KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Technology

First cycle, 15 credits

All quiet on the front-end?

Scanning websites for known dependency vulnerabilities

[redacted]

ISAK NYBERG

Stockholm, Sweden, 2023

All quiet on the front-end?

Scanning websites for known dependency vulnerabilities

[redacted]

ISAK NYBERG

Degree Programme in Information and Communication Technology
Date: September 7, 2023

Supervisor: [redacted]
Examiner: [redacted]
School of Electrical Engineering and Computer Science
Host company: [redacted]
Swedish title: På webbfronten intet nytt
Swedish subtitle: Skanning av webbsidor efter kända sårbarheter i beroenden

Abstract

As web applications become increasingly complex, the use of third-party libraries becomes increasingly prevalent. Dependence on third-party libraries can put the user of these web applications at risk if vulnerabilities exist in the dependent-upon library. From an IT operations perspective, gaining an overview of the dependencies a website uses– and any eventual vulnerabilities– is difficult. The front-end programming language JavaScript has a vast ecosystem of libraries and is not immune to this issue, coining terms such as *dependency hell*. Corporations and agencies that work with these ecosystems have substantial risks of being exposed to cyber-attacks if they have security vulnerabilities in their dependencies. As a result, there is a demand for tools that give insight and can detect these vulnerabilities before they are exploited by an adversary. The current academic research has not yet explored non-intrusive methods by which libraries that contain vulnerabilities can be discovered and addressed ahead of time. In this thesis, we research the current solutions and the methods used for discovering dependencies and vulnerabilities from a front-end client perspective. We develop a product that discovers dependencies by searching for license information in several places using different heuristics to discover library names and corresponding version numbers. We create an automated vulnerability scanner whose database and execution does not leave the host machine, in order to prevent third parties from accessing information about potential vulnerabilities. Testing our product together with the other discovered solutions on 14 websites, we show how our choices of methods and implementation are, in some metrics, superior to the solutions researched. Our implementation detected on average 23% more dependencies than the baseline. We also discuss which steps can be taken and which other methods can be used to improve the field further.

Keywords

Third-party libraries, Vulnerability scanning, Dependencies analysis, Web-security, Front-end dependencies

Sammanfattning

I samma takt som webbapplikationer ökar i komplexitet ökar också användningen av tredjepartsbibliotek. En webbapplikation beroende på tredjepartsbibliotek riskerar användarens säkerhet ifall sårbarheter existerar i biblioteken. För personer som arbetar med drift av webbservrar är det svårt att få en översikt av de beroenden och eventuella sårbarheter som används av ens servers webbapplikationer. Front-end programmeringsspråket JavaScript har ett enormt ekosystem av kodbibliotek som är hårt drabbat av just detta problem, till den grad att detta problem beträffande JavaScript har fått namnet *dependency hell*– beroendehelvetet. Som en direkt följd av detta problem riskerar organisationer och företag som arbetar med dessa ekosystem att bli utsatta för cyberattacker. Ett resultat av detta är ett stort behov av verktyg som ger insikt i applikationens komposition och som kan upptäcka eventuella sårbarheter. Existerande akademisk forskning har inte än fullt utforskat icke-intrusiva metoder för att upptäcka beroende som innehåller sårbarheter från ett användarperspektiv. I denna rapport undersöker vi befintliga lösningar och metoder som används för att, från ett användarperspektiv, hitta vilka beroende och sårbarheter en applikation innehåller. Vi utvecklade en produkt som kan hitta beroenden i en applikation genom att, med en heuristisk metod, härleda bibliotekens namn och tillhörande versionsnummer från licensinformation inuti applikationen tillika filnamn. I denna produkt har vi integrerat en sårbarhetsskanner med en lokal sårbarhetsdatabas vars exekvering sker helt lokalt, för att förhindra att information om sårbarheter hanteras av utomstående. Genom att testa vår produkt och existerande lösningarna mot 14 webbsidor visar vi hur våra val av metoder och vår implementation kan hitta 23% mer beroenden än de andra testade lösningarna. Vi diskuterar också vad som kan göras för vidareutveckling av produkten och forskning inom området.

Nyckelord

Tredjepartsbibliotek, Sårbarhetssökning, Beroendeanalys, Webbsäkerhet, Front-end beroenden

Zusammenfassung

Mit der zunehmenden Komplexität von Webanwendungen wird die Verwendung von Drittanbieter-Bibliotheken immer üblicher. Die Abhängigkeit von Drittanbieter-Bibliotheken setzt den Benutzer einem Risiko aus, wenn in der abhängigen Bibliothek Sicherheitsluken vorhanden sind. Aus der Sicht des IT-Betriebs ist es schwierig, einen Überblick über die Abhängigkeiten - und eventuelle Sicherheitsluken - einer Website zu erhalten.

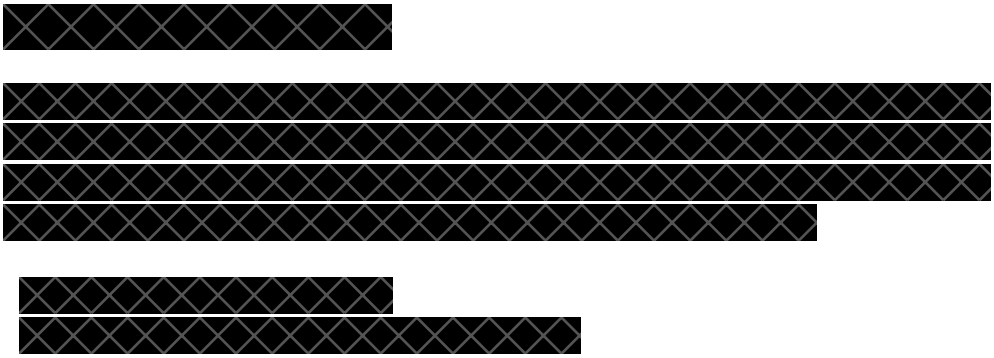
Die Programmiersprache JavaScript hat ein großes Bibliothekökosystem und ist nicht immun zu dieses Problem, was zu die Begriffe *Dependency Hell*- Abhängigkeitshölle geführt hat. Unternehmen, die mit JavaScript und diesenÖkosystemen arbeiten, haben ein erhebliches Risiko, Cyberangriffen ausgesetzt zu werden, wenn sie Sicherheitslücken in ihren Abhängigkeiten haben. Daher besteht ein Bedarf an Werkzeuge, die einen Einblick in diese Sicherheitsluke geben und sie aufdecken können, bevor sie von einem Angreifer ausgenutzt werden. Die aktuelle akademische Forschung hat noch keine nicht-intrusiven Methoden erforscht, mit denen Bibliotheken, die Sicherheitsluken enthalten, entdeckt und vorzeitig behoben werden können.

In dieser Arbeit untersuchen wir die aktuellen Lösungen und die Methoden, die für die Entdeckung von Abhängigkeiten und Sicherheitsluken aus der Sicht des Front-End-Benutzers verwendet werden. Wir entwickeln ein Produkt, das Abhängigkeiten entdeckt, indem es an mehreren Stellen nach Lizenzinformationen sucht und dabei verschiedene Heuristiken verwendet, um Bibliotheksnamen und entsprechende Versionsnummern zu ermitteln. Wir erstellen einen automatisierten Sicherheits-Scanner, dessen Datenbank und Ausführung den Host-Rechner nicht verlässt, um zu verhindern, dass Dritte auf Informationen über potenzielle Sicherheitsluken zugreifen können.

Indem wir unser Produkt zusammen mit den anderen entdeckten Lösungen auf 14 Websites testen, zeigen wir, dass die von uns gewählten Methoden und unsere Implementierung den untersuchten Lösungen in einigen Punkten überlegen sind. Unsere Implementierung entdeckte 23% mehr Abhängigkeiten mehr als die anderen getesteten Lösungen. Wir diskutieren auch, welche Schritte unternommen werden können und welche anderen Methoden verwendet werden können, um das Feld weiter zu verbessern.

