the CVE ID, name, version, CVSS scores if they are available and the CWE to the SE hash database under a different table reserved for standalone JavaScript libraries named “VulnerableRawNPM”.

4.7 Library hash checker

The library hash checker is a separate application that is not directly connected to the library hash creator. The user must provide the library hash checker with the location of the software project that is to be scanned as well as the location of the library signature database and its login credentials. The entire process of the library hash checker is depicted in Fig. 4.9.

4.7.1 Standalone Java and JavaScript libraries

From the project directory, every directory is searched recursively to find any jar archives containing Java libraries and JavaScript files.

The hash signature from the jar file is created from the SHA-1 hash function which is the hash function used in Maven for file verification.

JavaScript files are hashed with BLAKE2B-512 instead, since this the same function used for the hash signature creation that involves npm packages.

The hash signatures are then checked against the hash signature database. If the hash signature exists in the database then it returns a positive result and the user is warned

4.7.2 Maven projects

If the project is a Maven project which contains a pom file in the project directory then the list of all dependent libraries are fetched using the external program command “mvn dependency:list”. A full list is written to an external temporary text file containing all libraries required for the software project, including its dependencies. Every line from that list of processed.

All Maven dependencies are stored in an “internal Maven repository”[2] on the local computer. The exact location is derived from the groupId and the artifactId of the Maven dependency. The groupId with the dots replaced by the file separator is appended to the file location.

Using the artifactId logback-classic with the groupId “ch.qos.logback” and version “1.2.3” as an example, it is located at “C:\Users\e-laptop\.m2\repository\ch\qos\logback\logback-classic\1.2.3\logback-classic-1.2.3.jar”. The dots in the groupId must be replaced by the platform-dependent file separator. In Windows its a backwards slash “ \” while Linux uses a forward slash “/” albeit Windows 10 can handle the forward slash “/” as file separator.

[2] Default location of internal Maven repository is “C:\Users\[USERNAME]\.m2\repository”

The SHA-1 hash function is applied to any jar archive that contains the jar library, because this is currently the officially used hash function in Maven. That hash signature is then checked against the hash signature database.

4.7.3 npm projects

If the project directory contains a node_modules directory then it detected a Node.js project. The same algorithm described in section 4.5.6 is applied on each npm module found in the node_modules directory, the resulting hash from each npm module is then checked against the hash signature database.

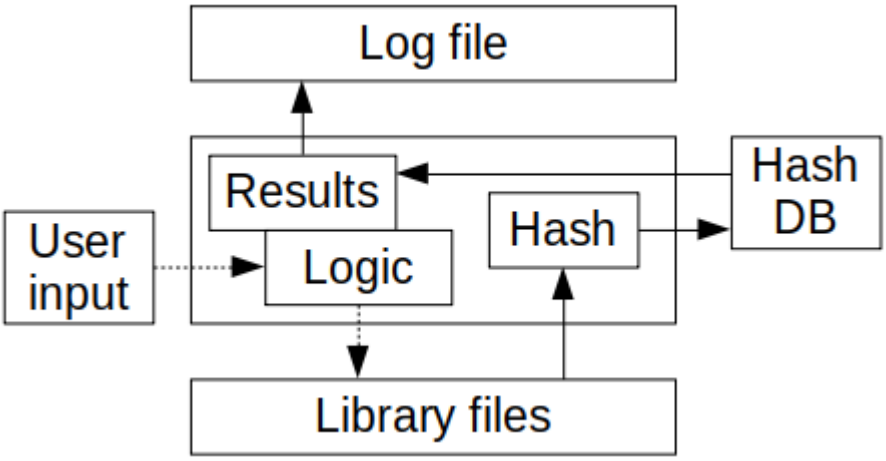


Fig. 4.9: Schematic of the library hash checker

4.7.4 Log file

All results, positive and negative, are stored in a verbose log file, which can be processed in other programs or used to evaluate the scan by the library hash checker.

4.8 Fundamental drawbacks

Even if some of the difficulties can be solved with more powerful hardware, there are a few fundamental drawbacks that cannot be resolved by using a hash signature database to check for vulnerable libraries

The tight integration with the NVD as given by the requirements means that no other vulnerablility databases can be used alternatively if the NVD is not available. Currently this application relies solely on the JSON datafeeds provided by the NVD to find vulnerable libraries.

Any approach that relies on a database that catalogues vulnerabilities cannot detect any zero-day vulnerabilities [73] (p. 1). From the example of the introduction, the package event-stream version 3.3.6 had a malicious dependency flatmap-stream version 0.1.1[15] that was not discovered for a month after the malicious package was published.

Only heuristic approaches and code clone detection techniques might detect zero-day vulnerabilities but these rely on how previous vulnerabilities and exploits worked, a zero-day exploit that uses a new approach is undetectable.

If the library is modified by other means, either by the user, hard drive corruption or by a malicious third party, then the library has a different hash signature. Any slight difference in the library causes a completely different hash signature, as required by the avalanche principle for cryptographic hash algorithms [41]. A different hash signature will not match with the existing hash signature in the hash signature database, causing a negative result.

Any operations that require accessing a website have a significant time penalty. To obey polite web crawling [74], there is a random delay from two seconds to 15 seconds before a HTML document of the website is retrieved. These delays are required, otherwise the user gets temporarily banned from the website for accessing it too frequently or if no bans are in place it can overwhelm their servers, which is basically a denial of service which has to be avoided.

Currently the library hash creator relies heavily on some websites, mostly mvnrepository for a large number of operations and npmjs for the false positive check. If any of these websites have a significant website layout change, which cannot be predicted, the library hash creator would not function anymore and require manual fixes.

The reason why these websites are accessed in the first place are the relevant metadata that is stored in these websites, usually in hyperlinks, that are required for the basic functionality of the library.

4.9 Removed features

One idea was to use a whitelist along with the current hash data to check if the current library is identical to the official library. The whitelist would detect if the user uses a different, possibly malicious library than the officially available library.

This is currently infeasible due the large amount of data that would need to be processed and stored. Assuming that downloading one jar file requires a second, downloading 16 million jar files [64] would require over 185 days. As there six months pass, newer libraries have already been added to the Maven repositories, so the time required is slightly higher than the 185 days to take account of these newly added libraries.

The database where the whitelist would be stored would not be small. Assuming that the name, the version, the vendor, the hash and the CVE ID would be saved, filling a database with 16 million records will require multiple gigabytes of space. A smaller database footprint can be achieved with compression.

Said whitelist would also have to be updated daily. When a software library has been released, it may require up to 24 hours until that library is in the whitelist. Until then the library would be wrongfully detected as a suspicious because it was not in the whitelist during that short window.

Another removed feature in the library hash checker was a spell checker for the npm libraries. Malicious packages, which mostly rely on typographical errors from the users. However, when the Damerau-Levenshtein distance [75] with an edit distance of one is used to compare the remaining 990,000 package names to the top 10,000 package names in npm, it turns out that there are 17,767 positive results instead of the expected 296 known malicious packages[76] in npm. Another indicator of a malicious package are a low number of versions, as a legit package is frequently updated. Restricting the maximum number of versions to five lower the number of positive results to 11,805 which is still too high.

A “high recall” configuration was implemented and briefly tested, which only requires a matching library name. The excessive time required to process all entries and the high chance of false positives meant the removal of this configuration. Out of 268,237 entries in Maven, 46,498 share a name, which would have increased the chance of a false positive.

Chapter 5

Implementation

In this chapter some of the exact implementation details and the overall architecture of the software are explained. In section 5.1 the software architecture is introduced and its components are explained in the next sections, the database is explained in section 5.2.

The applications and their usage are presented in section 5.3 for the library hash creator and in section 5.4 for the library hash checker. Finally some of the difficulties and the drawbacks of the software are discussed in section 5.5 and 5.6 respectively.

5.1 General architecture

The architecture depicted in Fig. 5.1 used on both hash creator and hash checker applications are different because they were primarily designed to run from the command line without any graphical user interface (GUI). The rationale here is that servers may not have any graphical user interface unlike most personal computers and therefore are incapable of running any Java applications that use any graphical user interface like Swing, AWT and JavaFX.

These applications also only require command line arguments and two additional console prompts for the username and the password. After that no further user interaction is required. The hash creation process does require a long time to run, from about 5-30 minutes for a fast scan to two full weeks for a full scan when the library hash creator is used for the first time. It is therefore highly recommended to run this application in the background so that other tasks can be performed.

There is a slim view which is only used if the GUI is used and it only exists for user input. The GUI framework used in Swing, as JavaFX requires a separate start method to launch a JavaFX application and the estimated overhead of JavaFX is greater if JavaFX is used as an optional GUI, which requires GUI and non-GUI code.

Once the task is started in the GUI the GUI does not accept any user input until the task is finished. Technically no view exists if no GUI is used due the lack of interactivity outside of the arguments and the login prompt. Not using a GUI also saves on some system resources.

Two more domains exist, the first domain handles any business logic. It does the require user inputs that were provided from the GUI or as launch arguments. Each application uses its own separate logic required to perform its tasks. The concept behind the business logic is already explained in Chapter 4.

The last domain is a model. It is the hash signature database that is accessible from both applications. It stores every hash signature from any vulnerable Java and JavaScript libraries. Further details for the database is the next section.

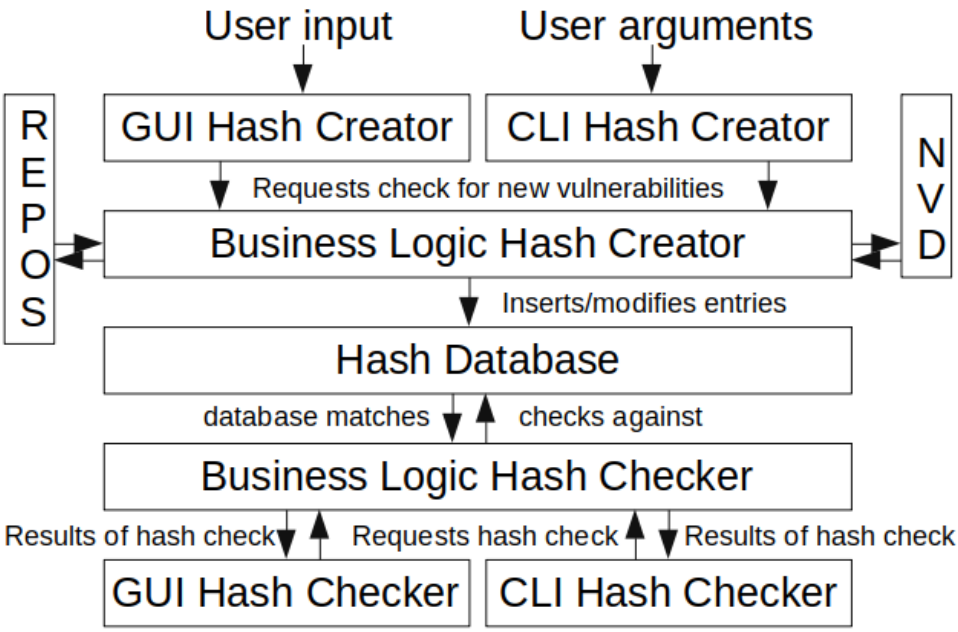


Fig. 5.1: General software architecture of the library hash creator and checker

5.2 Database

One core component out of the three of this entire application of the hash database.

