On Measuring JavaScript Vulnerabilities in the NPM Packages, Websites and Chrome Extensions

Maryna Kluban

A Thesis

in

The Department

of

Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Information Systems Security) at

Concordia University

Montréal, Québec, Canada

August 2022

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Maryna Kluban**

Entitled: **On Measuring JavaScript Vulnerabilities in the NPM Packages, Websites and Chrome Extensions**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Jeremy Clark

_____ Examiner
Dr. Jeremy Clark

_____ Examiner
Dr. Carol Fung

_____ Supervisor
Dr. Mohammad Mannan

_____ Supervisor
Dr. Amr Youssef

Approved by _____
Mohammad Mannan,
Graduate Program Director

_____ 2022

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

On Measuring JavaScript Vulnerabilities in the NPM Packages, Websites and Chrome Extensions

Maryna Kluban

JavaScript is often rated as the most popular programming language for the development of both client-side and server-side applications. Because of its popularity, JavaScript has become a frequent target for attackers, who exploit vulnerabilities in the source code to take control over the application. To address these JavaScript security issues, such vulnerabilities must be identified first. Existing studies in vulnerable code detection in JavaScript mostly consider package-level vulnerability tracking and measurements. However, such package-level analysis is largely imprecise as real-world services that include a vulnerable package may not use the vulnerable functions in the package. Moreover, even the inclusion of a vulnerable function may not lead to a security problem, if the function cannot be triggered with exploitable inputs. In this thesis, we develop a vulnerability detection framework that uses vulnerable pattern recognition and textual similarity methods to detect vulnerable functions in real-world JavaScript projects, combined with a static multi-file taint analysis mechanism to further assess the impact of the vulnerabilities on the whole project (i.e., whether the vulnerability can be exploited in a given project). We compose a comprehensive dataset of 1,360 verified vulnerable JavaScript functions using the Snyk vulnerability database and the VulnCode-DB project. From this ground-truth dataset, we build our vulnerable patterns for two common vulnerability types: prototype pollution and Regular Expression Denial of Service (ReDoS). With our framework, we analyze 9,205,654 functions (from 3,000 NPM packages, 1892 websites and 557 Chrome Web extensions), and detect 117,601 prototype pollution and 7,333 ReDoS vulnerabilities. By further processing all 5,839 findings from NPM packages with our taint analyzer, we verify the exploitability of 290 zero-day cases across 134 NPM packages. In addition,

we conduct an in-depth contextual analysis of the findings in 17 popular/critical projects and study the practical security exposure of 20 functions. With our semi-automated vulnerability reporting functionality, we disclose all verified findings to project owners. We also obtained four CVEs for our findings, two of them rated as 9.8/10 (critical) severity, one as 9.1/10 (critical), and one as 7.5/10 (high) severity; several other CVE requests are still in the process now. As evident from the results, our approach can shift JavaScript vulnerability detection from the coarse package/library level to the function level, and thus improve the accuracy of detection and aid timely patching.

Acknowledgments

I would like to thank Dr. Mohammad Mannan and Dr. Amr Youssef for their support and guidance throughout my Masters studies. Their knowledgeable input and their experience prompted me and my research to success. Without their advice and support throughout this project this work would not have come to life. I would also like to express my gratitude for their patience in times when the work on my experiments took too long to bring positive results. I am also grateful to the professors and administration of the faculty of Information Technologies in Taras Shevchenko University of Kyiv, for giving me the opportunity to pursue a Masters degree at Concordia University, and to the Mitacs organization for making it possible.

I received substantial financial support from my supervisors and Concordia University. I am thankful to all for easing the financial burden while doing this research.

Lastly, I would like to thank my family and friends. This journey would not have been possible without their encouragement and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we first introduce the reader to our area of interest by outlining statistical data and use cases for JavaScript programming language. Then we motivate our work, which is detailed in this thesis, by describing existing security weaknesses in JavaScript as well as a lack of studies and solutions for their detection and prevention. Afterward, we propose our approach to JavaScript vulnerability detection using multiple methods of static code analysis. We then describe our contributions and findings. Finally, we provide a brief overview of the structure of this thesis.

1.1 Overview

JavaScript is a dynamic computer programming language with object-oriented capabilities, created in 1995. It is used in 97.5% of all web applications [1]. According to GitHub [2] and StackOverflow [3], JavaScript has been a dominating language for software projects development for at least the last 7 years. Nowadays JavaScript is mostly known as the scripting language for websites. However, it is actively used in numerous other environments, both client-side and server-side. To adapt pure JavaScript language to different usage purposes, frameworks are created.

A JavaScript framework is a library of JavaScript code, which provides developers with complex solutions to specific environments and/or use cases. There are overall around 83 JavaScript frameworks [4]. Client-side JavaScript frameworks, such as React (`reactjs.org`), Angular (`angular.io`), Vue.js (`vuejs.org`) provide APIs and tools to develop interactive interfaces

for web pages, web extensions, games, desktop and mobile applications etc. Server-side JavaScript frameworks (e.g., Express.js (expressjs.com), Koa (koajs.com), Nest.js (nestjs.com)) are mostly based on the NodeJS execution environment (nodejs.dev) and provide functionality to simplify URL routing, interaction with databases, authorization management and help implement security measures against web attacks. The functionalities that the frameworks provide add complexity to the core JavaScript language, thus introducing additional potential vulnerabilities.

1.2 Motivation

Due to its wide usage JavaScript is a very common target for attackers. Client-side attacks occur on the users' system when the user initiates malicious actions knowingly (as an attacker) or unknowingly (as a victim). These attacks exploit vulnerabilities in the client-side JavaScript code, which, due to its nature, is fully accessible to the user. Client-side attacks include Cross-site Scripting (XSS), data infiltration, content injection, etc. In contrast, server-side attacks target the vulnerabilities in server-based JavaScript applications that manage databases, web / mail / file servers, etc. These attacks include SQL injection, improper access control, authentication bypass, Denial of Service (DoS), etc.

The consequences of exploiting vulnerabilities in JavaScript source code can be information theft or forgery [5], malicious code injection [6], redirection to attacker-controlled sources [7], disruption of an application functionality [8] and much more.

Previous work [9, 10, 11, 12, 13, 14, 15, 16] on JavaScript vulnerabilities mostly covers the propagation of vulnerable NPM packages among real-world projects, primarily based on the project dependency information. While such work is very useful in identifying projects with vulnerable dependencies, they do not provide fine-grained information on the use of actual vulnerable functions from the vulnerable packages. This leads to projects being flagged as vulnerable due to their use of vulnerable dependencies, when in reality many such projects (73.3% according to Zapata et al. [11]) are not vulnerable as they do not actually use the vulnerable functions from their dependencies.

On the other hand, only a few studies rely on code-based approaches for JavaScript vulnerability detection. Ferenc et al. [17] use static code metrics, generated for each function by static analysis

tools (OpenStaticAnalyzer [18] and escomplex [19]), as features for machine learning (ML) algorithms that predict the probability of the function being vulnerable. Mosolygó et al. [20] use generalized representations of code lines, and calculate vulnerability likelihood based on cosine distance between vulnerable and analyzed code lines. However, results from both approaches are not very encouraging (F1-measure of 0.7 for [17], and 97.3% false positives for [20]).

The issue of insufficient vulnerable code datasets also hinders research in this area. Ferenc et al. [17] compiled the first publicly accessible dataset of JavaScript functions which includes 1,496 functions marked as vulnerable. In a following study, Mosolygó et al. [20] reduced the dataset from [17] to 443 vulnerable functions by manually filtering out false positives. After examining the remaining functions we discovered that some non-vulnerable functions still exist in the dataset.

1.3 Problem statement

The objective of this thesis is to assess active JavaScript projects in the real-world, in order to find vulnerable functions in their source code. We gather JavaScript code from three different environments: NPM packages, Chrome web extensions, and top popular websites. To detect vulnerabilities in the collected code, we develop a vulnerable function detection framework based on the following approaches:

(1) vulnerable pattern recognition, which uses our manually developed patterns to perform a semantics-based search in code;

(2) search based on textual similarity, which uses content-sensitive and cryptographic hash comparison, and

(3) static taint analysis to verify the exploitability of the vulnerability.