Schlüsselwörter

Drittanbieter-Bibliotheken, Sicherheits-Scanning, Abhängigkeitsanalyse, Web-Sicherheit, Frontend-Abhängigkeiten



Contents

List of Figures

List of Tables

List of acronyms and abbreviations

API	application programmer interface
CDN	content delivery network
CSS	Cascading Style Sheets
CVE	common vulnerabilities and exposures
GAD	GitHub Advisory Database
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JSON	JavaScript object notation
MIT	Massachusetts Institute of Technology
NLP	natural language processing
NPM	Node Package Manager
NVD	National Vulnerability Database
OWASP	Open Web Application Security Project
SQL	Structured Query Language
URL	uniform resource locator
WASM	WebAssembly
XSS	cross-site scripting

Chapter 1

Introduction

In today's digital world, investment in cyber-security becomes increasingly essential as cyber-threats increase in frequency and sophistication. The reliance on digital solutions can be so extensive that a ransomware attack can entirely disrupt the operations of a large grocery store chain[1]. This threat is apparent to individuals, companies, and governments alike. As a result, necessary steps must be taken to protect information and systems against cyber-attacks from adversaries, whether malicious individuals or foreign governments. The leaking of personal information and disruption of services can have a cost on the order of millions of dollars, making proactive efforts to minimize the risk of breaches a wise investment[2].

Despite the significant investments made into cyber-security, the sheer number of attack vectors that exist and are utilized by attackers today makes it increasingly complex for defenders to get an overview of the scale and type of their environments' vulnerabilities. Additionally, as software gets increasingly complex, third-party libraries become more prevalent in many aspects of software development[3]. This helps developers by saving time and effort in developing certain functionalities that a third party has already developed. However, using third-party libraries creates dependencies that might lead to vulnerabilities in the developed product, as any vulnerability present in a third-part library or other library that a particular library depends on will propagate to the final product[4]. It is time and resource-consuming for developers to manually investigate their dependencies and find any potential vulnerability.

There are a number to underlying technologies in front-end applications. The most popular of these is the scripting language JavaScript which is, among its various use cases, commonly used in websites and web apps. However, this frequent use also makes it a common attack surface on the web. JavaScript

developers commonly use public packages and libraries written by third parties with the help of package managers like Node Package Manager (NPM). This product makes more than a million packages available to developers. Every package has the potential risk of containing vulnerabilities.

There exist several known vulnerabilities that are common in front-end JavaScript code. Cross-site scripting (XSS), for example, is an exploit that enables an attacker to send code to another end-user of the website, code that is then executed in the victim's browser. As many of these vulnerabilities have already been discovered and often patched, developers can mitigate or remove these vulnerabilities by simply updating or making slight alterations to the code. Updating dependencies would be a trivial solution were it not for the complex task for developers or IT personnel to find and verify the integrity of the website's dependencies[5]. Such a task consumes a lot of manpower and resources to do, especially since new vulnerabilities are found continuously, which calls for the need for dependencies to be updated. A previous paper has shown that the portion of users that update their software decays exponentially over time[6]. This paper also found that it takes slightly less than three and a half years for 95% of users to adopt a new Android update after a vulnerability was fixed. Websites in particular possess an inherent risk since they usually serve as an interface between an end-user and a back-end system. This means that a user with some technical knowledge can see both the code running locally on the website and information sent from their browser to the back-end, giving an attacker considerably more insight than with any compiled application. As a result, ensuring there are as few vulnerabilities and attack vectors as possible is particularly important. One of these vectors is the possibility of vulnerabilities arising from outdated dependencies in front-end web code. Previous studies have found that 37% of popular websites contain at least one dependency with a known vulnerability[7]. Two reasons for why this number is so high are firstly that the website administrators are not updating their dependencies versions where these vulnerabilities have been patched. Alternatively, its developers may have abandoned the dependency, and the vulnerability has never been addressed.

1.1 Problem

As web applications become more and more complex, so does the code that drives these applications. With the possibility of inherent vulnerability flaws in libraries, IT personnel must spend much of their time researching libraries used in applications if they want to create a secure web environment for their

company and its users. Organizations dealing with sensitive data often have restrictions on which products they use; for example, some organizations might not see it appropriate to use cloud-based, proprietary software when securing their web environments. For organizations that require a secure web environment and a good overview of front-end web dependencies, what methods may be used to create software that automatically performs the task of dependency checking a website from a front-end client perspective and, with the methods discovered, how does this product compare to others on the market?

1.1.1 Original problem and definition

Constructing an automated front-end vulnerability scanner of JavaScript library dependencies used in websites, using only a local database of known vulnerabilities.

1.1.2 Research question

What methods can be used and what methods are used when detecting the presence of libraries in front-end code? How can these be checked for known vulnerabilities and, how can these methods be implemented effectively with a local database?

1.2 Purpose

The project also aims to increase knowledge of the methods involved and practices used when scanning for dependencies and vulnerabilities in source code as well as– with a comparison between current solutions on the market and our local database scanner– explain the key advantages and disadvantages with these methods and practices.

We hope that our project increases knowledge of cyber-security, which will benefit not only the recipients of this knowledge but contributes to the recipient's consideration of vulnerabilities arising from dependencies on third-party libraries. Furthermore, this consideration will lead to more secure web applications, resulting in fewer security breaches and leakage of sensitive data. This would benefit all users of such applications.

1.3 Ethics & sustainability

This report's information may be used for malicious and benign purposes. Malicious use of the information in this thesis may increase security breaches, possibly leading to an increase in leakage of sensible data, damaging not only the affected product owners but also the users. However, the information from this report can serve a benign purpose by creating methods by which developers and clients can verify the integrity of their own code and dependencies to prevent security breaches before they happen.

From a sustainability perspective, these breaches can cause both social and economic harm to the individuals affected. Social harm could come from data leaks that reveal individuals marital status or other protected information, while economic harm can come from monetary loss due to cyber-theft. There is also a distinct positive effect on economical sustainability, as the product may prevent data breaches or exploits which are not only costly for the users but for the organization hosting the website. We see no ecological benefit of our work, on the contrary, the only clear impact on ecology is the energy consumption of the product and the demand this product may induce on hardware, whose production is both costly, energy consuming and has a distinct negative ecological impact [8].

1.4 Goals

This project aims to give developers a tool that will automate the process of finding vulnerabilities in their front-end dependencies without the need for third-party tools. This tool comes in the form of a vulnerability scanner that runs on-premise with a local database. The research process is documented, and the steps taken are discussed in this thesis. The project has been divided into the following three sub-goals:

1. Examine current academic and commercial solutions for vulnerability scanning in the areas selected by the product owner

2. Create a working vulnerability scanner that satisfies the requirements of the product owner

3. Deliver a report on current scanning methods for dependencies and vulnerabilities, as well as a comparison of our product to current commercial solutions

1.5 Research Methodology

First, a literature study on the subject was conducted. We aimed to discover the current solutions and methods for detecting dependencies in front-end code and to discover what databases are used, what information they contain and how, and from what sources they are built. We chose three existing solutions based on their popularity and source code availability and researched how they function and satisfy our requirement specification.

The project initially collected requirements for the vulnerability scanner. These requirements have measurable outcomes and were used evaluated continuously to evaluate the product iteratively.

These requirements include the accuracy and precision of the vulnerability scanner, the criterion found in the literature study, and the requirements from stakeholders. In addition, existing solutions found in the pre-study are evaluated against the same criteria. Finally, this evaluation is compared and acts as the final results and conclusion of the paper to establish which solution is ultimately the better-suited one.

1.6 Delimitations

The thesis only concerns itself with vulnerabilities in websites. Methods for finding these in other interfaces, such as public application programmer interface (API)s, servers, and other protocols, are not mentioned in this thesis. Additionally, the thesis does not discuss processes to find new vulnerabilities. Instead, it focuses on discovering known vulnerabilities that are still in use by a website. There is also some focus on the surrounding system that creates this functionality, such as databases to index the vulnerabilities, how these vulnerabilities are collected and how this information have been accessed in the first place.

1.7 Structure of the thesis

The thesis is divided into multiple chapters. Chapter 2 discusses the current state of web security, the current solutions, the current problems and threats, and what possible approaches already exist to combat them. Chapter 3 explains how we approach the project and what research techniques are used for both the search for information and the development. Chapter 4 focuses on the implementation, explaining what our final product looks like and how it can

be reproduced. Chapter 5 presents the product performance evaluation related to the requirement specification and comparisons to existing solutions found in the pre-study. This Chapter also includes a discussion. Finally, in Chapter 6, we conclude the thesis results and discuss future works on the subject.

Chapter 2

Background

This chapter focuses on the current state of computer security regarding webpage security. This includes how websites work, how dependencies are loaded and possible ways to find out the dependency version and potential vulnerabilities.

2.1 JavaScript & Hypertext markup language

HyperText Markup Language (HTML) is a simple markup language used to create hypertext documents, text documents that, by hyperlinks, are connected to other texts. *JavaScript* is a scripting language designed for and mainly used in web browsers to add dynamic functionality to websites by responding to user-initiated events and changing a website's content[9].