The hash database is split into two databases, one that stores all hashes from Java libraries which are packaged in jar archives, the other one stores all hashes from npm. It contains npm package hashes, as being used in a Node.js runtime enviroment and all relevant raw JavaScript file hash signatures that were found in npm. Having them separated prevents any potential concurrency problems that would occur if simultaneous writes happen on a single database.

The databases are stored online as MySQL/MariaDB database. The library hash checker can download the database as a SQLite database for offline use. For testing the library hash creator can write to a local SQLite database which is inaccessible to the clients.

Currently the full database in the high precision configuration requires about 79MB. If for some reason the obsolete inaccurate high recall configuration is used, then each year would require up to 200MB, each.

For the Java hash database, the database is split again into multiple tables, for each year, to prevent one large table of 1.000.000+ records that would have incurred performance penalties. Each table typically stores up to 100.000 records. The hash signature database for JavaScript and npm libraries has far fewer records. Only two tables are required, one to store all npm hash signatures and another one to store all standalone JavaScript library hash signatures.

Using both SQLite and MySQL/MariaDB means that some exclusive SQL statements used in MySQL cannot be used because it needs to run on SQLite.

If any values are stored as varchar then varchar(255) is used, as no libraries that have names longer than 255 characters are known. This length is mainly used to handle any possible outliers with very long names.

5.2.1 Recommended access privileges

The basic client user that only uses the library hash checker should only have the permission to use any SELECT statements which do not perform any write operations to the online library hash signature database. The library hash checker only requires read operations.

This does not apply to the offline database that can be downloaded from the user. The user has full administrative privileges on their own offline database.

Once the offline database is downloaded, its is disconnected from the online database, meaning that if any change occurs in the offline database, the online database will not be affected by these offline changes.

The server operator account which uses the library hash creator must have the ability to perform SELECT, INSERT and UPDATE statements, because the library hash creator must not just read the library hash signature database but also insert new database records and update existing database records. This account should only be used for the library hash creator and is not to be used in the library hash checker.

The administrator has full administrative privileges, including the ability to drop tables and delete the complete database. Any administrative operations that are not covered by the library hash checker and the library hash creator will require the use of an external SQL editor. Ideally, the administrator account should never be used, because all database operations are handled these two applications. The only purpose is to fix any unforeseen errors that might occur in the future.

5.2.2 Database records

Every table in the Maven hash signature database contains records of vulnerable libraries found in Maven uses twelve columns. The other two tables in the npm hash signature database have records of vulnerable npm and JavaScript libraries use the same columns, respectively. The vendor column is absent in any of the npm tables.

- CVE - Contains the CVE ID used in the NVD as varchar. This field is mandatory.
- Name - This mandatory field contains the name of the package/library.
- Vendor - Only used for libraries from Maven, this mandatory field contains the vendor/manufacturer of the library as varchar. The JavaScript and npm hash database does not have this column at all.
- Hash - This field stores a hash in a hexadecimal format as varchar.
  Maven uses a 160-bit SHA1 hash that were directly extracted from the Maven Central Index or downloaded from its repository. Packages from npm instead use a custom algorithm with BLAKE2B-512 described in section 4.4 to create the stored 512-bit hashes. For regular JavaScript files their own hash signature in BLAKE2B-512 are stored here.
- New CVSS (3.x) score - This optional field stores the CVSS 3.x score. Note that older NVD entries do not use the CVSS 3.x because it did not exist at that time and uses CVSS 2.0 instead. Some preliminary NVD entries may not have any CVSS scores which may be added later at some point. If this field is empty then NULL is stored.
- Old CVSS (2.0) score - This optional field stores the CVSS 2.0 score. As (of now), every CVE entry that uses a CVSS score always has a CVSS 2.0 score. Again, some NVD entries do not have any CVSS scores.

- Disputed - The default SQL does not have any dedicated bool datatypes. Instead, if the entry is disputed which is marked as ** DISPUTED ** in the description of the NVD entry, then 1 is used, if not then NULL is used. This is currently redundant because a disputed vulnerability is still a vulnerability.
- Rejected - Rejected entries usually do not appear here because these have no CPEs. If one entry does get marked as rejected at some later point, then 1 is used, otherwise NULL is used. Rejected entries are ignored by the client hash checker, but no rejected entries that affect Java and JavaScript libraries have been found.
- ProbablyTP – The default value is one which indicates a true positive entry. If the value is zero then this entry is probably a false positive.
- CWE - The CWE ID is stored here. The exact descriptions must be provided from the client hash checker for space reasons, which has a map of every description associated with each CWE ID.
- Time - For internal research purposes, including this thesis, entries also have timestamps. The timestamp is based of the time of the database insertion in this hash signature database, not the age of the NVD entry.

As there is not a native bool type in SQL, integer values are used instead. A zero value represents false while any positive value represents true. This is seen in Fig. 5.2, with “rejected”, “disputed” and “probablyTP” which are supposed to use true or false.

Column Name	#	Data Type	Not Null
cve	1	varchar(255)	☑
name	2	varchar(255)	☑
vendor	3	varchar(255)	☑
version	4	varchar(255)	☑
hash	5	varchar(255)	☑
cvsScore	6	double	☐
oldScore	7	double	☐
rejected	8	int(11)	☐
disputed	9	int(11)	☐
probablyTP	10	int(11)	☐
cwe	11	varchar(255)	☐
time	12	datetime	☐

Fig. 5.2: Database scheme of a Maven table.

5.3 Library hash creator application

All downloaded content will be permanently stored for any possible later use, because any download from the internet is expensive in terms of used time. For the same reason is is strongly recommended not to delete any of the downloaded content. If the required file is already downloaded locally then that file does not need to be downloaded again.

As seen in Fig. 5.3, Two presets are available. The fast preset only checks for new NVD entries, whose last modification or addition is not older than 30 days. This is meant for to be used as a fast daily scan and requires up to 30 minutes. The slower full preset checks every single NVD entry which needs to the done at the first scan of this application and requires multiple days, up to two weeks depending on the quality of the internet connection.

Alternatively the maximum age of newly added and modified NVD entries can be given in months, with the default value of one month for fast daily scans. This number is the number of previous months of new and modified entries that will be taken into account. The age of the NVD entry is calculated from the current day minus the date in the lastModified date. In an example, if the value is one, then all NVD entries whose last modification happened in the last 30 days are evaluated, otherwise the NVD entry is skipped. Setting this value to 360 months or higher will not ignore any NVD entries because of the timestamp alone. The NVD only exists from 2005.

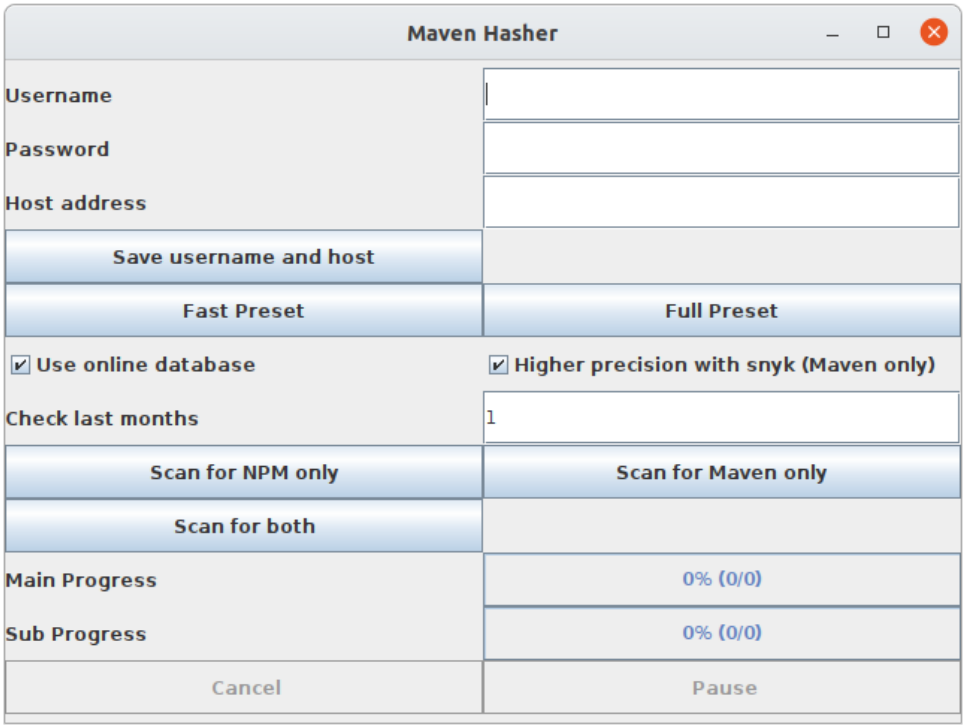


Fig. 5.3: GUI of the library hash creator.

A login form is provided to type any login credentials for the online MySQL/MariaDB database. The exact location of said database must also be provided by the user.

By default the online MySQL/MariaDB database is used and this option is recommended because any clients either use that database or download that database to be used locally. It can be changed to use the local SQLite database but no other user will be able to use the SQLite database because it is stored locally. The local SQLite database is mainly used for testing.

Pressing any of the three Scan buttons will start the scan, either only scanning the npm for any JavaScript libraries, only scanning any Maven repositories for any Java jar archives or both.

Two progress bars are display when the program is scanning. The main progress bar shows a broader range, in the case of scanning the number of years that were processed. The secondary progress bar shows a finer detail, in that case the number of NVD entries that were processed from the given yearly JSON datafeed. Both progress bars are only rough estimates and the only purpose is to show that any progress is happening. If the full scan is happening then the progress bars progress slowly. The title bar of the window also displays a small animation during any task to indicate that the program is running.