For the first approach, we utilize a static analysis tool Semgrep [21], which allows us to develop advanced patterns based on JavaScript semantics. For the second one, we tokenize each function from both vulnerable and real-world datasets and generate a content-sensitive hash value using the SimHash [22] algorithm along with a SHA-1 cryptographic hash value. For each hash value from the real-world dataset, we then search for matches in the vulnerable function dataset. In the third

approach, we develop a static taint analysis tool to verify that a flagged vulnerability is reachable by an attacker in the related project. We combine an Abstract Syntax Tree (AST) representation with our proposed File Dependency Graph (FDG), which enables multi-file taint-tracking. By combining these three approaches into a vulnerability detection and verification framework, we achieve good scalability from the first two approaches, and excellent precision from the third one.

We address the lack of the source of vulnerable code by composing our own dataset of vulnerable JavaScript functions. We use the Snyk vulnerability database [23] and VulnCode-DB project [24] to extract meta-information (e.g. vulnerability description, CVE number, affected project's name and version), and the code/functions for each vulnerability entry. We then perform semi-automated function vulnerability verification. Our final vulnerable JavaScript dataset contains 1,360 verified vulnerable JavaScript functions.

1.4 Contributions

Our contributions can be summarized as follows:

(1) We automatically crawl for vulnerable JavaScript functions from Snyk [23] and VulnCode-DB [24], allowing us to add the newly reported vulnerable functions and keep our dataset up-to-date. Then, we create vulnerability patterns and a web-based tool for efficient manual verification of vulnerable functions. In the end, we compose a relatively comprehensive, semi-automatically verified dataset of 1,360 JavaScript vulnerable functions.

(2) We develop an experimental vulnerable function detection framework that consists of the combination of pattern and textual similarity based approaches. The framework includes our manually developed rule sets for vulnerable pattern detection of two common vulnerability types: JavaScript prototype pollution and regular expression denial of service (ReDoS).

(3) We gather a large dataset of 9,205,654 JavaScript functions from active real-world projects from three different application types (NPM packages, Chrome extensions and top websites). This collection process is also fully automated, allowing us to increase the dataset as more projects become available. We then utilize our vulnerability detection framework to identify vulnerable functions in this dataset.

(4) We detect 124,934 vulnerable functions with an estimated average precision of 94.5% (based on manual verification of a small subset). We perform a case study on 16 popular/critical projects that contain 20 vulnerable functions; all these functions flagged by our framework are found to be, indeed, exploitable. We are currently in the process of contacting the affected parties. We also plan to coordinate with Snyk and Chrome teams for effective dissemination of our findings to the affected NPM and Chrome extension developers.

(5) With our taint analysis, we identify 309 cases from 134 NPM packages (5.7% of all findings in NPM packages), which are exploitable in the project context. Manual verification of 100 cases detected no false positives produced by the taint analysis mechanism.

(6) To deal with a large number of disclosure notices, we develop a semi-automated technique to report our findings. We firstly search for duplicates of our findings in the CVE database (and identify 19 cases), and then automatically compose readable vulnerability reports for the remaining 290 findings and send them to 112 responsible project developers. Furthermore, all findings will be submitted according to the responsible disclosure procedure either directly to MITRE by us or through the GitHub Security Advisory [25] by the project owners.

(7) We submitted eight findings directly to MITRE [26] and obtained four CVE IDs as of writing, with two severity scores of 9.8 (Critical) (Minimist, CVE-2021-44906, 50.7 million weekly downloads, nvd.nist.gov/vuln/detail/CVE-2021-44906, and SailsJS, CVE-2021-44908, 8,415 active websites), nvd.nist.gov/vuln/detail/CVE-2021-44908), one score of 9.1 (Critical) (Ramda, CVE-2021-42581, 10.4 million weekly downloads), nvd.nist.gov/vuln/detail/CVE-2021-42581) and one of 7.5 (High) (Async, CVE-2021-43138, 56.6 million weekly downloads). nvd.nist.gov/vuln/detail/CVE-2021-43138). The other four requests are being processed as of writing.

We have disclosed all our findings to the respective project owners and summarized the disclosure process in Section 4.3.2. Some of the work presented in this thesis has been peer-reviewed and accepted in the following article:

- Maryna Kluban, Mohammad Mannan, and Amr Youssef, On Measuring Vulnerable JavaScript Functions in the Wild. In Proceedings of the 2022 ACM Asia Conference on Computer and

Communications Security (ASIA CCS '22), May 30–June 3, 2022, Nagasaki, Japan. ACM, New York, NY, USA, 14 pages [27].

1.5 Thesis organization

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of background information and relevant work, containing the description of previous approaches to vulnerability detection in source code, including specifically for JavaScript; the explanation and existing studies of several JavaScript vulnerability types; previous attempts to develop a vulnerable JavaScript dataset. In Chapter 3, we firstly describe the process and the results of collecting and filtering our own dataset of JavaScript vulnerable functions. Secondly, we present our vulnerability detection algorithms, which include semantic pattern search, textual similarity analysis, and static taint analysis, as well as the process of gathering a dataset of real-world JavaScript functions. In Chapter 4, we discuss the results of running our experiments on the real-world functions and the manual verification of results; additionally, we perform 20 case studies of our findings and the process of responsible disclosure to the project owners. Finally, in Chapter 5, we present our conclusions and discuss limitations and future work.

Chapter 2

Background

In this chapter, we provide background on vulnerability detection in source code. We first describe the related work that targets vulnerabilities specifically in JavaScript. Then we present general information on different vulnerability detection methods. Besides, we overview two specific vulnerability types in JavaScript and discuss relevant studies. Finally, we review the existing work on the compilation of JavaScript vulnerability datasets.

2.1 JavaScript vulnerability detection

Most past work on JavaScript uses package-level vulnerability detection approaches that mostly target metadata on NPM packages [9, 10, 12, 13, 14, 15, 16]. There are several limitations to the metadata analysis approach. First, projects other than NPM packages are not covered by these studies. Second, the metadata for many packages is unreliable or even missing. Other studies have also shown that the metadata approach introduces a lot of false positives, because package-level vulnerability detection is too general, and most of the projects that use vulnerable packages are not exposed to the threat after all [11].

There are a few studies that leverage code-based vulnerability detection methods for JavaScript. Ferenc et al. [17] use machine learning algorithms to recognize vulnerable functions by processing their static code metrics, calculated by static analyzers (number of code lines, code complexity, nesting level, etc.) The best performance result showed an F1-measure of 0.7. The authors concluded

that static source code metrics need to be combined with other metrics to improve the results.

Mosolygó et al. [20] use code lines as targets, and develop a methodology to calculate the probability of the function being vulnerable based on a vector representation of tokenized code lines. This approach, in the best-case scenario when it detects all truly vulnerable functions, produces a lot of false positives (out of 228 flagged lines, only 6 were true positives). Apart from NPM packages, some studies also analyze JavaScript security in Chrome web extensions [28] and scripts from websites [29, 30]. Saxena et al. [31] develop a symbolic execution framework and test it on web applications. NPM packages are generally well-maintained (e.g., in terms of updates and timely security fixes), which is perhaps not true for other JavaScript projects (e.g., due to the lack of centralized package management systems). We target JavaScript code from NPM packages, Chrome web extensions, and top websites, and instead of package/project level analysis we focus on actual vulnerable functions alone. After this step, we perform further experiments to discover if the project, containing found vulnerable function, is affected as well.

2.2 Vulnerable code detection methods

Vulnerability detection in source code is an active research area, and most existing work can be divided into two main categories: static and dynamic code analysis [32]. In this section, we will describe the main techniques used in both categories of analysis, as well as the "taint analysis" technique, which may belong to either of the above categories.

2.2.1 Static analysis

Static source code vulnerability analysis examines the code without running it. This approach has more coverage than the dynamic one, as it can find vulnerable code regardless of the execution conditions. Moreover, static analysis is usually more efficient and therefore more suitable for analyzing a large amount of data. On the other hand, static vulnerability detection tools are more likely to produce higher false positive rates, as they do not actually follow the program logic, therefore omitting unpredictable code paths that may mitigate the vulnerability [33]. We divide static analysis

Clone Type	Description
Type 1	The clones are exact copies, except the comments and space symbols.
Type 2	Type 1 differences, extended with renamed variables, literals.
Type 3	Type 2 differences, extended with the addition, deletion or modification of one or more statements.
Type 4	The clones have the same functionality (semantics), but are implemented differently.