2.2 Dependencies

When developing software, it is often beneficial to reuse code already produced by another party. Already developed code packages or libraries consist of either open source code that is included directly in one's source code or is included at runtime dynamically. Software using a third-party library is, in most cases, dependent on it for full functionality.

2.2.1 JavaScript Libraries

JavaScript is an interpreted language and has, like any other programming language, the ability to reuse code in the form of libraries. However, JavaScript has no official repository of libraries. However, services such as NPM exist–

a repository with over a million packages. When developing a JavaScript application, developers can use services such as NPM to automatically download and resolve dependencies and package the complete application, including dependencies.

2.2.2 Loading dependencies

A website that relies on dependencies such as code libraries, scripts, or style sheets can load these dependencies from external sources in several ways. This can be done either through tags in the HTML or CSS or with calls in the JavaScript code executed in the browser. This allows developers to save time by importing work from other places. Once the browser has fetched these files, they can be parsed according to their file type and include the web page's functionality. The code that is retrived is usually loaded in plain text. This fetching can be done when the page initially loads or when the pages are loaded asynchronously, and the dependencies may load after the web page has been used for some time, as some features are only loaded if the user decides to use or click on them. One example could be images that only are loaded after the user clicks on a button to view the image. This facilitates a faster initial loading of the web page to improve the user experience and is an essential part of a website's functions from a client perspective. However, since there are numerous ways to load dependencies, each of these would need to be accounted for when a web scraper or crawler is used to find them.

The most common way of including a library is `<script>` HTML tag, that uses an argument to point to a file hosted either on the same server or on a library hosted remotely. When these files are hosted externally, they are commonly hosted with a content delivery network (CDN), that collects many different libraries; this allows libraries to be cached by the browser if the same library is used across many web pages. It also allows the client to load the library from a geographically closer server to the user to speed up and improve the user experience. In this scenario, it is relatively easy to find what library is sourced and what is specific to the website.

It is common for these files to have gone through a process called *minification* that attempts to cut the file size as much as possible without changing the execution behaviour. For example, removing white spaces or making variable names shorter.

2.3 Vulnerabilities

Websites work by executing code and files that the web browser fetches. This includes loading text and images and executing code that allows for animations and more advanced website features. This is, of course, useful for developers since it allows them to improve the experience for the end-user; however, unfortunately, this can also be exploited by adversaries that misuse this liberal treatment of websites. Instead, harmful code can be sent to and executed by a web browser to extract information or execute nefarious code on an unknowing individual's machine. While the service provider cannot ensure the safety of the client's machine, they can and should address vulnerabilities in their code that could expose the server or client to vulnerabilities.

One common vulnerability is cross-site scripting (XSS), which arises from insufficient sanitation in website inputs, allowing an adversary to inject their own JavaScript into a browser of a targeted individual or organization. In addition, XSS can be instigated when a website allows external users to publish content, but also through more innocuous ways, which may not be apparent to the programmer or website user.

2.3.1 Vulnerability databases

There exists several public databases contain information on existing vulnerabilities in libraries and software. The most popular are Mitre CVE, CVE details and the US NVD [10][11][12]. A vulnerability database provides information on particular vulnerabilities, vulnerability type, severity, what software they affect and what version range(s) are affected. Different databases provide different interfaces. In the above examples, Mitre provides a web search and a downloadable file, CVE details provide only a web-search interface, and NVD provides an API.

2.4 Scraping & Crawling

When an individual views a website, it is interpreted and displayed by a web browser. For a computer program however, displaying the website is unnecessary; it can simply read the plain HTML use that code to carry out automated tasks. These tasks can involve collecting information, saving images or any other task that would be inefficient for a human to undertake. A crawler is a sub-section of scrapers that also have the ability to traverse between web pages through hyperlinks and other means. This is useful for

scraping multiple similar web pages connected to each other. However, the terms are somewhat interchangeable. Search engines such as Google rely on crawling to find and index websites, which are then made searchable by the search engine; it is also common practice when automating data collection to crawl websites and scrape for particular data[13]. In addition, modern websites often use CDNs or host content on different servers and in different documents than in the requested HTML file. Therefore, web clients often download additional content after the initial request is made, which is done when the HTML is interpreted. Crawlers and scrapers, however, do not interpret the whole HTML, they only seek links or particular data.

Different techniques may be used depending on the type of data the scraping or crawling application is seeking. For a simple website with script segments embedded in the HTML, a simple HyperText Transfer Protocol (HTTP) GET request is necessary as all the source is present in the request body. More complex websites hide sources inside the JavaScript behind events triggered by the code, for example, a website might load JavaScript which in turn loads more JavaScript code via a HTTP request. Here there are several approaches, one of which is to try and parse the source and find different code bases by analyzing, for example, string literals, trying to find uniform resource locator (URL)s. Another approach is to use web browser drivers and programming interfaces to standard web browsers such as Google Chrome that interprets web code and outputs the resulting code[14].

One crucial aspect to be aware of is that it may be difficult for a website to tell real users apart from scrapers and crawlers, as they behave similarly from a server perspective. However, since scrapers and crawlers can be used nefariously, a website owner can take steps to limit the scraper's activity. Aside from aggressive methods of denying service to suspicious users, websites commonly refer to the so-called `robots.txt` file. The following is an extract from an anonymized website's `robots.txt`. Some paths have been changed to retain anonymity.

```
#
# well-known resource robots.txt from 17.392
#
User-agent: *
Disallow: /form/
Disallow: /public/
Disallow: /cm/
Disallow: /info/
Disallow: /work/
Disallow: /xyz/
...
Disallow: /about/vacancies/open/
Disallow: /about/vacancies/open?
Disallow: /en/about/vacancies/open/
Disallow: /en/about/vacancies/open?
Disallow: /blogs/tags
Disallow: /vacancies/internal/

Sitemap: https://[REDACTED]/sitemap.xml
```

The file specifies which areas of a web page are not allowed to be scraped. For example, the `robots.txt` above advises that scrapers and crawlers may not access the information about vacancies at the organization. However, beyond acting as a moral compass, the file does nothing to actually prevent a determined adversary from scraping the disallowed pages.

2.5 Existing solutions for detecting dependencies

Our study found four relevant solutions currently available for detecting a website's dependencies from a client perspective. Three of these are browser plug-ins, *Library Detector for Chrome*, *Retire.js*, and *OWASP pen test*. As plug-ins, these solutions extend the functionality of a web browser, using information gained from interfacing with the browser to detect dependencies on the website currently loaded by the browser. Another result of being implemented is that these solutions can not be easily automated as they require a user to operate the web browser and activate the plug-in. Furthermore, they are dependent on support from the browser to function

correctly. All these solutions use a small custom database containing known vulnerabilities and, differing from databases mentioned above, including methods of interfacing with the JavaScript code for detecting libraries from a client perspective[15][16].

Aside from the browser extensions, more extensive and more sophisticated commercial solutions exist, for example, Snyk[17], which provides other services and vulnerability scanners. These commercial solutions are also much broader in their scopes and do not stop at addressing dependency vulnerability, but they also detect flaws and code vulnerable to attacks in the user code itself. Moreover, these solutions usually have direct access to development environments, either via code repository systems such as GitHub or via a local installation, where a package manager manages open-source vulnerabilities.

There exist prevalent vulnerability scanning solutions that scan websites from a client perspective, such as OWASP Zed Attack Proxy[18] and Skipfish[19]. However, these solutions provide a more extensive set of tools, including active penetration, such as injecting malicious payload via POST requests or open ports. This is something that can prove disruptive to the website and something that our external stakeholders strictly prohibit.

2.5.1 Trust and Demand for Solutions

When using any proprietary software, one can never be certain of exactly what the software executes. This means that the user needs to trust the software providers not to be malicious. This is not a problem for most entities, and reputable third-party software is reasonable to use. However, for companies and governmental organizations that deal with sensitive information and fear being targets of cyber-attacks, using a third party for security analysis can be ruled out. Thus, these tools need to be either open source or developed on-premises.

We have found no open source, non-disruptive, automated alternative with a completely offline database of known libraries and their vulnerabilities.

Chapter 3

Methods

3.1 Research process

This project consists of a pre-study phase and an implementation phase. Knowledge gained in the pre-study is used to develop a process for implementing the product. First, we gather data on the theory behind and the methods used within the subject. Following this is a development process that works iteratively.

3.2 Pre-Study

The goal of the pre-study was to discover current techniques and methods that are proposed in academic papers or present in current solutions. Academic papers on the subject were found by searching Google Scholar with various combinations of the keywords "Vulnerability", "Dependency", "JavaScript", and "Detection".