Exclusively in the GUI are the cancel button and the pause button. The latter pauses the current task and pressing that button again continues the task. This is achieved by adding a method at the beginning of every method which checks for a global value. Clicking the pause button sets the global boolean pause value to true. If said value is true then all running threads are running in an infinite loop until the pause value is set to false. In the infinite loop the threads sleep for a second before checking for the pause value.

The cancel button has a similar functionality except the threads are put to sleep indefinitely when the cancel value is set to true. When that happens, any running task threads are put to sleep, which takes up to five seconds, then the GUI is unlocked so the user can start any new tasks. This approach is used because Thraad.stop() and Thread.destroy() are deprecated and because these methods are inherently unsafe [77].

Finally the info button shows a popup with any relevant credits and copyright info. This button is also always accessible. During any task most of the GUI is blocked except for the pause and the cancel button, which are only accessible if there is any running task.

If any arguments are provided, then no GUI is used and the application runs inside the terminal windows. Two arguments are required.

The first argument must be any of the nine arguments below:

--fastMaven, --fastnpm, --fast
: Runs a fast scan, only checking the most recently created and modified NVD entries, up to one month old since the last modification

--fullMaven, --fullnpm, --full
: Runs a full scan, checking all NVD entries.

--partialMaven=[integer],--partialnpm=[integer],--partial=[integer]
: Runs a partial scan. Any NVD entries whose last modification are older than the number of months provided by the user are ignored.

The second argument used after one of the previous arguments.

--host=[URL]
: Uses the online MariaDB/MySQL database. The URL must be provided from the user, replace [URL] with the real URL. When the application has been started with this argument then the user prompted for their username and password.

--offline
: Uses the offline SQLite database. No host address, username and password are required.

5.4 Library Hash Checker application

The hash checker application uses the hash database created by the hash creator application to check any local libraries stored on the local computer.

Three input strings are expected from the user, the host address of the online library hash database, the username and the password from the user account.

For security reasons the the password cannot be saved locally and must be entered every time when the user requires to access the online library hash signature database. The other two input strings can be saved.

The graphical user interface (GUI) as seen in Fig. 5.4 is primarily used to input any required input strings. The Start button then starts the process of scanning the folder of the software project for any jar archives or JavaScript files if the folder does not have any special properties.

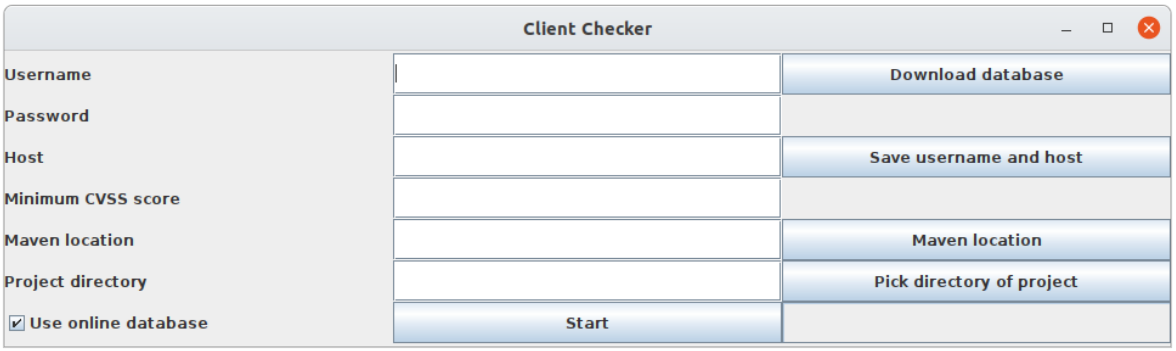


Fig. 5.4: GUI of the library hash checker

By default, it uses the online hash database. There is an option available that lets the user use the offline SQLite database, if it is already downloaded to the local computer.

The "Download database" button is used to download the current online hash database to a local SQLite database in the “resources” subdirectory. Once downloaded, this application will not require any Internet connectivity when the offline database is used, but updating the offline database still requires an Internet connection. Weekly updates to the offline database are recommended.

In a npm project which has a “node_modules” folder, the program detects the project as a npm project and scans the “node_modules” folder for any npm packages.

If the folder has a pom file which indicates a Maven project then the pom-file is scanned using Apache Maven. Using the “mvn --dependency-tree command” its redirected to a temporary text file, that text file briefly lists all used Java libraries from Maven and its dependencies. This list is then used to look up in the internal Maven repository stored on the local computer. As software projects may use multiple programming languages, it also scans for stray jar archives and JavaScript files inside the project directory.

While the GUI application is running, an animated bar is displayed which indicates that the program is running and has not crashed. The time duration of the scan is short and does not warrant a proper progression bar. All GUI interaction is disabled until the scan is over. At the end of the scan, a summary screen with the list of vulnerable libraries is shown. Using any arguments will not start the GUI. Any arguments in brackets are placeholders for the real file directory or an URL. These are the available argument combinations:

--dir={directory} --host={databaseHost}

The first parameter is the file directory of the software project where the library hash checker should scan for any vulnerable libraries. The second parameter is the host address of the library signature database. The username and password from an account is required to access the database. If no host argument is provided then the host address is prompted from the user.

--dir={directory} --offline

The first parameter is the file directory of the software project where the library hash checker should scan for any vulnerable libraries. The –offline parameter starts an offline scan instead, provided that there is an offline database on the local computer.

--download --host={databaseHost}

If the application is started with this only parameter then the online database is downloaded from that database. This also requires login credentials.

--help Shows the above description.

The single jar file of the library hash checker can also be used as an external library for other Java projects. In this case, two methods are available to use from the library hash checker.

- String checkDirectory(String directory, String user, String host, char[] password)
  Performs the same vulnerability check as the standalone application when a software project is scanned. The parameters require the user credentials for the online database, the location of the online database and the directory to be scanned for vulnerable libraries. A single verbose csv file containing the results is written to the hard drive. The results are also the return value of this method.

- boolean checkFile(String filename, String user, String host, char[] password)
  This method scans a single jar or js. It returns true if the library in that single file has confirmed security vulnerabilities, otherwise it returns false. User credentials to the online database are required too.

5.5 Difficulties

There were some severe difficulties during the development of the software.
At one point a whitelist of legitimate hash signatures of Java and JavaScript Libraries were planned. The client would take the already produced hash signature for each software library and check the hash signature against the local database that would store all known local hash signatures. This is not feasible because of time constraints of web crawling. At least five seconds per HTTP request are required to prevent being blocked from the web servers. If the HTTP request was denied, it would manifest as a HTTP error with the status code 403.

With over 16 million jar files across all Maven repositories and over one million libraries in npm, assuming that every library has ten versions on average which would result in ten million npm artifacts, 26 million files would need to be downloaded. Since only one artifact can be downloaded every five seconds, it would require a total of 130 million seconds or over 1504 days. In other words, over four years of continuous downloading is required and by that time the local database is already outdated because in these four years more projects and libraries were added since then.

In the case of Maven, the number of projects grow exponentially. Just from 2016 to 2018 Maven grew from 4 million stored artifacts to 11 million stored artifacts.
npm grows more linearly, from Jan 2017 to Jan 2019 it received 500,000 new packages, doubling in the number of total packages, not counting updates on existing packages [11].

Some optimizations were attempted to solve this problem, the existence of a large file containing all hashes of every artifact stored in the Central Maven Repository allowed to download one very large file instead. This is not without its problems, as the single binary file is too large to be handled by normal means. The exact process of extracting and handling the file manually is already described in 4.3.

Chapter 6

Related works

In this chapter multiple software with similar goals to the hash-based vulnerability checker are presented here. A direct comparison with the created software for the thesis must be possible or it uses the related work in one way or another in order to be included in this chapter. Only freely available software that rely on vulnerability databases are looked at in this chapter as well as two related theses that were created at the same faculty.

The presented software are the OWASP Dependency-Check in 6.1, Retire.js in 6.2, snyk.io in 6.3 and any other relevant related works in chapter 6.4. These sections also explain how the hash-based vulnerability checker is different to these related works. Neither of them use hash signatures to check for security vulnerabilities and these are all implemented in a single client application.

6.1 OWASP Dependency-Check

OWASP Dependency-Check by Long et al. [78] is a standalone Java program which attempts to detect vulnerable software libraries from a software project. The paper by Williams and Dabirsiaghi titled “Unfortunate Reality of Insecure Libraries” [79] was the main motivation of this software. Dependency-Check checks if there is a suitable Common Platform Enumeration (CPE) for any of the used libraries. This software tries to create these CPEs by analyzing the metadata from the libraries and using heuristics [80]. Then the program checks if that CPE has any associated NVD entries, which indicates that the associated software library is vulnerable. The standalone software writes a HTML file that contains the results when the scan is complete.

OWASP Dependency-Checks accesses multiple vendor security advisory lists to check if that library is listed in one of these. No hashing is involved at all.

This tool currently only detects vulnerable Java libraries, it supposedly supports .NET libraries but an error regarding a missing external dependency occurs when it tries to scan a .NET library, which is part of its own evaluation set. This part can be disabled. Aside from the standalone executable, it is also available as an Ant task, Maven plugin and Grade plugin.

6.2 Retire.js

Retire.js by Oftedal [81] is a software that can detect vulnerable JavaScript libraries and npm packages. It uses a fixed blacklist which can be viewed at their website and it does not use the NVD at all. That blacklist currently lists over 400 JavaScript and npm libraries.
It also exists as a browser plugin for Mozilla Firefox and Google Chrome which then runs in the background and can detect any vulnerable JavaScript libraries used by the website.

6.3 Snyk.io

Snyk.io[82] is a commercial tool that checks for vulnerabilities in projects. It maintains its own database containing vulnerable libraries in npm(Node.js), Maven(Java), NuGet(.NET), pip(Python), Composer(PHP), RubyGems (Ruby), Go and even Linux.

However, using it for commercial purposes is not free. There is a free version which has a limit of 200 uses per month for private projects, among other limitations, which for the purposes of this thesis does not suffice. It also requires a registration. The limit can be removed if the source code is made open source, but this is not an option for this closed project.

The library hash creator does sometimes use this vulnerability database to check a vulnerability and to increase its precision since the snyk.io database has the exact location for the vulnerable library in the Maven repository. Due the requirements however, it will only scan in that vulnerability database if it also has an entry in the NVD which is a drawback because sometimes the snyk.io database contains data about vulnerable libraries which are not in the NVD. No comparisons can be done to this software because it is not freely available.