Table 2.1: Types of function clones.

methods into textual similarity (working with syntactical structures of the code, finding new vulnerable syntax based on records of previously discovered vulnerable code) and semantic similarity (using previously discovered vulnerable code to extract vulnerable behavior and develop patterns of this behavior in an abstracted representation).

2.2.1.1 Textual similarity methods

These methods, also known as clone detection methods, can be used at 4 main levels of granularity: file level [34, 35, 36], function-level [37, 30, 38, 39, 40], line-level [41, 42] and token-level [43, 44]. The main principle of the textual similarity methods is searching for matches between the examined real-world code instances and a vulnerable code dataset. Based on the degree of similarity, the code clones are commonly divided into 4 types, described in the Table 2.1.

To be able to detect Type 1-2 clones (different layout, renamed variables), certain transformations are applied to each instance. For example, some studies use code tokenization, and afterwards compare bags-of-tokens to find similarities [40, 43, 45]. To make the search more efficient, some studies use cryptographic hash or vectorization (e.g., with Word2Vec) on the tokens [41, 46, 47]. Apart from tokenization, several other representations are also used in textual similarity approaches, such as Abstract Syntax Trees (AST) [35, 48, 29], Control Flow Graphs (CFG) [49], Program Dependency Graphs (PDG) [50], code binaries [51], or a combination of these representations [42]. In order to detect Type-3 clones, which have more significant differences (e.g., added/deleted/changed statements), some algorithms use locality-sensitive hashing [35, 29] on tokens, but unfortunately, such approaches introduce more false positives (compared to the methods described above) and cannot distinguish a vulnerable function from a patched one. This limitation is also applicable

to ML(Machine Learning)-based approaches [37, 52, 38], which create signatures for functions based on their syntactic features, and then compare the signatures between vulnerable and targeted functions. textual similarity tools are not able to cover Type-4 clones, as there are no syntactical similarities to detect.

Several textual similarity tools aim to detect vulnerable code regardless of the language. To be able to do that, they require any language to be brought into an abstracted representation, such as a high-level tree-based representation [53], PDG and CFG [37, 46, 42]. However, cross-language tools are only implemented for textual similarity search, so they can only detect nearly identical copies of the code. For significantly modified code, semantic-based methods are used. Since every programming language has its own semantics, there is no way of generalizing an approach for vulnerability detection in code for multiple languages.

2.2.1.2 Semantic similarity methods

These methods detect semantic (functional) similarities of Type-4 clones by searching for vulnerability patterns and can be divided into two categories: manual and automated pattern development. Compared to textual similarity approaches, these methods are generally better suited to functions with significant implementation differences. Manually created vulnerable patterns are developed by researchers based on their expertise. As an example, Yamaguchi et al. [54] model templates for several vulnerability types in C/C++ programming languages, such as buffer overflow and memory disclosure, by combining multiple function representations into a Code Property Graph (CPG) and extracting properties that form a vulnerable pattern. Another work by the same authors uses AST to extrapolate vulnerabilities [55]. Li et al. [56] uses their OPG to create some of the common prototype pollution vulnerable patterns. Note that since their representation is based on the JavaScript object features, the patterns can only be created to object-related vulnerabilities. Li et al. [57] also provide an ODG representation that allows creating patterns for several non-object-related vulnerabilities (e.g., XSS, code injection, path traversal).

Following a similar concept, GitHub developed the CodeQL engine [58]. The authors and community contributors maintain a list of vulnerable patterns for multiple languages (including

JavaScript), written on a specially-designed object-oriented query language called QL. Alternatively, the user can create vulnerable patterns themselves. The users need to use CodeQL's abstraction algorithms on their targeted code, and then run the resulting abstract code representation across the set of patterns. These methods are precise and produce few false positives. However, they only capture specific vulnerability implementations, for which the patterns are created. Additionally, the CodeQL pattern development process is designed for those with significant program analysis expertise, thus limiting its usability for non-expert users.

As part of our work, we manually develop vulnerable patterns as well. Since the algorithm for creating all function representations that are required to form a Code Property Graph (as in [54]) is non-trivial, we apply another simpler, yet effective, approach. More precisely, we utilize Semgrep – a static analysis tool, which allows us to develop patterns with regard to the language semantics.

Automatically extracted vulnerable patterns are usually the result of machine learning algorithms. Certain features, which are supposed to make a function vulnerable, are extracted by analyzing a large set of known vulnerable functions [52, 59, 39, 38, 60]. While these tools aim to spare researchers time and effort, we did not choose to proceed with machine learning approaches: ML/DL (Deep Learning) algorithms have to rely on a large dataset of vulnerable and patched functions with clear ground truth which, to the extent of our knowledge, does not exist for JavaScript.

2.2.2 Dynamic analysis

Dynamic code analysis examines the application during (or after) the execution. Tools that use dynamic analysis for vulnerability detection are given specific rules and inputs for the code to run with, similarly to unit testing (the practice of using the designated language to create scripts that perform execution and analysis of the targeted program and report the results based on correlation with the provided expected behavior). Dynamic analysis is usually more precise than the static one. On the other hand, this approach usually has such disadvantages as low code coverage, poor scalability (the tool cannot be generalized for multiple languages and used on a large number of projects), low speed, the requirement for professional expertise in interpreting the results, the requirement for an executable project. The methods of dynamic analysis commonly used to detect vulnerabilities in code are symbolic execution, concolic testing, and fuzzing.

2.2.2.1 Symbolic execution

This method of dynamic analysis simulates program execution by replacing concrete values with symbolic variables for inputs. The program is executed several times, each time modifying the symbolic inputs to reach the execution of new paths. It has been used for finding concrete vulnerabilities in code, such as file parsing vulnerability [61], DoS [62, 63], code injection [64], authentication bypass [65] etc. Saxena et al. [31] use symbolic execution for JavaScript to find client-side code injection vulnerabilities. The disadvantages of this approach are poor scalability (number of paths grows exponentially, causing the path explosion problem), limitations when working with aliases (different pointers to the same location in memory), arrays (they may be treated as one value or as separate values, both cases lead to difficulties / errors) and third-party components (e.g., libraries, kernels when system calls are performed in the code).

2.2.2.2 Concolic testing

Concolic execution (as a mix of "concrete" and "symbolic" executions) is basically a symbolic execution technique, but the generated inputs are concrete values, instead of symbolic ones. This approach was used to detect buffer overflow vulnerabilities [66], DoS [67], null point dereference vulnerabilities [68], etc. To the best of our knowledge, there is no existing work on vulnerability detection in JavaScript using concolic testing. The limitation of the concolic approach is poor code coverage, as there is a high possibility of avoiding the exploration of different paths.

2.2.2.3 Fuzz testing

Fuzz testing executes the program with self-generated input, which may contain knowingly invalid, unexpected or random values. The analysis tool then monitors the behavior of the program and reports undesired results (data leaks, unexpected modifications, program crashes, etc.). The objective is to generate an input that exploits vulnerabilities in the program. Fuzzers are most commonly used for DoS detection [69, 70], but there is also work on detecting Cross-site Scripting (XSS) [71], buffer overflow [72]. Several works developed fuzzing algorithms to find vulnerabilities in JavaScript [73, 74]. Fuzzing is one of the most common dynamic analysis techniques used

in vulnerability detection [75]. The main disadvantage that is special to fuzzing (as compared to other dynamic analysis approaches) is the difficulty of automatically generating useful inputs. For example, the CodeAlchemist fuzzer [74] can generate more than 40% semantically invalid JavaScript inputs [73].

2.2.3 Taint analysis

Taint analysis is a type of dataflow analysis that examines the path of input data through the program. Specifically, inputs from defined untrusted sources (i.e. inputs that are / can be controlled by the user) are tainted and followed from the source (an initial place in code, where the input was introduced) to the sink (potentially vulnerable functionality that can be exploited if the input is malicious). This type of analysis is special in the sense that its objective is to define whether a function containing a potential exploit is reachable by an attacker. Therefore this approach requires correct identifications of "vulnerable sinks". Taint analysis can be implemented both statically and dynamically.