Current solutions were found by making Google queries with the keywords "Library detection", "Vulnerability detector", "plug-in", and "application", as well as looking at references in articles on the subject. Current solutions were deemed acceptable if the provided product was free and that the solution provided a report on a website's JavaScript library dependencies, and if the source code was publicly available. Finding corresponding vulnerabilities where not deemed necessary, as finding reported vulnerabilities to a library with a known version number is trivial compared to finding the dependency and its version.

3.3 Requirement Specifications

We are interested in developing a solution that adheres to the following requirements, each of which is addressed in the subsequent sections of the report.

- The product should have an *interface* that is makes it possible to:
  - Initialize a scan of a website
  - Show the progress and results of an ongoing scan
  - Produce a report or a summary of the findings of the scan
  - Schedule a scan of a website
- The product should have a *scanner* that is able to:
  - Load the dependencies of a given website
  - Be able to identify dependency names and versions of a dependency with high accuracy
  - Efficiently traverse a website and links on that website
  - Be aware of robots.txt files
- Database: The database should be able to:
  - Run on a Linux server
  - Update the list of stored vulnerabilities
  - Efficiently index a dependency along with its versions number

3.4 Evaluation

The product is evaluated against existing products, namely Retire.js[15], Library Detector for Chrome[16] and OWASP pen-tester plug-in[18]. A collection of websites was selected on the criteria of fitting the environment of the external stakeholder, and their dependencies were manually discovered. Manual discovery entails thoroughly searching the website with a web browser and looking for license information and library signatures. The set of dependencies manually discovered is by no means an extensive set of all dependencies on the webpage, but gives a lower bound of how many dependencies that the webpage has. Finally, the products are used on the website, and their results are evaluated on the following points:

- How many of the known dependencies the product finds.
- Proportion of found dependencies to found dependencies manually
- How many false positives that are reported.
- Proportion of false positives to true positives.

Further, we present which methods of detecting dependencies that are most effective by showing how many dependencies each method finds. We also present the number of total vulnerabilities found on all the sites.

3.5 Documentation

The methods and strategies used during development and in the final implementation will be discussed in section 4. Note that the implementation will not be published due to security concerns from our external stakeholder.

Chapter 4

Design

The product is consists of several components. This chapter discusses different methods and parts of our implementation and their overall expected success. We further explain the challenges that appeared and how our implementations addresses them.

4.1 Web scarping & crawling

To determine the libraries used in front-end web code, we must first be able to gather all the source code used by a web browser when accessing a website. To traverse the whole website, we developed a scraper that looks exclusively at HTML `a`-elements and extracts their `href`-attribute. If this `href`-property is a URL that does not direct to an external website, this URL is added to a queue.

4.2 The Crawler

The crawler itself takes the form of a link crawler. It is given a base URL as an argument that is added to the workload queue. It then traverses the web pages according to Figure 4.1

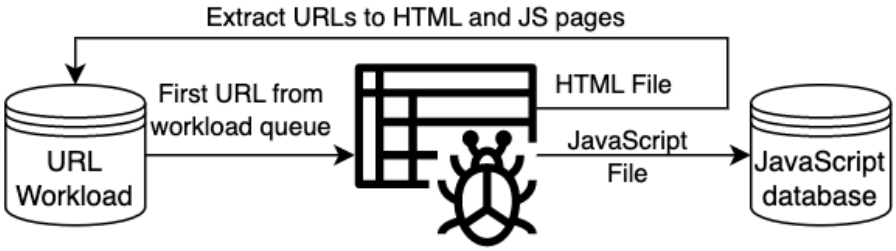


Figure 4.1: Protocol of link crawler

The *URL Workload* is a queue of URLs, initialized with the base URL given by the user. The first URL in the queue is given to the crawler that fetches the source code of the resource. If the resource is JavaScript file it will be added to the JavaScript database, and if the resource is a HTML page it will be scanned for links to HTML or JavaScript files, which will be added to the *URL Workload* queue [20][21].

On a modern website, URLs are represented in mainly two formats: absolute and relative. Absolute URLs contain the whole network location, for example

```
www.example.com/path?query=value#fragment
```

Relative URLs, however, only contain the path. For example,

```
/path?query=value#fragment
```

found on a website on www.example.com is interpreted by the browser as

```
www.example.com/path?query=value\#fragment
```

This needs to be accounted for when processing the URLs since the URLs may contain information that is relevant to the dependency that the resource contains.

4.3 Library detection

Given a specific JavaScript file, there are several ways to discover what libraries a website uses from the client's perspective. Since web browsers interpret the code from a website, there is no need for comments or any other metadata that declares the library name and version number. There is no guaranteed way of determining if the code is from a known library. Websites

often store the source code of JavaScript on different sites or in different directories on either a CDN or local, the path of which sometimes includes information about the libraries used and version number. In addition, libraries often include direct references to the library's name and version number. This is sometimes done by having a simple comment at the start of a file or a code segment with information including library name and even version numbers. This is made even more difficult since methods such as minification attempt to reduce the file size as much as possible without changing the execution behaviour, for example, removing white spaces or making variable names shorter. This also functions as a way of obfuscating the workings of the code, making it harder for a user to understand the code and thus making it harder to exploit any vulnerability. In this section, we describe a few possible approaches that our product implements and in Chapter 5, we show how successful these methods were.

4.3.1 File name

The first approach is the name of the file itself. For example, consider the file `js-sha3/0.8.0/sha3.min.js`[22], a popular JavaScript library used for the sha3 function. The path to the file itself clearly states not only that the name of the library is *js-sha3* but also that the version is *0.8.0*. Using a text parser on the path, the information of the library name as well as the version, can be extracted. The details of the parser will be discussed in a different chapter. This method alone is not enough since we found that it is not uncommon for the file paths to be obscured or unrelated to the library's name. However, in some instances, it proved very useful, and with the support of other methods, it was robust in the sense that it rarely gave false positives, and if a library combined with a version was found, it gave an accurate result.

4.3.2 Comments in Code

A second approach is annotations made in the comment blocks in the code itself. In the same file mentioned earlier, there is a comment at the top of the file that states the following[23].

```
/**
 * [js-sha3]{@link https://github.com/emn178/js-sha3}
 *
 * @version 0.8.0
 * @author Chen, Yi-Cyuan [emn178@gmail.com]
```

```
 * @copyright Chen, Yi-Cyuan 2015-2018
 * @license MIT
 */
```

Many other libraries begin with similar comments that provide the name of the author, the name of the library, and the version of the JavaScript package. However, like the convention of including the name and version in the file path, it is merely a convention and not all libraries and websites do this. The comments are also not standardized in a way that a comment parser can easily interpret them. An additional challenge is extracting the comments from the rest of the code in the JavaScript file, which proves hard if the file has gone through minification or obfuscation methods, provided there are any comments in the file at all. This approach may also be more prone to false positives compared to parsing the file name. It was common for libraries to allude to other libraries in their comments without using them, or some keywords such as *copyright* or *return* were incorrectly labelled as libraries. Library names that included other library names, such as *jquery* and *jquery-ui*, were also sometimes mismatched. These errors were mitigated through checks used in our text parser. Overall, however, this approach was very useful in identifying dependencies along with their versions, particularly when a single file contained multiple dependencies, each independently labelled.

4.3.3 Keywords in Code

A third approach is the code of the dependency itself; instead of having comments in the code, some libraries contain variables and functions that can be used to extract the name and version of the library. One example from the popular time parsing system *moment* contains the line `moment.version = '2.29.4';`. Using this line, a JavaScript interpreter can query the variable `moment.version` in the *moment* module and then have the string 2.29.4 returned to them[24]. This method is helpful if the name of the packet is already known; however, if the name of the dependency is not already known, this information is neither practical nor accessible. Additionally, it requires some form of parsing and interpretation of the JavaScript file, which can both be challenging to implement and a potential security risk for the implementation.

Using a similar approach to parsing the code, the version and name of a library can be extracted with the help of signatures of the function and variables found in the code itself. Several libraries provide information about themselves via functions or values that are distinguishable by name. For

example, the when using the popular library *jQuery* the client can call the function `jQuery.fn.jquery` to retrieve the version number. All the plug-ins we tested use this method to detect libraries[16][15]. This approach was experimented with but not deemed reliable enough to pursue. The main drawback of this method is that each library needs to be checked independently for each file found, which is not feasible, neither from an efficiency point of view nor from the time it would take to implement the functionality for each library. However, current solutions can use this way because they are already written in JavaScript, but also because they mainly focus on the most popular dependencies and are continuously maintained.