6.4 Other related works

The software created as part of a thesis later known as the Code Clone Detector was initially implemented as an Eclipse plugin [31]. It monitored any installed software dependencies used by the software project. Similar to the OWASP Dependency-Check it tries to create a suitable CPE from the metadata of the used library which is then checked against the NVD, which is also downloaded locally so it can work offline after the initial download.

Being an Eclipse plugin it requires no user interaction after the installation, which is useful because the developer might forget to check their own libraries for vulnerabilities. The user can also create metadata for any libraries that do not contain metadata so that a suitable CPE can be created.

However, it used the now missing XML datafeeds from the NVD which were replaced by the JSON datafeeds. The Eclipse plugin also is not compatible with the newest version of Eclipse. The older versions of Eclipse on which this plugin does work is not compatible with any Java version that is newer than 8. Because of these, this software is currently unusable.

That previous vulnerability checker was improved to the current Code Clone Detector in a different thesis by Akba [83]. This is a standalone software rather than an Eclipse plugin and therefore it is not locked to a single platform. It does require slightly more user interaction. It is also updated to work with the new JSON datafeeds from the NVD. This software does not just detect security vulnerabilities using similar methods used in the previously mentioned master thesis but it can also perform code clone detection which can detect common security vulnerabilities just by scanning at poorly written code. This software supports JavaScript and Node.js.

From now on the after mentioned software created as part of the paper is simply referred to as the Code Clone Detector.

The Node Package Manager has its own blacklist of vulnerable libraries [76], which it refers to when running its own internal checks.

In contrast, the hash-based vulnerability checker is the only vulnerability checker that uses hash functions and it does not do any manual handpicking of vulnerabilities.

Chapter 7

Evaluation

The hash-based vulnerability checker is developed to detect any vulnerable libraries used in software projects that are written in Java or in JavaScript. This chapter evaluates its performance against other related works that were mentioned in the previous chapter. After describing the methodology in section 7.1 the retrieval performance evaluation is in section 7.2 and the time performance is evaluated in section 7.3.

7.1 Testing environment

For every evaluation, unless noted, a computer specified as in Table 7.1 was used:

Operating system	Windows 10 Pro 64-bit
Central processing unit	AMD Ryzen 5 1600 (6 cores, 3.2 GHz)
Random-access memory	16GB DDR4 2133
Video card	AMD Radeon RX 570 8GB
Solid-state drive (SSD)	Samsung SSD 850 EVO 500GB (SATA)
Internet connection	201.07 Mb/s downstream, 12.11 Mb/s upstream, 17ms latency

Table 7.1: Specification of evaluation computer

This software was developed on a Windows laptop that has different specifications and on the already mentioned Ubuntu desktop. Ubuntu is a Linux distribution, meaning that the software should run on most Linux distributions.

For cross-platform testing the desktop had both Windows and Ubuntu installed and the applications ran on both operating systems. The results were from tests run on Windows.

There was no macOS device available for development and for testing. The estimated effort required to run port this checker to macOS should be minimal, most cross-platform incompatibilities are caused by different external dependencies locations.

As every test was performed on a fast solid-state drive, the library hash creator may perform slower if it is run on a slower hard disk drive. The library hash checker which handles fewer files than the library hash creator should only have a small performance penalty from using a hard drive.

No GPU computing was used in any of the applications. The video card is only installed because the CPU does not have any integrated graphics processing unit which is required for a basic video output.

For development Java 11 was used. The applications were tested to run on Java 8.

7.2 Evaluation Terminology and Methodology

Before going to the details a few important methods from the Introduction Information Retrieval by Manning et al. [84] are explained. The results are represented in a table similar to table 7.2. This example will be used to describe the methods and it does not represent any actual results from any evaluation. The example set uses ten entries, four of them are supposed to be vulnerable while six of them are not vulnerable.

	Example
True positives	1
False negatives	3
False positives	2
True negatives	4
Precision	0.333
Recall	0.25
F1 score	0.286
Accuracy	0.5
Time	5s

Table 7.2: Example result table

In this context, true positives are vulnerable libraries that were correctly detected as vulnerable and true negatives are libraries that do not have any security vulnerabilities and were correctly detected as secure.

False positives are libraries that have no security vulnerabilities but were detected as vulnerable. False negatives are vulnerable libraries that were not detected by the vulnerability checker as vulnerable.

$$Precision = \frac{TP}{TP + FP}$$

Here it is important that precision is not accuracy because accuracy exists as a different metric. Accuracy will be discussed later. Precision in this context is the percentage of correctly detected true positives compared to all entries that were classified as positive, even false positives. A higher precision indicates that more true positives were detected and fewer false positives were detected. In the context of the vulnerability checker, it means there are more entries of libraries that are actually vulnerable and fewer to none entries of libraries that were flagged as vulnerable but actually are safe to use. It is a measurement of correctness. In this example, the precision is 1/3

$$Recall = \frac{TP}{TP + FN}$$

The recall is the percentage of all entries that were actually detected compared to the number all entries that were supposed to be detected. For this context, a better recall means that there is a lower number of vulnerable libraries that were not marked as vulnerable. A higher recall rate is important, because false negatives are not reported to the end user. Here in the example the recall is 1/4.

$$F1 = 2\frac{Precision * Recall}{Precision + Recall}$$

Instead of using the arithmetic mean between the precision and recall the F1 score is used. Compared to the arithmetic mean the F1 metric scores lower, even more so if the difference between the precision and the recall is large. In one extreme case, if either precision or recall is zero then the F1 score is always zero even if the other value is one. The F1 score "favors" smaller differences between the recall and the precision which are desirable if a balance between precision and recall is the goal. In this example the F1 score is 0.286, which is low.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

The accuracy[1] measures the overall accuracy of the vulnerability checker, the percentage of true entries compared to the total number of all entries. In this example the accuracy is 5/10.

This metric is not always the best choice, it is possible to have a very high accuracy, by having a very large number of true negatives, but a very low recall by having zero true positives and one false negative.

In general, the evaluation set should have be equally split between positive and negative entries and in an ideal test, the positive entries are classified as positive and the negative entries are classified as negative

The evaluation does also not take into account how severe the positive entries are. Using the previous case of the single false negative with many true negatives, that one false negative could be a critical security vulnerability that steals all data from the user, can easily inject other systems and render the infected systems useless once they finished their task, while any other positive entry is just a minor nuisance.

A vulnerable software library can have multiple vulnerabilities. For this purpose if the library has at least one vulnerability that is detected it is a true positive. If no security vulnerability was detected when the library has vulnerabilities then it is false negative.

One last benchmark is time performance measured in seconds, this is the time required by the vulnerability checker to perform its actual task, without any setup or updates. The full setup time is handled in section 7.4. For the hash-based vulnerability checker, only the time required for the library hash checker scan is used, as any filling of the hash signature database by the library hash creator is considered to be part of the setup.

7.3 Retrieval performance evaluation

Using the methodology from the previous section, the retrieval performance will be evaluated using four different evaluation sets. Comparisons to related works will be applied whenever possible.

In case for any Java libraries, either obtained from Maven or by other means, this hash-based vulnerability checker is compared to OWASP Dependency-Check and to the Code Clone Detector used in [84] and the previous vulnerability checker used in [31].

JavaScript files are also tested with the Retire.js vulnerability scanner. For any Node.js projects the internal npm audit command that checks for security vulnerabilities from the Node Package Manger also used.

The previous vulnerability checker from the master thesis [31] is denoted as the previous vulnerability checker. The Code Clone Detector from the bachelor thesis [83] is abbreviated as CCD. The last evaluation was run on March 17 2020, to verify the multiple evaluation runs over the two weeks before March 15 2020.

[1] TP = True Positive; FP = False Positive; TN = True Negative; FN = False Negative

7.3.1 Evaluation for Java libraries

At least one NVD entry (vulnerable) [CVE ID]	No CVE entries (secure)
Apache Commons-beanutils 1.8.3 [CVE-2014-0114]	Assertj Assertj-core 3.15.0
Apache Commons-collections 3.2.1 [CVE-2017-15708]	Clojure 1.10.1
Google Guava 11.0 [CVE-2018-10237]	Apache Commons-io 2.6
Apache Groovy-all 1.7.0 [CVE-2016-6814]	Apache Commons-lang3 3.9
Apache Httpclient 4.3.1 [CVE-2015-5262]	Apache Commons-logging 1.2
FasterXML Jackson-databind 2.9.0 [CVE-2019-17267]	Easymock 4.2
Jackson-mapper-asl 1.9.1 [CVE-2019-10172]	Google Gson 2.8.5
Jetbrains Kotlin-stdlib 1.3.0 [CVE-2019-10101]	H2database H2 1.4.200
Apache Log4j 1.2.9 [CVE-2019-17571]	Hamcrest Hamcrest-core 2.2
Apache Log4j-core 2.0.1 [CVE-2017-5645]	Hibernate-core 5.4.12.Final
QOS Logback-classic 1.0.0 [CVE-2017-5929]	Jaxb-api 2.3.1
QOS Logback-core 1.0.0 [CVE-2017-5929]	Joda-time 2.10.5
Apache Maven-plugin-api 3.0.4 [CVE-2013-0253]	Json 20190722
Oracle Mysql-connector-java 8.0.11 [CVE-2018-3258]	Junit 4.12
Squareup Okhttp3 3.0.1 [CVE-2016-2402]	ProjectLombok Lombok 1.18.12
Plexus-utils 2.0.0 [CVE-2017-1000487]	Mockito Mockito-core 3.3.0
Retrofit 2.0.0 [CVE-2018-1000850]	Scala-Lang Scala-library 2.13.1
Spring-core 3.0.0 [CVE-2011-2894]	JavaX Servlet-api 4.0.1
Spring-web 4.0.0 [CVE-2014-0225]	Slf4j-api 1.7.30
Spring-beans 2.5 [CVE-2010-1622]	Testng 7.1.0

Table 7.3: Test Java libraries

All 40 libraries from Table 7.3 are from the top 100 most depended upon libraries from any of the Maven repositories [85] as of March 01 2020. The 20 libraries that are known to have security vulnerabilities use older vulnerable versions, because newer versions have fixed these vulnerabilities. For the same reason the 20 secure libraries use the newest available version. The only exception is Junit 4.12 which is not the newest version but is the exact most popular library from Maven, as it is the default testing framework.

The justification for choosing popular libraries is that it is expected that a large number of developers will use there libraries at some point.

In order for a vulnerable library to be correctly classified as a true positive, the output logging file of the vulnerability checker must state that the vulnerable library is vulnerable.