Static taint analysis works with a certain representation of the code. Usually, the code is converted into an abstract structure that contains information on the data nodes and their relationships.Li et al. [56] introduce Object Property Graph (OPG) that tracks tainted object values to detect specifically prototype pollution vulnerabilities in JavaScript. In the following work, Li et al. [57] present Object Dependency Graph representation and the extended framework to model and detect various JavaScript vulnerability types. The WALA framework developed by IBM uses Abstract Syntax Trees (AST) together with call graphs [76]. Initially the framework was developed only for Java, but later the developers introduced JavaScript support. Unfortunately, when we tried to utilize the framework, we found that their JavaScript normalizer does not convert some of the language's features properly (e.g., *this* keyword, and new features in ECMAScript 6 such as arrow functions, and block-scoped variables), which results in the wrong representation of the code flow. Joern [77] combines ASTs with Code Property Graphs (CPG), a representation introduced by Yamaguchi et al. [54]. This tool also supports JavaScript. However, we found that it also fails to process several JavaScript language features correctly (e.g., the *module.exports* syntax, "first-class function" concept). Due to this limitation, Joern currently does not support multi-file taint tracking for JavaScript.

The CodeQL team presents Dataflow Graphs (DFG) [78] for taint analysis.These graphs are based on the language-agnostic abstract code representation (called CodeQL database), developed by the authors, and therefore can be applied to JavaScript code. As a comparison, our framework uses the traditional AST representations combined with an FDG - a graph structure that allows multi-file taint tracking due to the correct processing of file relationships in JavaScript.

The static taint analysis approach is applied in this thesis as a secondary vulnerability detection tool. After identifying vulnerable functions in NPM packages and marking them as sinks we proceed with analyzing the relevant projects to determine whether the found vulnerability can be exploited by an attacker.

In dynamic taint analysis, the tool works like a selective debugger: the analyzed program is executed step by step. For each step, the tool records the transformations of the tainted input value. After that, it reports whether the value is still considered tainted (based on policies that are defined manually, e.g., if the value was not converted to the specific data type or heavily sanitized) and whether it reached a specific place in the program, marked as a sink. Several studies conduct dynamic taint analysis on JavaScript [79, 80], however few of them do it for the purpose of vulnerability detection [81]. Dynamic taint analysis shares the disadvantages with other dynamic approaches (i.e. low coverage, high complexity, time- and resource-consuming).

2.3 JavaScript vulnerability types

This section describes related work that focuses on detecting specific vulnerability types in JavaScript. As of July 2021, the most frequent vulnerability reports in the Snyk vulnerability database [23] belong to 4 vulnerability groups: cross-site scripting, code injection, prototype pollution, and Regular Expression Denial of Service (ReDoS) (see the distribution of reports among vulnerability types in Table 3.2). In our research we focus on prototype pollution and ReDoS, therefore we will overview the background and related work dedicated to detecting these 2 vulnerability groups.

2.3.1 Prototype Pollution

This vulnerability occurs in JavaScript when the "reserved" object keys are reassigned. In JavaScript, all data types are essentially objects (including functions and primitives). All objects have common "root" properties, which are `__proto__`, `constructor` (for objects created using the "new" operator, e.g. `new Date()`), and `prototype` for function objects. The attacker manipulates these properties by tampering with their values. Once one of the root properties is changed for one object, it is changed for all JavaScript objects in a running application, including those created after property tampering. The prototype pollution attack occurs when the objects receive properties and/or values that they are not designed to have. For example, a common way to represent a user on the server side of the web application is in an object with a following structure:

```
{``name'': ``Jane Doe'', ``age'': 30,
    ``education'': {``primary'': true, ``secondary'': false}}
```

To change their education information, a user sends a request with the following information:

```
    {``education'': {``secondary'': true}}
```

This data then gets recursively merged into the user's record. If the functionality that performs the merging operation does not check for the validity of the received data, the attacker can send a forged request, for example:

```
    {``__proto__'': {``isAdmin'': true}}
```

When merging object properties, the program performs the following assignment operation: `userID.__proto__.isAdmin = true`. As a result, all users in the system will inherit the isAdmin property by default, which grants them full access to the system. Depending on the implementation, prototype pollution vulnerability can cause several attacks types, including cross-site scripting, remote code execution, denial of service, and SQL injections [82].

To avoid prototype pollution attacks, modification of an object's root keys should be prohibited. This can be done while creating the object: by "freezing" it so that it becomes immutable, by using `Object.create(null)` which replaces the object's prototype with `null` etc. In addition, the developers can include explicit checks of the object properties and performed operations, where the property names equal to `__proto__`, `constructor` and `prototype`.

This vulnerability was first identified by Snyk researchers in 2018 in a popular NPM package "Lodash" [82]. Kim et al. [83] released their work 3 years after, claiming to be the first to detect prototype pollution with pattern recognition. In their approach, they utilize ASTs and CFGs for code representation. However, their approach produces a large number of false positives (50.6%) and false negatives (84.6%). The first problem is due to the fact that both of their patterns are rather generic and frequently flag non-vulnerable code. Additionally, the patterns do not account for the presence of protection measures in the source code. The latter problem is due to the low coverage of developed patterns: they do not handle recursive calls, object lookups via aliases, etc.

Later, Li et al. [56] presented a novel approach called *ObjLupAnsys* for detecting prototype pollution in NPM packages with a static taint analysis algorithm based on their own representation called Object Property Graph (OPG). However, ObjLupAnsys' false positive rate is relatively high, as it cannot confirm if the found paths between "sources" and "sinks" are valid. It also suffers from poor scalability, as large projects may lead to the path explosion problem. In the following study, Kang et al. [84] introduce the *ProbeTheProto* tool that targets websites. The tool applies dynamic taint analysis and introduces "joint taint flows". Here, the algorithm first traverses from a "source", which has to match a predefined set of code samples (e.g., `document.location`, `location.search`), to a "sink", which is a concrete prototype pollution syntax (meaning that out of all possible syntactical and semantic implementations of prototype pollution, the tool focuses on one specific implementation). Then it traverses from a found "sink" to a "secondary sink", which has to match a predefined set of potential exploit code samples (e.g., `eval`, `innerHTML`). This approach is very precise but has a low recall, in addition to being time- and resource-intensive. A similar approach *SilentSpring* is developed by Shcherbakov et al. [85] to detect prototype pollution vulnerabilities that lead to remote code execution. The authors use a combination of static and dynamic analysis techniques, as opposed to *ProbeTheProto*, and identify 11 "secondary sinks" that

they refer to as "code gadgets". Additionally, unlike *ProbeTheProto*, Shcherbakov et al. [85] target NPM packages. *SilentSpring* focuses on achieving high recall (75%-96%), but unfortunately results in poor precision (23%-27%).

2.3.2 Regular Expression Denial of Service

The second vulnerability type that we develop patterns for is ReDoS. It is a specific case of a Denial of Service (DoS) attack that happens when a program runs a user's input through an evil regular expression [86], which takes exponential time to process specially-crafted complex strings. This usually happens when operators in the regular expression are used in a particular combination. For example, due to the irresponsible use of the repetition operator `'+'` a regular expression `^((([a-z])+.)+[A-Z]([a-z])+$` will result in a long execution loop, which may cause denial of service, if ran on the input `'aaaaaaaaaaaaaaaaaaaaaaaa!'` [86]. The protection measures for ReDoS are either sanitizing the input (i.e., limiting the length or discarding repetition patterns), or rewriting regular expressions without using "dangerous" combinations of operators (such as '+' and '*', '?' and '¿', '?' and '='). There are several tools [87, 88, 89] that can analyze regular expressions and flag the dangerous ones in a given codebase.

ReDoS detection approaches require confidence in the decision on the level of maliciousness of the regular expression. Therefore, studies are mostly focused on creating exploits / attacks on regular expressions using static [90], dynamic [91] and hybrid [92] approaches. Static and dynamic approaches suffer from either low precision or low recall. In ReDoSHunter [92], the authors achieve high precision and recall rates (100% in both), although it detects only most common regex extensions and characters and does not consider less frequently used ones.

2.4 Vulnerability datasets

To create vulnerability datasets, several past studies collect meta-information, packages, and functions/code-snippets from known/reported vulnerabilities. For example, VulData7 [93] uses the National Vulnerability Database (NVD) to extract information on vulnerabilities. However, it does not extract the code from files.