4.3.4 Comparing File Hashes

Hash functions take a string as an input and produce a unique value by applying certain operations on the data of the string itself. They allow for an easy way of quickly comparing strings, or data in general, as comparison need not be performed for the whole data. As the libraries shared and used on the internet are, in most cases, from the same source, the file content is identical and can therefore be compared by simply comparing the hash of the file. Using this approach, a database could be created with the hashes of the most commonly used dependencies. Then, to verify a dependency found by the scraper, it would hash the dependency file and then use the hash as an index in the database to quickly look up the dependency. The drawback, however, is that minor alterations to the file may result in the signature hash being different, thus not yielding a match in the database. This is the key issue as to why this method was not explored further since there are many ways that files are modified before they are presented on the end-user's device. In addition, the hash of the source files for multiple different versions needs to be obtained somehow to construct the index table that is used to compare the hashes against.

4.3.5 License

Almost all dependencies come with a license attached. The license dictates under which circumstances the code can and should be used, for example, the extent to which the code can be used in a commercial setting, how it may be modified or distributed, and whether the dependency owner can be held liable. One standard license is the MIT license, commonly used by widespread dependencies due to liberal permissions. However, the MIT license does

require that the entity that imports a dependency must include a copy of the license along with the dependency[25]. This results that the list of licenses used by a website can give clues as to the libraries that the web page uses. One such example is the following:

```
/*!
 * jQuery UI Draggable 1.11.4
 * http://jqueryui.com
 *
 * Copyright jQuery Foundation and other contributors
 * Released under the MIT license.
 * http://jquery.org/license
 *
 * http://api.jqueryui.com/draggable/
 */
```

Looking at this, it is trivial for a human to see which dependencies are used along with their version number. However, a computer program still requires some form of text parsing, which will be explained in a different section. This file can sometimes be accessed by adding the string `LICENSE.txt` and the end of the path to the JavaScript file. It is unlikely to produce an inaccurate result since all the information is clearly laid out.

4.3.6 Map files

Some JavaScript bundling services provide the original, non-minified source code on a web location available to the client, usually indicated by a comment in the minified JavaScript source code file containing the absolute or relative URL of a *sourcemap* file. The *sourcemap* file is formatted in JavaScript object notation (JSON). It contains the original source code and information about how to expand the bundled source code to several files.

```
//# sourceMappingURL=example.js.map
```

The original source that these files contain sometimes includes license comments that are not present in the minified code or the LICENSE.txt file if such a file exists.

4.3.7 Version Extracting

For the different approaches, there needs to be a system that can extract the version name from the different forms of strings obtained from either the file

name, comments in the file, license files, or map files. This was done by implementing a word-trie data structure. Using the GAD, a trie data structure is constructed where each root node represents the first word of each library in the database. A list of popular NPM libraries was also added to the trie to account for dependencies that do not have a reported vulnerability. The children of each node represent the composition of the root word and the following word in the library name. In addition, each node has a boolean value indicating if the particular composition this node's relative position in the tree represents is a full name in the database. This structure allows for matching library names, which are composites of other library names. Here an algorithm is constructed which consumes input text and matches the longest possible composite name in the trie. For example, *jQuery* and *jQuery UI* both exist in the database, but we do not want to match *jQuery* if *UI* is present. Using this trie, we can get the longest possible match of a composite library name without the need to store each composite name as a separate entry. For example, Figure 4.2 shows how the trie will be constructed; several React packages are inserted, the names in bold font indicate that the composite word is a valid library, while roman font indicates composite words that are not valid libraries.

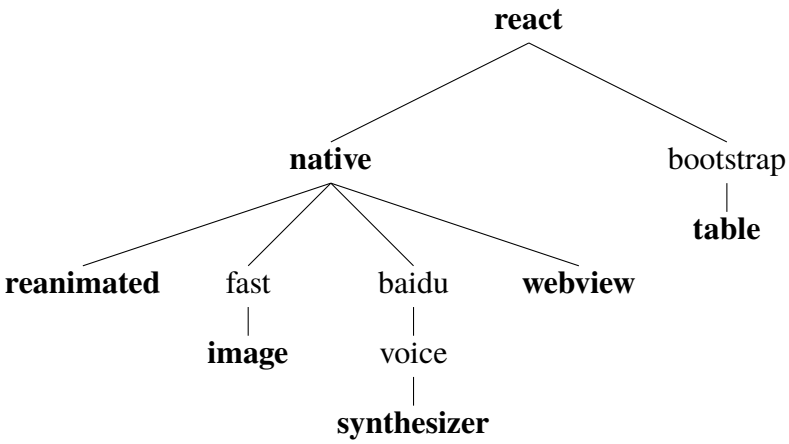


Figure 4.2: trie constructed from composite names of react packages, the full path to a bold font entry represents an existing package name

The trie not only allows for easy construction of context-sensitive matching algorithms, but the search is also a relatively fast procedure, where the asymptotic lower bound, of the time complexity, is $\mathcal{O}(n)$.

The existence of a known library name in a comment does not give any

particularly meaningful information as to what libraries are used by itself. To extract the name and version of the library, certain steps are taken to parse this information. The first one is to tokenize the words into arrays of tokens. Given the example comment below.

```
/*! jQuery UI - v1.13.2 - 2022-07-14
* Copyright jQuery Foundation and other contributors;
* Licensed MIT */
```

The tokenization would first remove all non-alphanumeric characters except the exclamation point (`!`) and the period (`.`) as they are used in other steps. This results in the following array of tokens.

```
['!','jQuery','UI','v1.13.2','2022','07','14',...]
```

In order to determine if the comment contains information about the library, it is scanned for the following keywords `license, copyright, c, !, mit, bsd, apache, gpl` after adjusting everything to lowercase. If at least one of these is present, the comment is considered relevant. Thereafter, the following regular expression is used to scan for version numbers.

```
v?(\d{1,3}\.\d+(?:\.\d+)+).*
```

The regular expression works as follows:

- `v?` matches the optional character `v`.
- `\d{1,3}` matches one to three digits.
- `\.` matches a single dot (punctuation mark).
- `\d+` matches one or more digits.
- `(?:\.\d+)+` matches one or more groups of a dot followed by one or more digits.
- `.*` matches zero or more of any character.

The regular expression will match expressions similar to `vX.Y.Z`, where X is a number with one to three digits and both Y and Z are numbers with one or more digits. In the example, it will match token `v1.13.2`, which has position 2 in the token array (counting from 0).The regular expression will match expressions similar to `vX.Y.Z`, where X is a number with one to three digits and both Y and Z are numbers with one or more digits. In this case, it

will match token `v1.13.2`, which has position 2 in the token array (counting from 0). A simple algorithm then calculates the distance between discovered names in the comment block and the version number token and subsequently chooses to match the closest known name to a particular version number.

```
['!','jQuery','UI','v1.13.2','2022','07','14',...]
[ 3 ,   2    , 1  ,    0    ,  1   , 2  ,  3 ,...]
```

The largest continuous match of tokens was then found by querying the trie with the tokens in the comment block. In this case, the largest match is `jQuery UI`. If there are multiple matches of the same length, the one which is closest to the beginning of the paragraph is selected. If there are multiple version numbers in the same comment, the process is repeated for each version number that is found. In the end, the result is a list of library names and version numbers that are present in the comment block. In this case, the result would be `"jQuery UI"` with version number `1.13.2` after removing the `"v"`. We raise natural language processing (NLP) as a different interesting approach as a topic of further research.

4.4 Database

Since JavaScript dependencies are so widely used for web applications, there is a significant interest in keeping track of which dependencies that contain a vulnerability. Therefore, databases exist solely to collect information and data about potential vulnerabilities and security flaws of most commonly used JavaScript packages and dependencies. This is done to help developers locate and address these and allow them to improve their code. These databases include the National Vulnerability Database (NVD), run by the US government [12], the common vulnerabilities and exposures (CVE) database[10], and the Open Web Application Security Project (OWASP) Dependency Check [26]. There are also commercial applications with the same issue in mind, such as Snyk, Retire.js, and OWASP ZAP. Along with their databases, they provide tools and plug-ins that also help scan for source code vulnerabilities and check for outdated dependencies [17]. These databases and tools are essential to reduce security risks but should also not be relied upon as they only consider exploits that have been found and published. Furthermore, developers should take other steps to stay ahead of security threats.