In the context of the hash-based vulnerability checker this means that an entry of the vulnerable library, which contains the hash signature of the library, must exist in the hash signature database and the library hash checker produces a log file that states that this library is vulnerable.

	P. Recall	P. Comb.	P. Precision	OWASP D.C.	Hash checker
True positives	17	7	5	19	19
False negatives	3	13	15	1	1
False positives	9	2	2	0	0
True negatives	11	18	18	20	20
Precision	0.654	0.778	0.714	1.0	1.0
Recall	0.85	0.35	0.25	0.95	0.95
F1 score	0.739	0.483	0.37	0.974	0.974
Accuracy	0.7	0.625	0.575	0.975	0.975

Table 7.4: Results of the evaluation[1]

The single false negative as seen in Table 7.4 was maven-plugin-annotations 3.0.4. Examining this entry reveals a very specific configuration that requires this software to be used with another software named Maven Wagon 2.1. On its own, maven-plugin-annotations 3.0.4 is not a vulnerable library. Maven Wagon 2.1 was not installed on this machine during the evaluation, so this specific case was not applicable in this evaluation.

OWASP Dependency-Check had a different false negative. The library logback-classic 1.0.0 was not detected its vulnerability scanner. If logback-classic 1.0.1 in the evaluation set instead of version 1.0.0 then the newer but still vulnerable version is correctly detected. However, if both libraries use the same version number, even with different vulnerable versions, only one of them is detected. One hypothesis is that out of the CPEs that are created to match with the NVD, no duplicate CPEs can be created and only one CPE can be assigned to a vulnerable library at any given time.

The previous vulnerability checker cannot perform any code clone detection in a jar archive so it only scans if that library in the jar archive has suitable CPEs for the NVD which indicates that is has vulnerability. Any of its configurations that is not the recall configuration has severe issues at detecting libraries. Despite this, it has a worse recall and a worse precision than these two vulnerability checkers. For every evaluation, no code clone detection will be performed. The three false negatives are Apache Groovy-all 1.7.0, Maven-plugin-api 3.0.4 and Jackson-mapper-asl 1.9.1.

Overall the previous vulnerability checker [31] in the recall configuration performs worse than either of the other two vulnerability checkers and in the other two configurations it had a even worse retrieval performance.

[1] P. means previous vulnerability checker is from [31], but the results are new

7.3.2 Evaluation for npm packages

At least one CVE entry		No CVE entries
general-file-server 1.1.8	[CVE-2018-3724]	lodash 4.17.15
libnmap 0.4.10	[CVE-2018-16461]	chalk 3.0.0
waterline-sequel 0.5.0	[CVE-2016-10551]	request 2.88.2
mathjax 2.7.3	[CVE-2018-1999024]	express 4.17.1
angular-http-server 1.9.0	[CVE-2018-3713]	commander 4.1.1
reduce-css-calc 1.2.4	[CVE-2016-10548]	moment 2.24.0
serve-static 1.0.0	[CVE-2015-1164]	debug 4.1.1
mustache 2.2.0	[CVE-2015-8862]	prop-types 15.7.2
negotiator 0.6.0	[CVE-2016-10539]	react-dom 16.13.0
mqtt 2.0.0	[CVE-2016-10910]	async 3.2.0
node-jose 0.9.2	[CVE-2018-0114]	fs-extra 8.1.0
pidusage 1.1.4	[CVE-2017-1000220]	bluebird 3.7.2
decamelize 1.1.0	[CVE-2017-16023]	tslib 1.11.0
converse.js 2.0.4	[CVE-2017-5858]	axios 0.19.2
uri-js 2.1.1	[CVE-2017-16021]	uuid 7.0.1
		underscore 1.9.2
		vue 2.6.11
		classnames 2.2.6
		mkdirp 1.0.3
		react 16.13.0

Table 7.5: Test npm packages. Based off [84].

The evaluation set as seen in table 7.5 which is based of a similar evaluation set found used for the Code Clone Detector [84] contains 15 obscure npm packages that are known to have security vulnerabilities and 20 secure npm packages that are in the top 36 most depended upon pacakages. The rationale is the same as in section 7.3.1.

The same methodology from section 7.3.1 also applies here. Dependencies of libraries that are not in the above table are ignored in this evaluation.

Serve-static 0.1.2 from the original is replaced with serve-static 1.0.0 because version 0.1.2 is not available on npm and cannot be downloaded by any means. Version 1.0.0 of that package is still vulnerable. Otherwise the same evaluation set is used to allow a direct comparison to the Code Clone Detector [84] (p. 39).

	CCD Precis.	CCD Comb.	CCD Recall	Retire.js	npm	LHC
True positives	12	15	15	11	13	14
False negatives	3	0	0	4	2	1
False positives	0	4	4	0	0	0
True negatives	20	14	14	20	20	20
Precision	1.0	0.789	0.789	1.0	1.0	1.0
Recall	0.8	1.0	1.0	0.733	0.867	0.933
F1 score	0.889	0.882	0.882	0.846	0.929	0.965
Accuracy	0.914	0.886	0.886	0.886	0.943	0.971

Table 7.6: Results of evaluation for npm modules[1]

The only false negative by the library hash checker as seen in table 7.6 from this evaluation set is angular-http-server and it can be traced to the CPE. There is no version information from the CPE and it also does not use a version range. Without any of these information it is assumed that it has effectively an infinite version range, meaning every version of this library is vulnerable.

The hash-based vulnerability checker cannot handle infinite version ranges and therefore there are no hash signatures of that library in the hash signature database.

There is also another entry in the list of npm security advisories [86]. It recommends to upgrade angular-http-server to version 1.4.4 or later which suggests that older versions were vulnerable. The reference points at a git commit which fixes the security vulnerability and updates the version from 1.4.3 to 1.4.4 [87].

Again, the library hash checker has the highest precision, the highest F1 score and the highest accuracy out of all tested vulnerability checkers. It also has the second-highest recall.

The internal npm audit function also has this false negative and has another one. Mathjax was not detected as a vulnerable NPM package. The full ist of official npm security advisories which is the basis for npm audit does not mention mathjax [76].

The interesting aspect is that the internal security checks performed by npm can complete with the other solutions, especially if a high precision is desired. Since its security checks are already in npm and these checks always occur when the user installs new npm packages without further user interaction while all other tools require the user to start an external program.

The logs do show that lodash is vulnerable because waterline-sequel 0.5.0 uses an outdated version and vulnerable of lodash.

Retire.js also has these two false negatives for the same reasons. Two more false negatives occur with Retire.js, which are converse.js and mqtt. Its blacklist does not contain these two vulnerable libraries [81].

[1] LHC stands for library hash checker. CCD results were all taken from [84] (p. 39)

7.3.3 Evaluation for standalone JavaScript libraries

JavaScript package	Exists in npm	CVE (if it is vulnerable)
Jquery 2.0.0	Yes, as jquery	CVE-2019-11358 [up to 3.3.0]
Bootstrap.js 4.1.0	Yes, as bootstrap	CVE-2018-14042 [up to 4.1.1]
MomentJS 2.19.1	Yes, as moment	CVE-2017-18214 [up to 2.19.2]
Lodash 4.17.10	Yes as lodash	CVE-2019-10744 [up to 4.17.11]
Yahoo User Interface 3.9.0	Yes, as yui	CVE-2013-4942 [3.5.0 – 3.10.2]
React 16.4.0	Yes, as react	CVE-2018-6341 [16.0.0 – 16.4.1]
Prototype 1.7.2	No	none
EnquireJS 2.1.6	Yes, as enquire.js	none
MooTools 1.6.0	Yes, as mootools	none
Backbone.js 0.3.1	Yes, as backbone	CVE-2016-10537 [up to 0.3.3]
JsTimezoneDetect 1.0.7	Yes, as jstimezonedetect	none
Socket.IO 0.9.2	Yes, as socket.io	CVE-2017-16031 [up to 0.9.6]
KnockoutJS 3.0.0	Yes, as knockout	CVE-2019-14862 [up to 3.4.2]
Moxie 1.5.6	Yes, as mOxie	none
Lightning.js[1]	No	none
URI.js 1.19.2	Yes, as urijs	none

Table 7.7: Test standalone JavaScript libraries

Standalone JavaScript libraries consist of a single large JavaScript file. It can either be a full JavaScript file or a minified file. Minification removes any unnecessary data and comments while retaining full functionality [88]. As as side effect these files become unreadable by humans. Because there are multiple free minification tools that are available online [89][90]. Each of them have a different minification algorithm. Only official JavaScript files by the developer are supported in the hash-based vulnerability checker.

This evaluation only checks for official JavaScript files, but if a specific version is not properly detected then the issues are described in the evaluation. The list in Table 7.7 are 16 from the 20 most used JavaScript libraries [5]. Only the versions 3.5.0 to 3.18.1 of yui are in npm, older versions are not in npm.

Polyfill IO is a fully modular JavaScript library, the user can download as much functionality as required by the user, without having to download the full version. There is no official release for Polyfill IO. Preact.js does not exist as a standalone library despite being listed in the top 20 most used JavaScript libraries. Formvalidation.io is not a free library. It requires a one-time payment, so it cannot be included in this evaluation set. Script.aculo.us does not exist as a single JavaScript file and requires a more complex installation process.

These four libraries had to be removed from the planned evaluation set. The JavaScript evaluation set as seen in Table 7.7 consists of nine vulnerable and seven secure libraries.

As there is no centralized way to download JavaScript libraries without npm it is difficult to automatically find these JavaScript libraries.

[1] Lightning.js does not use version numbers, the recent library from March 03 2020 is used.

Urijs is not the be confused with the similarly named uri-js. Even if a JavaScript library exists in npm, it may not have older versions because some of these libraries predate npm.

	CCD Recall	CCD Combined	CCD Precision	Retire.js	Hash checker
True positives	4	2	0	4	7
False negatives	5	7	9	5	2
False positives	0	0	0	0	0
True negatives	7	7	7	7	7
Precision	1.0	1.0	N/A	1.0	1.0
Recall	0.444	0.222	0	0.444	0.778
F1 score	0.615	0.363	N/A	0.615	0.875
Accuracy	0.688	0.563	0.438	0.688	0.875

Table 7.8 Results of evaluation for standalone JavaScript libraries[1]

There are two false negatives from the library hash checker as seen in table 7.8, the first one is jquery 2.0.0, because npm does not have this version on jquery. If it is not saved on npm, then the library cannot be downloaded from npm. If a different vulnerable version of jquery was used in the set and is available in npm then the library hash checker would have detected it.

The other false negative is socket.io, the hash signature on the client is a different hash signature than the one saved on the hash signature database.