Ferenc et al. [17] use the GitHub repositories of the Snyk vulnerability database [94] and the Node Security Platform (NSP [95]) as vulnerability sources, and create the first publicly available JavaScript dataset consisting of 12,125 functions, 1,496 of which are flagged as vulnerable. However, the Snyk database on GitHub that they used has not been maintained since 2018, and NSP has been made private by NPM and thus is no longer publicly available.

Mosolygó et al. [20] worked with vulnerable functions from dataset in [17]. They manually filtered all 1,496 vulnerable functions and extracted only 443 functions, which they deemed to be actually vulnerable. However, when examining the resulting filtered set of functions, we noticed that it still contained some false positives. This lack of a reliable vulnerability dataset is the primary motivation for our work. We use up-to-date and well-maintained vulnerability sources (Snyk [23] and VulnCode-DB project [24]), and design a comprehensive methodology to avoid flagging unrelated or bug-free functions as vulnerable.

Another popular approach for dataset compilation is to perform a search through GitHub projects, find commits that include specific keywords in the commit message, such as “bug”, “fix”, “CVE” etc. and extract functions from commits [46, 45, 46, 42]. This method relies on developers’ use of proper conventions when they give a name to their commits;[1] as such, only vulnerabilities with correctly formulated commit messages can be detected.

[1] https://www.conventionalcommits.org/en/v1.0.0/

Chapter 3

Methodology

There are different granularity levels for vulnerability detection: line-level, function-level, file-level, and package-level. We choose to work with function-level as this code block has sufficient information on functionality, unlike separate lines of code, and is more likely to be correctly identified in other projects compared to an entire file (with many functions) containing a single vulnerability inside.

To detect vulnerable JavaScript functions in the wild, we first need a dataset of such functions. Since we are unable to find a suitable dataset, we develop one, relying on the existing approaches for function collection, and developing our own methodology to verify the dataset. We also collect a large number of JavaScript functions from active real-world projects for measuring the prevalence of vulnerable functions in those projects.

We group our work into 2 major steps. The first step is the collection of JavaScript functions for creating the vulnerable function dataset, followed by refinement and verification of the dataset. This step is discussed in Section 3.1. The second step consists of the collection and preparation of the target dataset of real-world functions, as well as the implementation of our JavaScript vulnerability detection and verification framework. The second step is described in Section 3.2.

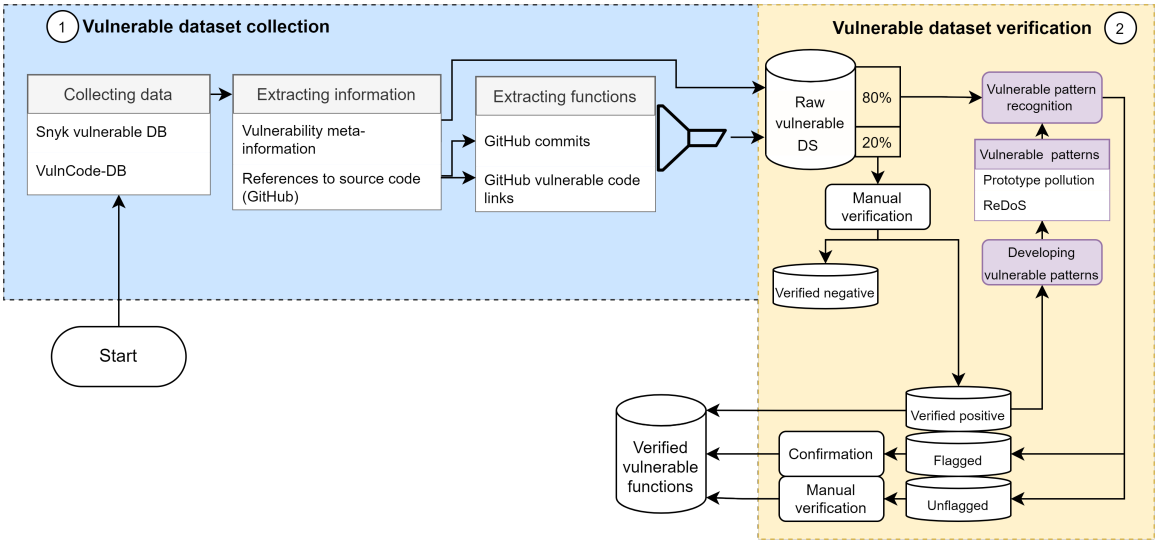

Figure 3.1: Vulnerable dataset compilation approach.

3.1 Vulnerable function dataset preparation

In this section, we describe the process of creating our vulnerable functions dataset. First, we collect possibly vulnerable functions from major vulnerability sources. We then analyze the collected functions and develop an approach to assist us in vulnerability verification. Finally, we perform a semi-automated filtering step to distinguish the truly vulnerable functions. Our approach is depicted on Figure 3.1.

3.1.1 Collecting possibly vulnerable functions

We use two sources for vulnerable functions collection: Snyk vulnerability database [23] and Google VulnCode-DB project [24].

Snyk database gathers information from other vulnerability databases (e.g., NVD, CVE), from threat intelligence systems, research and developer communities, and from scanning multiple platforms by the Snyk security team. Then each entry is manually verified and enriched by Snyk, and published on their website [23]. Snyk collects data for various programming languages. For JavaScript, they store vulnerabilities from NPM packages only, even though their sources for vulnerability collecting include other types of projects [96]. While having a massive codebase (350,000 packages), the NPM repository is only one part of a massive JavaScript ecosystem, hence we reach

out to an additional JavaScript vulnerability source that collects vulnerabilities from other environments. Like Snyk, VulnCode-DB also gathers data from CVE and NVD but additionally includes projects beyond NPM packages. VulnCode-DB also relies on the community to add relevant information for each vulnerability [24], as well as actual vulnerable code examples.

An entry from either of the vulnerability sources contains the following: vulnerability scores in several categories (difficulty, impact, scope); CVE (common vulnerabilities and exposures [26]) identifier; CWE (common weakness enumeration [97]) identifier; affected project name, version and a short description; remediation actions; references; and other relevant information, e.g., detailed description of the vulnerability, and proof of concept attacks.

We develop a JavaScript program to automatically collect available JavaScript vulnerability entries from Snyk and VulnCode-DB websites. We only collect entries that have at least one link under the "References" section, because later we extract source code from the provided links.

As of April 2021, the Snyk database has 2,975 entries under the category "NPM", 2,810 of which have at least one reference link. In the VulnCode-DB project, there are 3,680 entries, 201 of which are for JavaScript projects with at least one reference link.

3.1.1.1 Function extraction and preliminary filtering.

To extract functions for our dataset, we first collect all the links that are provided for each entry of our vulnerability sources and isolate the links that lead to the source code. We also collect detailed metadata on each vulnerability. Out of the 3,011 collected entries with links, only four entries overlapped between Snyk and VulnCode-DB.

We then sort the reference links for each vulnerable entry into multiple categories. If an entry has multiple references, we rank them by priority, based on our assumption that some categories are more useful and convenient for our purposes than the others (links directly leading to the vulnerable source code), and save the link with the highest rank. Other links in the same entry are discarded. The distribution of the links by category is presented in Table 3.1.

For function extraction, we took two categories of links into consideration: GitHub commits and GitHub vulnerable code (50.3% of all links). Links of the first category point to the GitHub "diff" page, where all code changes in the given commit are displayed with the vulnerable files on the left

Category	# of links	%
GitHub commits	1291	44.43%
Advisories	474	16.31%
GitHub issues	244	8.40%
Pull requests	201	6.92%
Vulnerable code	170	5.85%
Bug reports	123	4.23%
Other links	403	13.87%
Total	2906	100%

Table 3.1: Categories of collected links.

and patched files on the right. Links of the second category point to a specific vulnerable file and include a range of lines that contain vulnerable code. The function extraction process from GitHub commits and GitHub vulnerable code is straightforward, while other categories of the links either do not contain the source code or have too many unrelated code parts that cannot be automatically filtered (e.g., pull requests links). We extract functions from GitHub commits as follows. First, for each commit link we use GitHub REST API to get vulnerable and patched versions of the committed files, along with positions of code lines that were modified (added, deleted, or changed). Secondly, we parse both versions of files with `espree` [98] and receive the range of each function (first and last symbol positions in the string). Lastly, we extract all functions with modified lines within their range. We also collect nested functions that include modified lines.