In previous studies, it has been pointed out that despite this plethora of alternatives, there is no readily centralized database for JavaScript dependency

vulnerabilities[7]. However, since early 2022, the code hosting website GitHub has opened up the GitHub Advisory Database (GAD) to public contribution. This database is an open-source database of vulnerabilities from commonly used development ecosystems. It is maintained by GitHub, with the advantage that all entries are stored in a regular git repository[27]. This repository can thus easily be cloned with the `git clone` command and updated in the future when new vulnerabilities are added with `git pull`, making the implementation future-proof and easy to maintain. Each entry contains the name of the package, the ecosystem from which it originates, which versions the vulnerability affects, the severity ranked from low to critical, and a short summary and description of the vulnerability[28]. By cloning the repository and allowing a script to extract the needed information, this data is stored in a structured database to be later indexed when the crawler finds packages that are in use. When using a database such as the GAD, the correct license needs to be adhered to. A downside to using a third-party database is the risk that the database will eventually not be funded and no longer be updated. In this case, however, we have reason to believe that the backing from GitHub will continue to fund the development into the relevant future.

This database has an emphasis on being well-structured, and each report follows a strict format that facilitates quick lookups by a computer. The GAD consists of JSON files, one for each reported vulnerability. A Structured Query Language (SQL) database is constructed using the information in the GAD files, and a table is created containing information on each vulnerability.

```
CREATE TABLE vulnerabilities (id, name, summary, ...);
```

By iterating through thesis files and singling out vulnerabilities in the "NPM" ecosystem, we insert each file's data into the table.

```
INSERT INTO vulnerabilities (id, name, summary, ...)
VALUES (GHSA-xxxxx,yyyy,...);
```

Chapter 5

Results and Analysis

This chapter discusses the results of our implementation and compares it to an existing baseline. The performance of the different methods used are also compared with each other. The results also create some commentary on the amount of currently unsolved vulnerabilities on the web. Lastly the reliability of our data is evaluated.

5.1 Comparison of Found Dependencies

After testing the different implementation on the 14 websites we plotted the percentage of dependencies found by each solution in relation to the number of manually found dependencies. We also plotted the average percentage of dependencies found. This can is illustrated in Figure 5.1. A table with the data that produced this Figure is avaiable in the appendix.

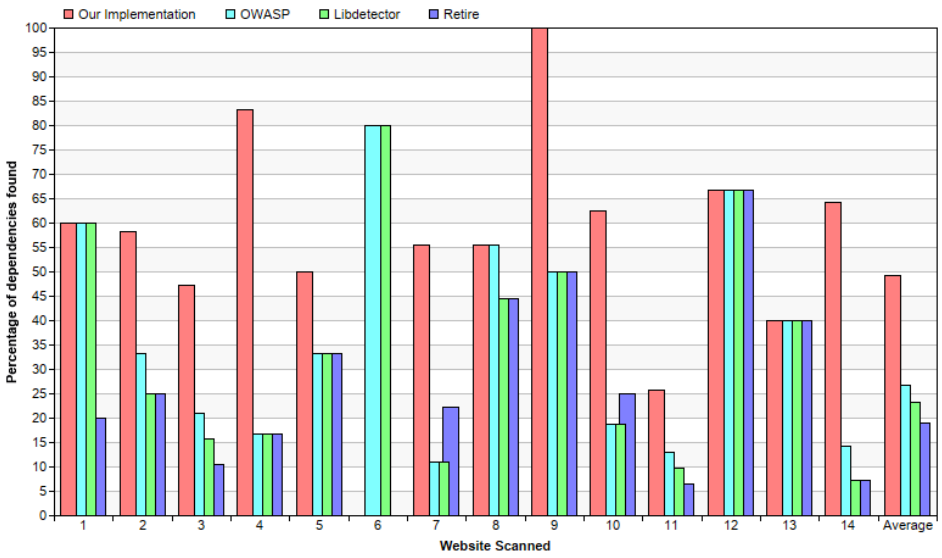


Figure 5.1: Percentage of dependencies found by each solution for each scanned website

As visible in Figure 5.1, our implementation managed to find more dependencies than all the tools it was compared against as it on average found 49% of dependencies on a webpage compared to the 27%, 23% and 19%, found by OWASP, Libdetector and Retire respectively. Additionally, we found that our product reported one false positive on web page number 6; a library that on closer inspection did not exist. None of the other solutions reported any false positives.

5.2 Performance of different methods

The final implementation used a combination of the information found in the URL, License file, JavaScript comments and potential source map files if references to them existed. The performance of these methods was evaluated by counting the number of dependencies found with each method from the aforementioned test case. If the same dependency was found by more than one method, it was counted towards both methods. The performance of the different methods used for dependency detection is displayed in Table 5.1. Here is each dependency, either a true positive or false positive found by a particular method across all the sites, recorded. Unique dependencies, which were not discovered by any of the other products tested, are also shown.

Table 5.1: Different type of finding depending on method used

Type of Finding	URL	License File	JavaScript Comment	Source Map
True Positive	15	20	44	0
False Positive	0	0	1	0
Unique	1	16	26	0

Table 5.1 shows the comments in JavaScript code were the method that found the most libraries, followed by the license file and the URL. The source map file provided no findings, most likely because none of the websites used it to package their web pages. License files also produced proportionally more unique values, with nearly all the license findings being unique compared to just over half of the JavaScript comments. License files may produce more unique values due to not directly linked in the HTML file but rather are fetched based on the JavaScript file, an approach that may not be considered by the other tools we compared against. We also suspect that some websites are overly cautious when importing licensing from bigger projects and thus include licensing for programs that they do not use. In these cases, we still consider the finding a true positive since it is unfeasible to determine the extent a licensed dependency is actually used in the code. The library that was false reported to be present was the library `inflect`, whose name in the NPM repository is `"i"`[29].

5.3 Found vulnerabilities

Most of the libraries found did not contain vulnerabilities and thus originated from the list of popular NPM libraries. From inspection, it was also very common that the same library was used in many of the websites scanned. For all the libraries that contained a vulnerability, the vulnerability would be fixed if the library was updated to the latest stable release. The GAD rates the severity of a vulnerability from *low* to *critical*, table 5.2 shows the breakdown of all websites and libraries with their highest severity.

Vulnerability found in	None	Low	Moderate	High	Critical
Dependency	51	0	5	2	0
Website	8	0	4	2	0

Table 5.2: Highest severity of found vulnerabilities, for each dependency and website

A total of seven vulnerabilities were found, and from these vulnerabilities, we found that six out of 14 sites had at least one known vulnerability on their homepage. Most of these were the same dependency used in multiple websites. This portion of websites (43%) with at least one dependency with a known vulnerability is slightly higher than the 37% found by[7].

5.4 Reliability of Data

The data collected by our implementation and by the implementations we compared against were both objective in the sense that the results were deterministic each time a website was scanned. However, the manual search is less objective because human error makes the approach prone to mistakes and may miss certain dependencies or mistakenly report on ones not used. We conducted the manual search without knowledge of the results of our own implementation in order to prevent bias in the dependencies we picked. After the first manual round of collection, we also ensured that each dependency reported by any implementation was verified. We did have to make a distinction of which libraries we would count as a web dependency versus a different form of iteration, such as a bundler or API interface. We also ensured that each scan was carried out on the same website and that no update of the websites had occurred between scans.

The websites were chosen on the criteria of being deemed similar in some aspects to the intended target websites of external stakeholders. These criteria were that the websites were to be similar in functionality and that the websites belong to Swedish organization. This may have had an effect on the results, as the test sites used during development of our product were chosen on the same criteria, this could create an inductive effect on the product's development, skewing the results in the favour of our product.

5.5 Discussion

We observe that our approach found more dependencies than any of the existing solutions tested. Our product, however, had one false positive while the other solutions had none. This discrepancy depends on the matching techniques and the different databases for recognizing known libraries. Our product uses a bottom-up approach of semantically matching known library names to a version number in a general license-style comment format. The existing solutions tested instead use library-specific matching, i.e. a

specific regular expression tailored to match a specific library's license comment[15][16][30]. The databases also differ in that our product uses an aggregate of libraries in the NPM ecosystem with known vulnerabilities and the top 1,000 NPM libraries. The other solutions use a smaller, tailored database containing the library and the specific matching algorithm or regular expression for that library. This allows us to match a much larger variety of libraries with slightly lower accuracy than the alternative solutions.

Many of the libraries that were missed by our implementation were simply missed because they were free from vulnerabilities and thus not part of the dependency trie discussed earlier. Manually adding the missed dependencies to the trie often resulted in the dependency being found, meaning that it likely would have been found if it had a vulnerability that had been reported to the GAD.