In general if the JavaScript library is minified using a different algorithm than the one used by the developers then the resulting minified JavaScript library has different hash signature than any of the official versions and the library hash checker will not detect the minified library as a vulnerable library.

In case of socket.io the npm package does not contain any JavaScript files that have the same hash signature as the standalone JavaScript library.

Retire.js performed worse than the library hash checker, having five false negatives. Three of them were not in the Retire.js JavaScript blacklist [81]. The other two, yui and react, are actually in the blacklist but for unknown reasons they were still not detected by Retire.js.

7.3.4 Conclusion of retrieval performance

As the evaluation was performed with three small evaluation sets, these only represent a small fraction of every software library available and therefore these evaluations do not apply to every available library.

False negatives from the hash-based vulnerability checker can be attributed to two faults. The first one are CPEs which contain wrong information, as in one example where the CPE only lists version 3.0 of Apache Commons HttpClient as vulnerable even though version 3.1 also has security vulnerabilities. Related to the first reason are infinite version ranges, because the library hash creator assumes that there are either a fixed set of single versions or a limited version range of vulnerable libraries. A fix was applied to the library hash creator in an attempt to solve that issue but it has no effect so far.

The other reason why the library hash checker may not detect a vulnerable library is because that library does not exist on Maven or npm. As the library hash creator relies on these repositories to obtain its libraries, it cannot download a library if it is there.

[1] *CCD is from [83], but not the results*

For a better evaluation a much larger set of libraries, at least one thousand, are required to obtain more accurate evaluation figures. In practice this is not feasible because these thousand libraries have to be manually evaluated by hand first to check if these are vulnerable. Downloading these libraries which also have dependencies also requires time. That hypothetical evaluation is beyond the scope of this bachelor thesis.

The hash-based vulnerability check does have the highest precision, the highest F1 score and the highest accuracy out of every vulnerability checker, but not the highest recall. Achieving the perfect recall seems to be impossible without losing some precision. This hash-based vulnerability checker is not recommended if a perfect recall is desired.

On the other hand, if the user wants a high precision then the other vulnerability checkers are only worse by a small margin. The integrated npm audit in particular may be good enough for most users when this is already built into the default Node.js package manager without any further user interaction required other than the initial installation of Node.js.

All other solutions require some user interaction, which mostly involve manually opening the application and manually scan the project directory unless said solutions are directly integrated into any integrated development environments (IDE).

7.4 Time performance evaluation

This section the time to perform the required actions are measured and evaluated. As the hash-based vulnerability checker is split into two applications, the time performance of each individual application is evaluated. The time for the average run and for the first run are measured.

7.4.1 Library hash checker and other vulnerability checkers

The time performance of the library hash checker depends whenever the offline database is used or an online connection was made to the online database. Connecting to the online database and performing queries requires more time because of the additional latency and bandwidth limitations from the online connection.

OWASP Dependency-Check also requires more time when it is run for the first time, because it has to download every datafeed from the NVD. This also applies if it was not run for more than a week [770]. If this application is run more frequently then it only has to download the “recent” and “modified” datafeeds, which are smaller than the yearly datafeed. If this application is run again shortly after the initial scan, then it will not check for datafeeds until the next day, shortening the total time for the scan in that instance.

Retire.js uses a blacklist and does not require NVD datafeeds to work. Its time performance is consistent, as its time performance for the first run is practically identical to subsequent runs. The Node Package Manager performs a check every time a new package is installed or updated by the user. After running an “install” command, if there are any packages with security vulnerabilities then npm warns the user with a text message. A more detailed security report can be seen after running “npm audit”. All listed times are in table 7.9.

The high amount of time required to check a npm project with the library hash checker can be explained by the larger number of files that the library hash checker has to process, in contrast to the other projects, in which each library consists of a single file.

Vuln. Checker (time in seconds)	1	2	3	4	5	min	max	mean	median
OWASP Dep.-Check initial (Java)	160	162	170	169	169	160	170	166	169
OWASP Dep.-Check repeat(Java)	5.83	5.97	6.03	5.97	5.97	5.83	6.03	5.954	5.97
LHC. offline – Java project	4.90	4.85	4.87	4.82	4.87	4.82	4.90	4.862	4.87
LHC. online – Java project	14.2	14.7	14.7	14.5	14.8	14.2	14.8	14.58	14.5
CCD [83] – Java project	83.1	84.3	84.1	82.8	82.9	82.8	84.1	83.44	83.1
Retire.js – npm packages	5.01	5.03	4.98	5.04	5.01	4.98	5.04	5.014	5.01
npm audit	1.49	1.49	1.43	1.54	1.46	1.43	1.54	1.482	1.49
LHC. offline – npm project	5.69	5.52	5.46	5.54	5.47	5.46	5.69	5.536	5.52
LHC. online – npm project	7.11	7.43	7.90	7.41	7.75	7.11	7.90	7.52	7.43
CCD [83] – npm project	155	153	152	152	150	150	155	152.4	152
Retire.js – Standalone JS	0.78	0.76	0.76	0.76	0.75	0.75	0.78	0.762	0.76
LHC. offline – Standalone JS	0.74	0.76	0.72	0.74	0.73	0.72	0.76	0.738	0.74
LHC. online – Standalone JS	1.14	1.17	1.21	1.24	1.18	1.14	1.24	1.188	1.18
CCD [83] – Standalone JS	22.0	22.1	21.8	22.3	22.2	21.8	22.3	22.08	22.1

Table 7.9: Time required for a single scan[1]

More hash signatures have to be created per library and more files have to be accessed per library, increasing the overall time required to process the entire npm project. For standalone JavaScript and Java libraries, the library hash checker, ignoring the setup time required in the library hash creator, performs slightly faster than the other vulnerability scanners when an offline database is used.

7.4.2 Library hash creator

A single daily update of the hash signature database requires, depending on the amount of new vulnerable libraries that have to be processed, eight minutes to 60 minutes. This single update fetches libraries from Maven and packages from npm simultaneously.

If the library hash creator is run for the first time, the initial filling of the database requires up to two to three weeks of continuous filling. The big reason for the slow performance are the access limits imposed by the websites this application requires to access. There is a limit on the number of requests this library hash creator can perform to these websites before it gets blocked. In practice, one access every five seconds will not get this application blocked.

As the result, every downloaded content will be saved to the local computer so in the event this application requires that content again, it first tries to use any locally saved resources if these exist before it has to download any online resources. The full installation requires at least 8GB of hard drive space.

The library hash creator also requires a relatively high amount of memory. The minimum amount of memory required for this application is 2GB of RAM, and the recommended amount of RAM is 4GB as at this point internal caching can be used, which reduces the overall time required for the total process.

[1] LHC stands for library hash checker, JS stand for JavaScript

The main reason for the high memory consumption is the usage of nested HashMaps which require a large amount of memory and the temporary loading of large files into the memory for processing.

7.4.3 Hash function time performance

As the BLAKE2B hash function claims to be faster than both MD5 and SHA-1, multiple hash functions were evaluated for their time performance. The tested files were a mixture of large files over 100MB, mostly archives, which have a total combined size of 3.43 GB.

Hash function	Bit size	Average time in seconds (lower is better)
MD5	**128**	**10677.2**
SHA-1	**160**	**15481.4**
SHA-256	256	21982.4
SHA-512	512	18450.6
SHA-512/256	256	17355.4
SHA3-256	256	17636.6
SHA3-384	384	22366.2
SHA3-512	512	30973
BLAKE2S-256	256	22947.6
BLAKE2B-512	**512**	**15811**
SKEIN-256-256	256	17531.2
SKEIN-512-512	512	15384.4
SKEIN-1024-1024	1024	16512

Table 7.10: Time performance of hash functions in Java

The slightly slower performance of BLAKE2B can be explained because this implementation was originally implemented in the C programming language and then was converted to the Java programming language as part of the Bouncy Castle library.

BLAKE2S is optimized for 32-bit systems while BLAKE2B, which is the default variant of BLAKE2 is optimized for 64-bit systems. In a 32-bit system BLAKE2S would perform faster than BLAKE2B [47]. Not included in in this comparison is BLAKE3 [91] which was just released on January 2020. The hash functions were compared in December 2019. No Java implementation exists for BLAKE3 and it is not recommended for anyone but security experts to do any implementations of any cryptographic functions.

A larger output hash size does not necessarily increase the time required as proven in Table 7.10. Using a 512 bit hash instead of a 128 bit hash lowers the chance of an accidental collision caused by the birthday paradox from very low to virtually zero. Using a larger hash size also slightly increases the space requirements of the hash-based vulnerability checker, but not enough to be problematic. If a large number of small files are used, the time required would be dependent on the number of I/O operations that the hard drive can perform and not by the hash function. This evaluation uses a solid-state drive to remove this factor as much as possible but the actual results are slower on a slower hard disk drive.

Chapter 8

Conclusion

8.1 Summary

The security vulnerability checker was developed to warn Java and JavaScript developers when they use vulnerable software libraries so these can be updated to newer secure versions.

It consists of two separate applications, a heavyweight library hash creator which is used to create a database containing hash signatures of vulnerable libraries, which the lightweight library hash checker uses to check for any vulnerable libraries.

The library hash creator first creates an internal CPE tree structure to map every single Maven library, using a generated vendor and product name string from the library's metadata as keys. Only the vendor and then the product name from the CPE are required afterwards to find the correct library during the scan of the NVD datafeeds. The hash signature from that library is downloaded and the hash alongside with the CVE ID, CVSS scores, product name, vendor, version and CWE are inserted to the hash signature database.

Once the hash signature database is completed, the library hash checker can scan any Java or JavaScript projects for any vulnerable software libraries. Every time a library a library is found its hash signature is created and checked against the hash signature database. If a match is found then that library is deemed to be vulnerable and the user is warned accordingly. The library hash checker is lightweight and can be easily integrated into other software.

The retrieval performance is slightly better compared to other vulnerability checkers, especially when a very high precision is required while also having a high recall. It does not have the highest recall compared to other solutions. The other vulnerability checkers can have a higher recall at the expense of having a lower precision.

The library hash checker does not have a significantly better time performance compared to other vulnerability checkers on its own. It already requires more time scanning npm packages compared to npm. Once the time required by the library hash creator is factored in, the hash-based vulnerability checker has a worse time performance than the other solutions.

Most of the work is still done by the library hash creator, when it is run the first time it requires a long time to fill up the hash database, up to a month of continuous filling. Subsequent operations after the first run are faster because only new entries need to be added per day.

In the case the user uses the standalone library hash checker, any positive results will be warned to the user in a summary window.