Links of GitHub's "vulnerable code" type contain a modified line range at the end, in the format "#Li-j", where `i` is the first affected line, and `j` – the last. We save the modified line range and retrieve code from the same link using a GitHub REST API. We then repeat the procedure of function extraction same as for commit links.

The structure of each entry in our dataset is presented in Figure 3.2. For each vulnerability entry, we place the affected files in the "files" property (see Figure 3.2). If the link category is "GitHub commit", we include references to both vulnerable and fixed files ("link" and "fixedLink" properties) and a list of extracted functions in "vulnerable-fixed" pairs ("affectedFunctions" property). For "vulnerable code" links, we just add a vulnerable link to the "files" property, and vulnerable functions in the "affectedFunctions" property.

After collecting these possible vulnerability entries, we perform preliminary filtering to remove

```
{
    "link": "GitHub link with source code",
    "name": "category of the link (Commit, Pull Request, etc.)",
    "page": "vulnerability entry in Snyk / VulnCode-DB",
    "CVE": "CVE identifier",
    "CWE": "CWE identifier",
    "packageName": "affected package / project",
    "versions": "affected versions of packages / projects",
    "files": [
        {
        "link": "link to a file with vulnerable code",
        "fixedLink": "link to a file with fixed code",
        "affectedFunctions": [
                "a list of functions in pairs vulnerable-fixed"
            ]
        }, "..."
    ],
    "errors": "files that could not be processed",
    "details": "paragraph of detailed description of vulnerability",
    "vulnType": "category of a vulnerability"
}
```

Figure 3.2: Structure of an entry in our vulnerable function dataset.

the following:

(1) test files (links to such files contained "spec.js" or "test" keywords);

(2) files that do not contain JavaScript functions;

(3) empty functions (with no body); and

(4) cases where the code snippets for both the vulnerable and fixed functions were identical.[2]

After filtering, the number of functions from Snyk and VulnCode-DB "GitHub commit" links reduced to 4,288 (from 9,552) and 184 (from 538), respectively; "vulnerable code" functions remained unaffected (169). At this point, our dataset contains 4,870 functions (from 895 entries). We also group entries by the vulnerability type. Note that we clustered 116 vulnerability types into 25 generalized groups (e.g. Code injection group includes SQL, template, hash, content injection, etc.; Procedure bypass group includes sandbox, signature, authentication bypass, etc.). The most frequently reported vulnerabilities are cross-site scripting (228 entries), command injection (167

[2]This happened when, even though the range of the function was overlapped with the range of affected lines, modified parts belonged to a different function – such as where one function ends and another one begins on the same line.

entries), regular expression denial of service (ReDoS, 121 entries), and prototype pollution (101 entries); see Table 3.2.

Following a similar approach, Ferenc et al. [17] produced a dataset of 1,456 vulnerable functions; however, at least 1,013 of these functions are in fact not vulnerable [20]. We identify two main reasons for this: (i) not considering cases with identical vulnerable and fixed functions (item (4) above); and (ii) not excluding the non-vulnerable functions from commits, where a given commit includes patched functions along with several unrelated/new functions. The second case is difficult to resolve automatically as it requires a clear distinction of vulnerable functions from the rest, which we address using a semi-automated verification step (Sec. 3.1.3).

Vulnerability type	No. of entries	No. of functions
Cross-Site Scripting	228	1183
Code Injection	167	983
ReDoS	121	582
Prototype Pollution	101	356
Denial of Service	45	245
Directory Traversal	42	300
Information Exposure	23	201
Insecure Download Protocol	19	35
Improper Input Validation	18	50
Request Forgery	17	100
Memory Exposure	13	54
Insecure File Access	13	121
Procedure Bypass	12	316
Improper Auth	8	79
Insecure Defaults	7	26
Improper Cred. Protection	7	11
Timing Attack	6	19
Open Redirect	6	10
Insecure Randomness	6	15
Improper Access Control	6	18
Man In The Middle	4	7
Token Disclosure	3	8
Curve Attack	2	33
Buffer Overflow	2	23
Other	19	95
Total	895	4870

Table 3.2: Distribution of vulnerability types among all collected entries.

3.1.2 Manual verification of vulnerable functions

To perform further verification we develop a web application that makes manual verification faster and easier. We upload our collected data to the web interface, which allows us to easily navigate between entries, files, and functions (see Figure 3.3 in the Appendix). For each function, the objective is to make a decision, on whether the function is vulnerable or not. For that, we examine the vulnerability description and the differences between vulnerable and fixed functions. If necessary, we also check Snyk and GitHub for additional information (sometimes other technical sources), to make a concrete decision for each entry.

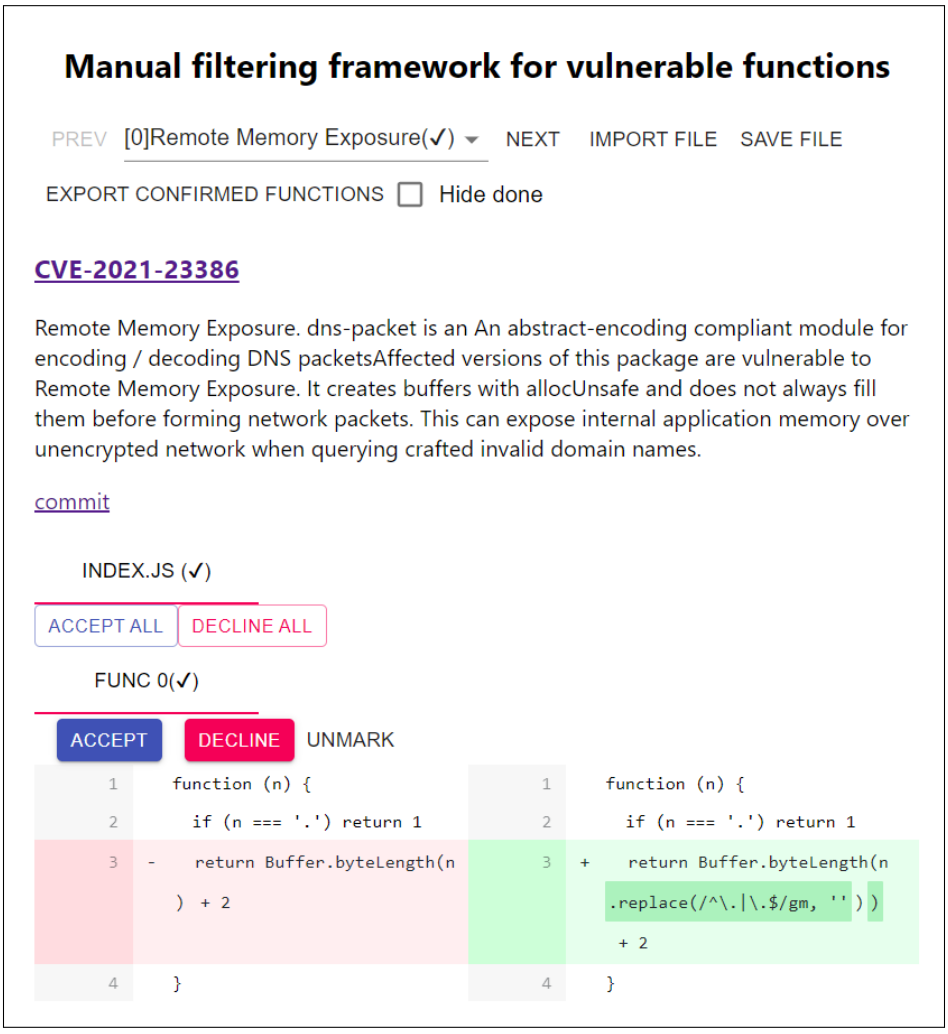

Figure 3.3: GUI of the framework for manual vulnerable functions verification.

With help of our tool, we manually analyzed 150 vulnerability entries (∼17% of the collected

data) and found that, while some vulnerability types do not have any regularities and each case is very specific to each project, others are often implemented in the same way. Based on this observation, we implement a pattern-based detection approach for the functions that follow specific patterns. For these functions, the manual verification process involves less scrutiny than for other functions.