Reporting false positive dependencies portrays an incorrect composition of a web-application. When vulnerabilities stemming from a false positive dependency are reported, the user receives false warnings, possibly leading to alert fatigue or overall distrust of the results. We recognize this problem as a severe negative impact on the quality of our product, however, we see it as unavoidable to a certain degree when using heuristical methods such as the ones described in Chapter 4.

The results showed that the method of checking license comments in the raw JavaScript files– not including source-mapped files– was by far the most superior in terms of the number of found dependencies. Checking license files, however, was the method which, proportionally, yielded more dependencies that were exclusively found by only our implementation. The reason for this is that the other solutions tested do not search for license files, and developers that use license files are– depending on license type– usually not required to include the license information in the license file in any other part of the web page.

Our solution uses a local database relying completely on the accessibility and maintenance of Github's advisory database, GAD, to keep the database up to date and working. The other discovered solutions relies on local databases of library-specific regular expressions or similar as a means to store known libraries. Where our approach suffers from complete dependency on GAD, the other solutions depends completely on contingencies within the library itself, for example, the format of a library's license comment or the name of a variable containing the library's version. We found no issues implementing the database with a proven and efficient SQL database. The database itself is 3.7 megabytes in size and is well optimised for fast accesses [31].

Chapter 6

Conclusions and Future Work

In this Chapter we conclude the study and discuss possible improvements to the product while also suggesting further areas of exploration.

6.1 Conclusions

Although our pre-study found little research on the subject of front-end library detection, we found several existing solutions with open source code. Looking at a few key papers and existing solutions, we found several methods being used for detecting dependencies in JavaScript web applications. These methods are:

- matching known names with filenames
- parsing license comments in the source code
- searching for and parsing license files
- comparing hash values of source code with known libraries
- searching for keywords within the source code itself
- searching for and applying the methods above on map files

Our product did not implement all techniques we discovered in our pre-study, for example, comparing file hashes or interpreting the JavaScript and using console commands to deduce the existence of libraries. We also opted for a different approach with our database, using Github's Advisory Database together with a list of the top 1,000 NPM libraries to create a local SQL database.

Even though we did not use all of the methods discovered in the pre-study, we still managed to outperform all the other products we tested. In the end, our results show that the most efficient combination of the tested methods was to check the license information of the websites by looking at comments and searching for license files. The products we tested and the only truly relevant solution described in an academic paper [7] all use a predefined set of known libraries with matching procedures specific to those libraries. Our method, on the other hand, applies a large set of known names to a general matching procedure, allowing us to match more libraries. However, this approach is more prone to producing false positives.

Our implementation was successful because it satisfies the requirements of our external stakeholder. The original problem was also successfully addressed. Through this project, we have learned numerous things, a major one being the need for quality and a wide variety of testing data. In order to let the product successfully generalize, it needs to be developed with a wide variety of websites in mind to adapt to unseen scenarios. This thesis work provided us with a great deal of insight into web development, which frameworks, tools and libraries are used, how developers work with these and how prevalent the issue of *dependency hell* – the numerous libraries used in web development and the lack of insight into these libraries dependencies– is in web development.

If we were to continue or rework our product with the knowledge gained from this thesis work, we would use a browser driver framework such as Selenium instead of creating our own crawler. This would not only save time but provide more functionalities, such as the possibility of executing JavaScript code on a browser when crawling and searching for dependencies. We recommend that anyone planning to construct a similar product use the data or techniques from existing products.

6.2 Future works

We found as the research and development of the product proceeded that there were many aspects of the subject which were, because of the time limit, necessarily neglected. Although there are plenty of products providing website pen-testing capabilities, we found very few products focused on detecting libraries and their vulnerabilities solely from a client perspective. Moreover, all the solutions found used the same, relatively simple methods for detecting dependencies.

We see significant potential in applying more sophisticated methods to

front-end library detection, such as using named entity recognition or other NLP methods. This would potentially allow for the identification of libraries that were not seen before. The hashing method explained in Section 4.3.4 could also be beneficial. Although it is doubtful that this method identifies new libraries, it may be beneficial to use it in combination with more uncertain methods, such as parsing comments in JavaScript source code.

We believe that an exhaustive library database could be used to identify more niche and less common libraries. Construction of such a database might be difficult and impractical, but it would be beneficial not only for the purpose of dependency detection but might be a valuable resource for other applications and for web developers.

We also see further demand for scanning of sides that are less accessible, such as web pages hidden behind logins that require PUT requests or more sophisticated forms of authentication. This might be easily done using a web driver framework such as Selenium.

Additionally, specifically tailored scanners for popular frameworks such as React [32] or Angular [33] can be explored as the usage of frameworks increases in popularity. This approach can also be extrapolated onto websites not using JavaScript but also alternative web execution such as WebAssembly (WASM).

Further research into the subject might benefit from the use of static analysis of JavaScript code, such as dead code detection, to see if the website actually uses the included or referenced libraries on its site.

Finally, since many modern apps for phones and tablets are emulated websites, a scanner targeted explicitly for them could also be an area that needs further exploration.

The results of this thesis shine a spotlight on the need for an adaptation of more systematic methods to manage and proactively discover vulnerabilities in JavaScript dependencies; until then, vulnerable dependencies will quietly remain on the front-end of numerous websites.

References

[1] S. Nyheter and J. Toresson, "It-attacken mot coop – detta har hänt," *SVT Nyheter*, 07 2021. [Online]. Available: https://www.svt.se/nyheter/inrikes/it-attacken-mot-coop-detta-har-hant [Page 3.]

[2] IBM, "Cost of a data breach report 2022 2," 2022. [Online]. Available: https://www.ibm.com/downloads/cas/3R8N1DZJ [Page 3.]

[3] S. Raemaekers, A. Van Deursen, and J. Visser, "An analysis of dependence on third-party libraries in open source and proprietary systems," *Sixth International Workshop on software quality and maintainability*, vol. 12, pp. 64–67, 2012. [Page 3.]

[4] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns, "Scriptprotect," *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 07 2019. doi: 10.1145/3321705.3329841 [Page 3.]

[5] S. Jain, D. S. Tomar, and D. R. Sahu, "Detection of javascript vulnerability at client agen," *INTERNATIONAL JOURNAL OF SCIENTIFIC AND TECHNOLOGY RESEARCH*, vol. 1, 08 2012. [Online]. Available: https://www.researchgate.net/publication/236155747_Detection_of_JavaScript_Vulnerability_At_Client_Agen [Page 4.]

[6] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor, "The lifetime of android api vulnerabilities: Case study on the javascript-to-java interface," *Security Protocols XXIII*, pp. 126–138, 2015. doi: 10.1007/978-3-319-26096-9$_1$3 [*Page* 4.]

[7] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," *Proceedings 2017 Network and Distributed System Security Symposium*, 2017. doi: 10.14722/ndss.2017.23414. [Online]. Available: https://arxiv.org/abs/1811.00918 [Pages 4, 28, 32, and 36.]

[8] L. Hilty and B. Aebischer, *ICT for Sustainability: An Emerging Research Field*, 01 2015, vol. 310, pp. 3–36. ISBN 978-3-319-09227-0 [Page 6.]

[9] T. Berners-Lee and D. Connolly, "Hypertext markup language - 2.0," *www.rfc-editor.org*, 11 1995. doi: 10.17487/RFC1866. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1866 [Page 9.]

[10] M. Corporation, "cve-website," www.cve.org. [Online]. Available: https://www.cve.org/About/Overview [Pages 11 and 27.]

[11] C. Details, "Cve security vulnerability database. security vulnerabilities, exploits, references and more," Cvedetails.com, 2009. [Online]. Available: https://www.cvedetails.com/ [Page 11.]

[12] NIST, "Nvd - nvd dashboard," nvd.nist.gov. [Online]. Available: https://nvd.nist.gov/general/nvd-dashboard [Pages 11 and 27.]

[13] M. Khder, "Web scraping or web crawling: State of art, techniques, approaches and application," *International Journal of Advances in Soft Computing and its Applications*, vol. 13, pp. 145–168, 11 2021. doi: 10.15849/ijasca.211128.11 [Page 12.]

[14] M. Gheorghe, F.-C. Mihai, and M. Dârdală, "Modern techniques of web scraping for data scientists," *Revista Romana de Interactiune Om-Calculator*, vol. 11, pp. 63–75, 2018. [Online]. Available: http://rochi.utcluj.ro/rrioc/articole/RRIOC-11-1-Gheorghe.pdf [Page 12.]