8.2 Outlook

Currently any entries containing vulnerable libraries must be listed in the NVD which is not the only vulnerability database in the world. There are other security vulnerability databases [62][92], some which have listed vulnerabilities that are not in the NVD. More vulnerability databases are used to create more hash signatures for the hash signature database so the clients using the library hash checker will not require additional installations.

An application could aggregate every available vulnerability database to a single large hash-based vulnerability database. That one combined vulnerability database might be the only database that the user needs.

The library hash creator could be expanded to support other programming languages, including Perl that has the Comprehensive Perl Archive Network (CPAN) repository [93] and Python which uses the Python Package Index (PyPI) [94]. No real performance penalty in a modern system with a multi-core CPU occurs because filling up multiple hash signature databases for each programming language can be done in parallel. A library hash creator for Python would never need to access any Maven repositories and vice-versa.

Some user experience improvements are possible. The hash signature database can contain the version number of a secure library. In the warning, that number is fetched to inform the user about newer more secure versions of the library. If the difference between the older and newer version number is great enough, the warning can also warn the user about any potential incompatibilities when the newer software library is used instead of the old vulnerable library.

One hypothetical idea with the approach of using hash signatures is to use them to create a community-created whitelist.

Every user automatically uploads every hash signature of their used software libraries to a newer larger database. A record from that database would include the name and version of the library which are associated with its hash signature. Said record also contains a tally which is incremented every time a user tries to upload that hash signature. If the record already exists, increment the tally by one instead.

If a user uploads a different hash signature than the commonly used one, then the user can be warned that their used library might be corrupted or infected by malware because the users hash signature does not match the common hash signature.

This does not just work on software libraries but on any other file on the computer, albeit there might be issues with intellectual property.

However, it does require a far larger amount of users before a consensus is reached on which library is the correct one, which is beyond the scope of this thesis. Obscure software libraries do not work with this idea because not enough developers use them. It will not help either if the agreed version already has security vulnerabilities to begin with. The idea is to detect outliers. It can also warn users if obscure libraries are used which are more likely to contain vulnerabilities.

The last sets of ideas are unrelated to hash signatures but can improve the rate of detecting vulnerable libraries by other means. By heuristic means similar how some anti-malware software work, if vulnerable code is detected in the project, the detection can occur even if no vulnerability database has this project listed as vulnerable.

The previous two ideas do increase the complexity of the library hash checker used by the client which goes against its main advantage, being lightweight.

8.3 Conclusion

The library hash checker has a slightly better retrieval performance and a measurably faster time performance if the library hash creator is ignored, but these advantages are small. Too small to justify anyone that already use existing vulnerability checkers to switch to the hash-based vulnerability checker.

Even for new users it is hard to justify the hash-based vulnerability checker because of the upkeep cost of the library hash creator. The initial first scan takes a long time, up to several weeks if the Internet connection is poor. To remedy this, the full installation of the library hash creator should be used as the first scan is already done on the full installation but that requires several gigabytes of hard drive space depending on how much has been scanned. Even then, weekly updates are required which may take up to an hour for the library hash creator.

The very idea of the hash-based vulnerability checker is also questionable when during the hash creation, it still requires the use of metadata to match the libraries with the CPEs, which is how some of the previous vulnerability checkers have worked. The hash seems like an extra step that is unnecessary. The user could have created suitable CPEs from their used libraries, then check the CPEs against the NVD datafeed to find out if their used libraries have security vulnerabilities.

The improvements should not be overlooked, at least for the client using the library hash checker. In the case for Java and non-npm JavaScript projects, the hash-based vulnerability checker does technically win over these previous solutions, but not by a large enough margin. This application can also be integrated into other applications.

As the client library hash checker is smaller, it easier to integrate the library hash checker to existing Java applications than having to integrate the full hash-based vulnerability checker.

In its current form the improvements are small, but this idea does warrant some further research. It is shown to be slightly better than the previous solutions while it also was not well optimized. Optimizing this application and incorporating some of the suggestions in section 8.2 could result to a better overall performance, even compared to the current hash-based vulnerability checker.

Appendix

A.1 Contents of the disc

The full contents of the disc are listed below

- The digital thesis is saved as a pdf file in its own directory “thesis”.
- The source files for the thesis, which includes the images and tables for this thesis, are in the directory “thesisSoruce”. The written thesis is also in this directory as an .odt file, which should be opened with LibreOffice 6.
- The minimum installation of the library hash checker is in the directory “library-hash-checker-min”
- The maximum installation of the library hash checker is in the in the compressed zip-archive “library-hash-checker-max”.
- The minimum installation of the library hash creator is in the directory “library-hash-creator-min”
- The maximum installation of the library hash checker is in the in the compressed zip-archive “library-hash-checker-max”.
- Source code for the library hash checker
- Source code for the library hash creator
- Javadoc documentation for the library hash checker
- Javadoc documentation for the library hash creator
- The evaluation sets are in the directory “EvaluationSets”.
  Use the provided vulnerability checkers to reconstruct the evaluation.
- Original evaluation results as .ods tables, best opened with LibreOffice 6.
- Maven [52], which is another dependency is already in the maximum installation of the library hash checker.
- Old evaluation for Java libraries
- Modified Code Clone Detector [83]. Some modifications by the author.
- Additional instructions.

Note that retire.js [81] is not included in the disc, it must be downloaded with “npm install -g retire.js”. This may require adminstrator/superuser rights.
It also does not include the large set of files used for the hash function evaluation.

7-Zip and OWASP Dependency-Check must be downloaded first.

Any listed software that has a citation was not created by the author of this thesis.

Bibliography

[1] ITU (2019). Fact Figures. Retrieved March 17, 2020 from https://www.itu.int/en/ITU-D/Statistics/Documents/facts/FactsFigures2019.pdf

[2] Worldometers (2020, March 16). World Population by Year. Retrieved March 16, 2020 from https://www.worldometers.info/world-population/world-population-by-year/

[3] Internet Live Stats (2020, March 16). Google Search Statistics – Internet Live Stats (2020). Retrieved March 16, 2020, from https://www.internetlivestats.com/one-second/#google-band

[4] Buildwith (2020, March 16). jQuery Usage Statistics. Retrieved March 16 2020, from https://trends.builtwith.com/javascript/jQuery.

[5] Builtwith (2020, March 16). JavaScript Library Usage Distribution in the Top 1 Million Sites. Retrieved March 17, 2020 from https://trends.builtwith.com/javascript/javascript-library, accessed Feb 25 2020

[6] Snyk (2020, March 16). jquery vulnerabilities. Retrieved March 16, 2020, from https://snyk.io/vuln/npm:jquery

[7] Snyk (2019, October 30). 84% of all websites are impacted by jQuery XSS vulnerabilities. Retrieved March 16, 2020, from https://snyk.io/blog/84-percent-of-all-websites-impacted-by-jquery-xss-vulnerabilities/

[8] W3Techs (2020, March 16). Usage Statistics and Market Share of jQuery for Websites, March 2020. Retrieved March 16, 2020, from https://w3techs.com/technologies/details/js-jquery

[9] Builtwith (2020, March 16). Bootstrap.js Usage Statistics. Retrieved March 16, 2020 from https://trends.builtwith.com/javascript/Bootstrap.js

[10] Snyk (2020, March 16). bootstrap vulnerabilities. Retrieved March 16, 2020 from https://snyk.io/vuln/npm:bootstrap

[11] Snyk (2019, June 4). npm passes the 1 millionth package milestone! What can we learn? Retrieved March 16, 2020 from https://snyk.io/blog/npm-passes-the-1-millionth-package-milestone-what-can-we-learn/

[12] OpenJS Foundation (2020, March 16). About node.js. Retrieved March 16, 2020 from https://nodejs.org/en/about/

[13] Snyk (2019, April 22) How much do we really know about how packages behave on the npm registry? Retrieved March 16, 2020 from https://snyk.io/blog/how-much-do-we-really-know-about-how-packages-behave-on-the-npm-registry/

[14] npm (2020, March 16). event-stream. Retrieved March 16, 2020 from https://www.npmjs.com/package/event-stream

[15] Snyk (2018, November 26). Malicious Code found in npm package event-stream downloaded 8 miilion times in the past 2.5 months. Retrieved March 16, 2020 from https://snyk.io/blog/malicious-code-found-in-npm-package-event-stream/

[16] Vaidya, R. K., De Carli, L., Davidson, D., & Rastogi, V. (2019). Security issues in language-based sofware ecosystems.

[17] npm (2020, March 16., eslint-scope. Retrieved March 16, 2020 from https://www.npmjs.com/package/eslint-scope

[18] Garrett, K., Ferreira, G., Jia, L., Sunshine, J., & Kästner, C. (2019, May). Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)* (pp. 13-16). IEEE.

[19] eslint (2018, July 12). Postmortem for Malicious Packages. Retrieved March 16, 2020 from https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes

[20] Voss, L. (2014, July 22). numeric precision matters: how npm download counts. Retrieved March, 2020 from https://blog.npmjs.org/post/92574016600/numeric-precision-matters-how-npm-download-counts

[21] Bloch, M., Blumberg, S., & Laartz, J. (2012). Delivering large-scale IT projects on time, on budget, and on value. *Harvard Business Review*, 2-7.

[22] Gammage, M. (2011). Why Your IT Project May Be Riskier Than You Think. *HARVARD BUSINESS REVIEW*, *89*(11), 22-22.

[23] Rohr, M. (2015, July 28). Sicherheit im Software-Entwicklungsprozess. Retrieved March 16, 2020 from https://www.informatik-aktuell.de/betrieb/sicherheit/sicherheit-im-software-entwicklungsprozess.html

[24] MITRE (2019, August 2). CVE – CVE and NVD Relationship. Retrieved March 16, 2020 from https://cve.mitre.org/about/cve_and_nvd_relationship.html

[25] MITRE (2017, December 15). Terminology. Retrieved March 16, 2020 from https://cve.mitre.org/about/terminology.html.

[26] NIST (2020, March 16). NVD – General. Retrieved March 16, 2020 from https://nvd.nist.gov/general

[27] MITRE (2019, November 6). CVE – Home. Retrieved March 16, 2020 from https://cve.mitre.org/about/index.html

[28] NIST (2020, March 16). NVD – CVE FAQs. Retrieved March 16, 2020 from https://nvd.nist.gov/general/FAQ-Sections/CVE-FAQs.