3.1.3 Semi-automated function verification

To improve the efficiency of the verification of vulnerabilities in our collected functions we develop an approach that uses vulnerable pattern search to detect functions of certain vulnerability types. For that, we utilize a static analysis tool Semgrep [21], which performs an advanced semantic search in code based on provided patterns. The advantage of Semgrep is that it can understand variables and structures unlike a simple grep search. Thus Semgrep can also detect patterns in minified code.

3.1.3.1 Semgrep rule development for selected vulnerability types.

Semgrep search is performed with rule sets, provided in `.yaml` format. Each rule defines which patterns the tool should look for, which to dismiss, and whether the target is inside of a specific structure or not. There are multiple open-source rule sets for many programming languages developed by Semgrep authors, as well as by the community. Most of them are written to find common bugs and inconsistencies in the code, but some rule sets also target vulnerabilities.

We performed a Semgrep search with the available community rules for JavaScript on our dataset, but unfortunately, it produced no matches. To understand the reason, we examined several patterns from those rule sets and reached several conclusions. Some rules, while covering a certain vulnerability type, come with patterns that are too specific. For example, a pattern requiring two code lines to be placed together would not be triggered if other code separates them. In other situations, our functions are simply not covered by the rules.

Therefore, we decided to develop our own rules for Semgrep pattern search. To write a Semgrep rule, we need to create a pattern for a line (or lines) of code that we want to match. Semgrep also allows to add conditions like: `pattern-not`, `pattern-inside`, `pattern-not-inside` etc.

to make rules more targeted and avoid false positives. In the end, we add meta-information to each rule to describe the pattern. To develop rules for pattern recognition, we need to clearly understand each targeted vulnerability type and the protection measures against the vulnerability. The inclusion of patterns for preventive measures helps us avoid flagging fixed functions as vulnerable.

We create rule sets for several common vulnerability types, covering as many functions from each vulnerability type as possible. However, to rely on a pattern as an indicator of a vulnerability, we need to consider the context where it appears. This proved challenging for certain vulnerability types, such as cross-site scripting and command injection, where these vulnerabilities can be mitigated by the surrounding context in numerous ways. As such, creating patterns for these vulnerability types is bound to generate many false positives, which we want to avoid. Therefore, we choose to create pattern rules only if the vulnerability has a very specific set of mitigating patterns. Prototype pollution (11.3% of all types) and ReDoS (13.5% of all types) vulnerability types primarily meet our selection patterns.

By understanding the vulnerability and its preventive measures described in Section 2.3.1, we can develop Semgrep rules for prototype pollution. The pattern for this vulnerability is the object key assignment statement, e.g., `object[key] = value`. The key, and possibly the value and the object have to come to the function from the outside (through arguments, global variables, or in another way). Considering that, we add more rule properties to account for the context of the pattern, including mitigating factors. As a result, we created seven rules with different prototype pollution scenarios. A rule example is presented in Figure 3.4, along with an example of two vulnerable functions, targeted by the rule.

To detect ReDoS vulnerability we utilize a tool that analyses regular expressions and can identify them as "evil" or "benign". For this purpose, we choose to use the NPM module `safe-regex` [89]. To extract regular expressions from the functions, we utilize the Abstract Syntax Tree (AST) function representation, and collect nodes representing regular expressions, such as "new RegExp("...")" and "/regex/". Then we execute `safe-regex` check on each regular expression, and save the functions that are flagged.

The presence of the evil regular expression in the code is already potentially dangerous, but

Semgrep rule

```
rules:
- id: prototype_pollution
  patterns:
    - pattern: $SOME_OBJ[$KEY] = ...
    - pattern-inside: |
        function ...(...,$KEYS,...) {
          ...
        }
    - pattern-inside: |
        for ($KEY in $KEYS)
            ...
    - pattern-not-regex: |
        ((__proto__[\s\S]*prototype[\s\S]*
constructor))[\s\S]*
    - pattern-not-regex: |
        Object\.freeze\([\s\S]*
    - pattern-not-regex: |
        Object\.create\(null\)[\s\S]*
  message: loop through keys in object
  severity: WARNING
  languages: [javascript]
```

Function examples

```
function writeConfig(output, key, value, rec
urse) {

  if (isObject(value) && !isArray(value)) {
   o = isObject(output[key]) ? output[key] :
 (output[key] = {});
    for (k in value) {
      if (recurse && (recurse === true
          || recurse[k])) {
        writeConfig(o, k, value[k]);
      } else {
        o[k] = value[k];
      }
    }
  } else {
    output[key] = value;
  }
}
```

```
function _recursiveMerge(base, extend) {
    if (!isPlainObject(base))
        return extend;
    for (var key in extend)
       base[key] = (isPlainObject(base[key])
       && isPlainObject(extend[key])) ?
    _recursiveMerge(base[key], extend[key])
        :
          extend[key];
    return base;
}
```

Figure 3.4: Example of a Semgrep rule for prototype pollution.

we also execute a pattern search to check whether the evil regular expression is applied to a user-supplied string. As a result, we created two Semgrep rules that account for four different ReDoS scenarios. Figure 3.5 shows an example of a ReDoS rule and two different targeted functions.

3.1.4 Final vulnerable functions dataset

With two rule sets, developed for prototype pollution and ReDoS vulnerability types, we executed a Semgrep pattern search on the remaining unverified functions from our vulnerable function dataset. We repeated the search several times, each time modifying the rule sets to match as many true positives as possible while reducing the rate of false positives. We summarize the results in Table 3.3; for each vulnerability type, the table includes the number of vulnerability entries, the number of all functions in this type, how many of them were flagged by our pattern search, and how many from all functions were then manually verified. Iteratively we adjusted our rule sets to not match any false positives in our dataset.

Although we cannot rely on the pattern search completely and all flagged functions still need

Semgrep rule	Function examples
rules: - id: redos1 patterns: - pattern-either: pattern: <...$ARG.$METHOD(/.../,...)...> - pattern: | <...$ARG.$METHOD(new RegExp('...'),...)...> - metavariable-regex: metavariable: $METHOD regex: (match\|replace) - pattern-inside: \| function ...(...,$ARG,...) { ... } - pattern-not-inside: | function ...(...,$ARG,...) { ... if(<...$ARG.length...>) ... } message: direct use of match or replace severity: WARNING languages: [javascript]	function (name, value) { if (name == 'user_host') { return value.replace(/(\[.*?\])+/g, ''); } return value; } function isExternal(url) { let match = url.match(/^([^:/?#]+:)?(?:\/\/([^/?#]*))?([^?#]+)?(\?[^#]*)?(#.*)?/); if (typeof match[1] === 'string' && match[1].length > 0 && match[1! == location.protocol) { return true; } return false; }

Figure 3.5: Example of a Semgrep rule for ReDoS.

	Proto. Pollution	**ReDoS**
No. vulnerability entries	101	121
No. functions	356	582
No. functions flagged	141	68
No. functions manually verified	166	164

Table 3.3: Summary on the vulnerable pattern findings in our vulnerable function dataset.

to be manually confirmed, it does speed up the process of verification. Instead of performing an in-depth analysis of all the available information about a given vulnerability, we simply ensure that the flagged code is not an exception to the patterns.

During the manual verification we observed that in some cases, the "fixed" file version that is supposed to eliminate the vulnerability contains only partial fixes. For example, for the prototype pollution vulnerability the function performs a check for the value `__proto__`, but not for `constructor` or `prototype` that can still be used in a malicious way. An example of the ReDoS partial fix scenario is when one evil regular expression is removed, but the others remain present, or new evil regular expressions are introduced.

We had another unexpected but positive outcome from our manual validation. Since we ran a pattern search on the whole vulnerable function dataset, and not only on the targeted vulnerability types, the other functions were checked too, and matches were found. This means that a function that was reported to Snyk or VulnCode-DB under one vulnerability type also has another potential vulnerability present. In general, Semgrep search with our rule sets matched 195 prototype pollution patterns and 121 evil regular expressions (106 of them have ReDoS patterns) in other entries. This is an important finding, because while developers might fix a specific vulnerability after it is reported, other vulnerable code snippets may remain unchanged.

As a result, from the semi-automated verification step, we confirm 1,360 unique vulnerable functions of 43 different vulnerability type groups. This makes our dataset three times bigger than the dataset created in [20], and the biggest dataset of verified vulnerable JavaScript functions.