[15] RetireJS, "Retire.js," GitHub, 03 2023. [Online]. Available: https://github.com/RetireJS/retire.js [Pages 14, 16, 23, and 33.]

[16] J. Michel, "Release v6.0.0 · johnmichel/library-detector-for-chrome," GitHub, 07 2020. [Online]. Available: https://github.com/johnmichel/Library-Detector-for-Chrome/releases/tag/v6.0.0 [Pages 14, 16, 23, and 33.]

[17] Snyk, "Open source security management | snyk," snyk.io, 2023. [Online]. Available: https://snyk.io/product/open-source-security-management/ [Pages 14 and 27.]

[18] O. ZAP, "Owasp zap," GitHub. [Online]. Available: https://github.com/zaproxy [Pages 14 and 16.]

[19] S. Pinkham, "spinkham/skipfish," GitHub, 05 2022. [Online]. Available: https://github.com/spinkham/skipfish [Page 14.]

[20] V. Cothey, "Web-crawling reliability," *Journal of the American Society for Information Science and Technology*, vol. 55, pp. 1228–1238, 2004. doi: 10.1002/asi.20078. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.20078 [Page 20.]

[21] G. Agre and S. Dongre, "A keyword focused web crawler using domain engineering and ontology," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 4, 2015. doi: 10.17148/IJARCCE.2015.43111. [Online]. Available: https://www.ijarcce.com/upload/2015/march-15/IJARCCE%20111.pdf [Page 20.]

[22] CDNJS, "js-sha3," cdnjs. [Online]. Available: https://cdnjs.com/libraries/js-sha3 [Page 21.]

[23] C. Yi-Cyuan, "js-sha3," Cloudflare.com, 2015. [Online]. Available: https://cdnjs.cloudflare.com/ajax/libs/js-sha3/0.8.0/sha3.min.js [Page 21.]

[24] Moment.js, "Moment.js," GitHub, 06 2022. [Online]. Available: https://github.com/moment/moment/blob/develop/src/moment.js [Page 22.]

[25] L. E. Rosen, *Open Source Licensing*, 1st ed. Prentice Hall, 2005, vol. 1. [Online]. Available: https://www.immagic.com/eLibrary/ARCHIVES/EBOOKS/R050225R.pdf [Page 24.]

[26] OWASP-Foundation, "Vulnerabilities," owasp.org. [Online]. Available: https://owasp.org/www-community/vulnerabilities/ [Page 27.]

[27] Github, "About the github advisory database," GitHub Docs. [Online]. Available: https://docs.github.com/en/code-security/security-advisories/global-security-advisories/about-the-github-advisory-database [Page 28.]

[28] GitHub, "github/advisory-database," GitHub, 03 2023. [Online]. Available: https://github.com/github/advisory-database [Page 28.]

[29] P. K. Sunkara, "i," npm, 2021. [Online]. Available: https://www.npmjs.com/package/i [Page 31.]

[30] DenisPodgurskii, "pentestkit," GitHub, 04 2023. [Online]. Available: https://github.com/DenisPodgurskii/pentestkit/tree/master [Page 33.]

[31] SQLite, "35Available: https://www.sqlite.org/fasterthanfs.html [Page 33.]

[32] M. O. Source, “React,” react.dev, 2023. [Online]. Available: https://react.dev/ [Page 37.]

[33] Angular, “Angular,” Angular.io, 2019. [Online]. Available: https://angular.io/ [Page 37.]

Appendix A

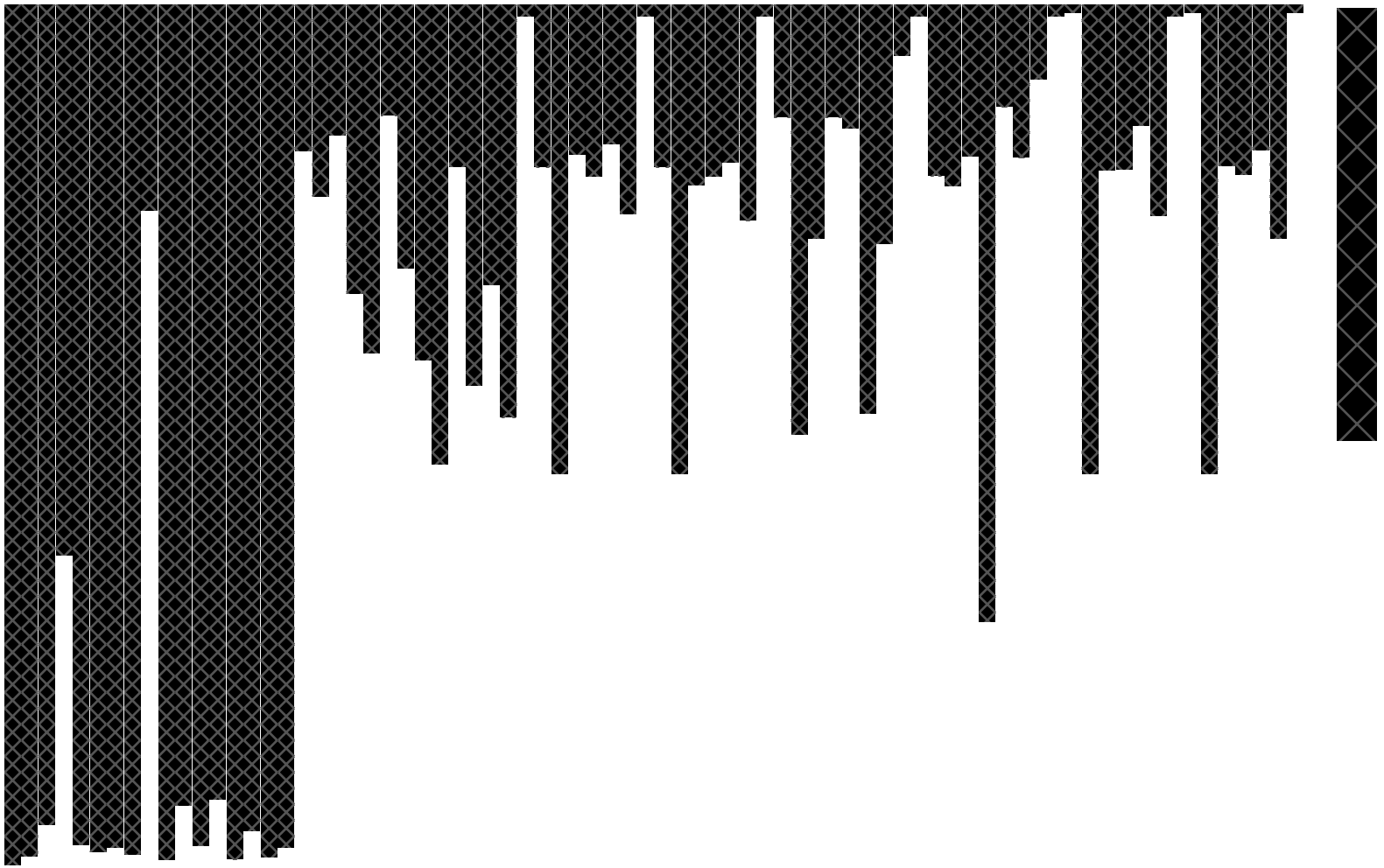
Implementation Comparisons

Figure 5.1 was produced from the data visible in Table A.1. After testing the websites, we evaluated them against our implementation on the criteria of the number of dependencies found and the fraction of found dependencies compared to the manually found dependencies. In addition, for each tested webpage, the solution(s) with the highest number of found dependencies is shown in bold font. The final row displays the total sum of found dependencies along with the proportion of total true positives found into the number of manually found dependencies as a percentage.

Table A.1: Found dependencies on each website, per product

Webpage	Our Implementation	Libdetector	Retire	OWASP	Manual
1	**3**	**3**	1	**3**	5
2	**7**	3	3	4	12
3	**9**	3	2	4	19
4	**5**	1	1	1	6
5	**3**	2	2	2	6
6	0	**4**	0	**4**	5
7	**5**	1	2	1	9
8	**5**	4	4	**5**	9
9	**2**	1	1	1	2
10	**10**	3	4	3	16
11	**8**	3	2	4	31
12	**2**	**2**	**2**	**2**	3
13	**2**	**2**	**2**	**2**	5
14	**9**	1	1	2	14
Sum	**70**	33	27	38	142
Percentage	49.3%	23.24%	19.01%	26.76%	100%

TRITA-EECS-EX- 2023:0000




CCCC,
CCCC,
CCCC,

acronyms.tex