[29] NIST (2020, March 16). NVD – CPE. Retrieved March 16, 2020 from https://nvd.nist.gov/products/cpe

[30] Cheikes, B. A., Cheikes, B. A., Kent, K. A., & Waltermire, D. (2011). *Common platform enumeration: Naming specification version 2.3*. US Department of Commerce, National Institute of Standards and Technology.

[31] Viertel, F. P., Kortum, F., Wagner, L., & Schneider, K. (2018, October). Are third-party libraries secure? a software library checker for java. In *International Conference on Risks and Security of Internet and Systems* (pp. 18-34). Springer, Cham.

[32] MITRE (2020, February 10). CWE – About – CWE Overview. Retrieved March 16, 2020 from https://cwe.mitre.org/about/index.html

[33] MITRE (2020, February 20). CWE-448: Obsolete Feature in UI. Retrieved March 16, 2020 from https://cwe.mitre.org/data/definitions/448.html

[34] MITRE (2020, February 19). CWE – CWE-1006: Bad Coding Practices. Retrieved March 16, 2020 from https://cwe.mitre.org/data/definitions/1006.html

[35] MITRE (2013, February). Common Weakness Enmeration- CWE. Retrieved March 16, 2020 from http://makingsecuritymeasurable.mitre.org/docs/cwe-intro-handout.pdf

[36] NIST (2020, March 16). NVD – Categories. Retrieved March 16, 2020 from https://nvd.nist.gov/vuln/categories

[37] FIRST (2019). CVSS v3.1 Specification Document. Retrieved March 16 2020 from https://www.first.org/cvss/v3.1/specification-document

[38] Brigjaj, J. D. (2016, February 29). What's new in CVSS version 3. Retrieved March 16 2020 from https://www.acunetix.com/blog/articles/whats-new-in-cvss-version-3/

[39] NIST (2020, March 16). NVD – Vulnerability Metrics. Retrieved March16, 2020 from https://nvd.nist.gov/vuln-metrics/cvss

[40] RiskBasedSECURITY (2017, May 2). CVSS v3: When Every Vulnerability Appears To Be High Priority (2017). Retrieved March 16, 2020 from https://www.riskbasedsecurity.com/2017/05/02/cvssv3-when-every-vulnerability-appears-to-be-high-priority/

[41] BSI. (2019). *Kryptographische Verfahren: Empfehlungen und Schlüssellängen*. BSI TR-02102-1 v2019-1.

[42] Katz, J., & Lindell, Y. (2014). *Introduction to modern cryptography*. CRC press.

[43] Corn, P. et al. (2020, March 16). Birthday Problem. Retrieved March 16, 2020 from https://brilliant.org/wiki/birthday-paradox/

[44] Leurent, G., & Peyrin, T. (2019, May). From collisions to chosen-prefix collisions application to full SHA-1. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 527-555). Springer, Cham.

[45] Stevens M., Lenstra A., de Weger B. (2007) Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In: Naor M. (eds) Advances in Cryptology - EUROCRYPT 2007. EUROCRYPT 2007. Lecture Notes in Computer Science, vol 4515. Springer, Berlin, Heidelberg

[46] Apache (2020, March 16). MurmurHash3 (Apache Commons Codec 1.14 API). Retrieved March 16, 2020 from https://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/MurmurHash3.html

[47] Aumasson, J.-P., Neves, S., Wilcox-O'Hearn Z., Winnerlein C. (2017). BLAKE2 Retrieved March 16, 2020 from https://blake2.net/

[48] Chang, S. J., Perlner, R., Burr, W. E., Turan, M. S., Kelsey, J. M., Paul, S., & Bassham, L. E. (2012). Third-round report of the SHA-3 cryptographic hash algorithm competition. *NIST Interagency Report*, *7896*, 121.

[49] NIST (2012, October 02). NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition. Retrieved March 16, 2020 from https://www.nist.gov/news-events/news/2012/10/nist-selects-winner-secure-hash-algorithm-sha-3-competition

[50] Password-Hashiung (2019, April 25). Password Hashing Competition. Retrieved March 16, 2020 from https://password-hashing.net/

[51] Luykx, A., Mennink, B., & Neves, S. (2016). Security Analysis of BLAKE2's Modes of Operation. *IACR Transactions on Symmetric Cryptology*, *2016*(1), 158-176.

[52] Apache (2020, March 11). Maven – Introduction. Retrieved March 16, 2020 from https://maven.apache.org/what-is-maven.html

[53] Apache (2020, March 11). Maven – Introduction to Repositories. Retrieved March 16, 2020 from https://maven.apache.org/guides/introduction/introduction-to-repositories.html

[54] Apache (2020, March 11). Maven – POM Reference. Retrieved March 16, 2020 from https://maven.apache.org/pom.html

[55] Apache (2020, March 11). Maven – Guide to Naming Conventions. Retrieved March 16, 2020 from https://maven.apache.org/guides/mini/guide-naming-conventions.html

[56] npm (2020, March 16). About npm. Retrieved March 16, 2020 from https://docs.npmjs.com/about-npm/

[57] openjsan (2011, August 7). JSAN Home. Retrieved March 16, 2020 from http://www.openjsan.org/

[58] NIST (2020, March 16). NVD – Data Feeds. Retrieved March 16, 2020 from https://nvd.nist.gov/vuln/data-feeds

[59] Pavlov, I.: 7-Zip (2020, March 16). Retrieved March 16, 2020 from https://7-zip.org

[60] Snyk (2018): Zip Slip Vulnerability. Retrieved March 16, 2020 from https://snyk.io/research/zip-slip-vulnerability

[61] Apache (2020, March 11). Maven – Central Index. Retrieved March 16, 2020 from https://maven.apache.org/repository/central-index.html

[62] Snyk (2020, March 16). Vulnerability DB. Retrieved March 16, 2020 from https://snyk.io/vuln

[63] Maven (2020, March 16). Central Repository. Retrieved March 16, 2020 from https://repo1.maven.org/maven2/

[64] Mvnrepository (2020, March 16). Top Indexed Repositories. Retrieved March 16, 2020 from https://mvnrepository.com/repos

[65] Mvnrepository (2020, March 16). Jackson Databind. Retrieved March 16, 2020 from https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind

[66] Mvnrepository (2020, March 16). Central Repository. Retrieved March 16, 2020 from https://mvnrepository.com/repos/central

[67] Maven (2020, March 16). jackson-databind. Retrieved March 16, 2020 from https://repo1.maven.org/maven2/com/fasterxml/jackson/core/jackson-databind/

[68] Raemaekers, S., van Deursen, A., & Visser, J. (2017). Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, *129*, 140-158.

[69] Maven (2020, March 16). jackson-databind 2.9.0. Retrieved March 16, 2020 from https://repo1.maven.org/maven2/com/fasterxml/jackson/core/jackson-databind/2.9.0/

[70] npm (2020, March 16). all-the-package-names. Retrieved March 16, 2020 from https://www.npmjs.com/package/all-the-package-names.

[71] NIST (2019, December 30). NVD – CVE 2019-8806. Retrieved March 16, 2020 from https://nvd.nist.gov/vuln/detail/CVE-2019-8806

[72] npm (2020, March 17). xcode. Retrieved March 17, 2020 from https://www.npmjs.com/package/xcode

[73] Grace, M., Zhou, Y., Zhang, Q., Zou, S., & Jiang, X. (2012, June). Riskranker: scalable and accurate zero-day android malware detection. In Proceedings of the 10th international conference on Mobile systems, applications, and services (pp. 281-294).

[74] Stumm, V. (2016, August 25). How to crawl the Web politely with Scrapy. Retrieved March 17, 2020, form https://blog.scrapinghub.com/2016/08/25/how-to-crawl-the-web-politely-with-scrapy

[75] Zhao, C., & Sahni, S. (2019). String correction using the Damerau-Levenshtein distance. *BMC bioinformatics*, *20*(11), 277.

[76] npm (2020. March 17). Security advisories. Retrieved March 17, 2020 from https://www.npmjs.com/advisories?page=0&perPage=5000, accessed Mar 04 2020

[77] Oracle (2020. March 17). Java Thread Primitive Deprecation- Retrieved March 17, 2020 from https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

[78] Long, J. et al. (2020, March 17). OWASP Dependency-Check. Retrieved March 17, 2020 from https://owasp.org/www-project-dependency-check/

[79] Williams, J., & Dabirsiaghi, A. (2012). The unfortunate reality of insecure libraries. *Asp. Secur. Inc*, 1-26.

[80] Long, J. et al. How does the dependency-check work? https://jeremylong.github.io/DependencyCheck/general/internals.html, accessed Mar 08 2020

[81] Oftedal, E. (2020, March 17). Retire.js. Retrieved March 17, 2020 from https://retirejs.github.io/retire.js/

[82] Snyk (2020, March 17). Open Source Security. Retrieved March 17, 2020 from https://snyk.io/product/

[83] Akbariazirani, A. (2019, August 26). Advancement of Security Monitoring in JIRA with JavaScript Support.

[84] Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge university press.

[85] Mvnrepository (2020, March 17). Top Projects at Maven Repository. Retrieved March 17, 2020 from https://mvnrepository.com/popula

[86] npm (2018, May 16). Path Traversal – angular-http-server. Retrieved March 17, 2020 from https://www.npmjs.com/advisories/656

[87] Hampton S. (2019, May 11). Security fix. Retrieved March 17, 2020 from https://github.com/simonh1000/angular-http-server/commit/8bafc9577161469f5dea01e0b98ea9c525d063e9

[88] Google (2018, December 14).Minify Resources (HTML, CSS and JavaScript). Retrieved March 17, 2020 from https://developers.google.com/speed/docs/insights/MinifyResources,

[89] Uglifyjs (2020, March 17). JS Minify and Beautify - Online. Retrieved March 17, 2020 from https://www.uglifyjs.net/

[90] Mullie M. (2020, March 17). Minfy – JavaScript and CSS minifier. Retrieved March 17, 2020 from https://www.minifier.org

[91] O’ Connor J. et al (2020, March 17). BLAKE3. Retrieved March 17, 2020 from https://github.com/BLAKE3-team/BLAKE3

[92] JPCERT (2020, March 12). Japan Vulnerability Notes. Retrieved March 17, 2020 from https://jvn.jp/en

[93] CPAN (2020, March 17). The Comprehensive Perl Archive Network – www.cpan.org Retrieved March 17, 2020 from https://www.cpan.org/

[94] Python Software Foundation (2020, March 17). The Python Package Index Retrieved March 17, 2020 from https://pypi.org/

[95] The Eclipse Foundation (2020, March 18). Eclipse. Retrieved March 18, 2020 from https://www.eclipse.org/