3.2 Vulnerability detection

In this section, we discuss the collection and preparation of the target dataset of real-world functions, as well as the implementation of our detection framework, designed to find and verify vulnerable functions. The approach is presented in Figure 3.6.

3.2.1 Dataset of real world functions

To collect functions from real-world projects we choose three sources: NPM packages, Chrome web extensions, and popular websites (as per the Cisco Umbrella Popularity List [99]). In the NPM open-source package registry, packages are mostly written in JavaScript. We extract JavaScript files from GitHub repositories of 3000 most popular NPM packages.[3] These data sources were accessed between January and March of 2021.

The majority of scripts in Chrome web extensions are also written in JavaScript. To collect JavaScript files from extensions we download and unpack the source code for 600 first most popular extensions from Chrome Web Store. In 43 cases, our script was not able to retrieve the source code from Chrome Web Store API, hence only 557 extensions are processed further for our dataset.

[3]We used a list, which sorted packages by the frequency of their usage. https://github.com/nice-registry/all-the-package-names

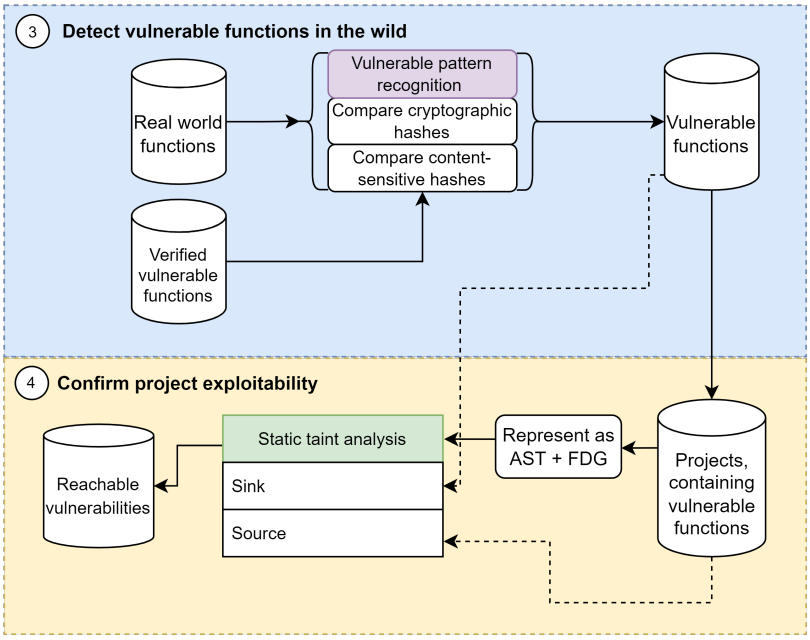

Figure 3.6: Vulnerable function detection and project exploitability verification approach.

From the Cisco Umbrella popularity list, we choose the 20,000 most popular websites. After sending an HTTP request to each website we receive a response with an HTML page. Then we either extract the JavaScript code from the `<script>` tag or save JavaScript files by following the links, defined in the same tag. In 18,108 cases the websites instead returned either a static HTML page with no JavaScript, a response in a different format, or an HTTP error. As a result, we collected JavaScript files for 1,892 websites.

From the crawled JavaScript files we extract all functions. The datasets contain a list of functions, mapped with the file link of the source URL. Finally, we filter all collected data to exclude test files (by searching for "spec.js", "test" keywords in file links) and those functions that either had an empty body or only one statement such as printout to a command line or return. The resulting dataset may contain duplicate functions that belong to unique sources. A summary of the dataset is provided in Table 4.1 in the first 3 columns.

3.2.2 Vulnerable function detection

Our primary approach for detection is vulnerable pattern search. While this procedure is able to detect most of the vulnerable function variations, as long as the vulnerable pattern is present, it is

limited to detecting only the patterns of the vulnerable types that we cover. To extend our detection range to the remaining vulnerabilities from our dataset, we also utilize textual similarity detection methods: content-sensitive fuzzy hashing and cryptographic hashing. Note that these hash-based mechanisms only target near-duplicate functions from the vulnerable dataset, and we use them for completeness.

3.2.2.1 Vulnerable pattern search

For implementing vulnerable pattern search, we use the Semgrep static analysis tool with the rule sets we developed for prototype pollution and ReDoS vulnerability types; note that we also use these rule sets for a semi-automated verification of our vulnerable function dataset (see Section 3.1.3). Our search logic is implemented using JavaScript, which iterates over the real-world functions, and for each of them executes the following steps:

(1) Run Semgrep search with the rule set for prototype pollution, and flag the function if a match is found.

(2) Find and extract regular expressions from the function. If successful, run the `safe-regex` module and flag the function if `safe-regex` fings any evil regular expressions.

(3) Run Semgrep search with the rule sets for ReDoS patterns on functions with evil regular expressions; flag the function if the pattern is matched.

We save all flagged functions, including the projects/files where they are found. Note that for this approach we do not need to refer to the functions of our vulnerability dataset until we want to find new patterns and develop additional rules.

Since our Semgrep rules are biased towards entries of our vulnerable functions dataset, it is reasonable to expect false positive matches among the real-world functions. In order to improve the precision of our approach, we make adjustments to the rules after analyzing around 100 flagged real-world functions (i.e. adding new conditions `pattern-not` and `pattern-not-inside`).

3.2.2.2 Textual-similarity based approaches

We use fuzzy hash and cryptographic hash comparison for vulnerable function detection.

Data abstraction. Before conducting the experiments with textual similarity based methods, we tokenize both vulnerable and real-world functions in order to bring functions to the same generalized format. We apply tokenization representation similar to the approach in [20]. After taking into consideration multiple syntactic parts of JavaScript code, and deciding on their importance and influence on the functionality, we decided to apply the following tokenization rules: (i) remove all space characters and comments; and (ii) rename all variables and arguments to unified format (e.g., varName1, varName2). Note that we leave all punctuators, as assignments or arithmetical operators play a vital role in the meaning of the function. We also do not generalize the primitive variable values, such as strings, numbers, and regular expression patterns; note that we want to account for the variable values, since the difference between numbers 0 and 1 can be crucial for the vulnerability context (or, in the prototype pollution case, we want to check if the values of object keys are checked in the code).

To rename function variables, we parse each function to its AST and recursively process each node (i.e., tree leaf). In each round we look for "Variable declaration" and "Function declaration" nodes, save their position in the function (range), and then locate and replace characters between stored ranges. As for the arguments, we locate them by searching "Function declaration" nodes and looking for identifiers between the parentheses that follow. Then we perform the same renaming procedure as for variables. Figure 3.7 shows an example of a function and its tokenized version.

Function	Tokenized representation
function (n) { if (n === '.') return 1 return Buffer.**byteLength**(n)+2 }	function (varName1) { if (varName1 == = '.') return 1 return Buffer . byteLength (varName1) + 2 }

Figure 3.7: Example of a function and its tokenized version.

Content-sensitive hash comparison. Content-sensitive hashing (also known as fuzzy hashing), creates a fixed-size string that reflects the content of the input. It means that, unlike cryptographic hashing, small changes in the input result in small changes in the hash. It allows more flexibility in the function content, so if a few lines are added, removed, or modified, the content-sensitive hash will still be similar. It enables detecting slightly modified functions but may introduce false

positives or false negatives. For example, this approach cannot distinguish the difference between vulnerable and patched functions, if the patch requires only small changes.

For fuzzy hash comparison we use the `SimHash` [22] Python module. First, we create content-sensitive hashes for all functions of real-world and vulnerable datasets using the `SimHash()` method. Then we utilize the `get_near_dups()` method, which uses the hamming distance to compare the hashes. SimHash allows choosing the similarity threshold, by which it decides whether to flag hashes as near-duplicates. The value of the threshold can vary from 1 (strings are almost identical) to 64 (strings are completely different). To minimize the false positives rate we set the threshold to 1 so that it matches only highly similar functions. Finally, if SimHash finds a match, we save the function from the real-world dataset.

Cryptographic hash comparison. We also create SHA-1 hashes for all tokenized functions from both vulnerable and real-world functions datasets. We then perform a search for matches: for each hash of the tokenized function from the real-world project, we search for the same hash in the vulnerable function dataset. Finally, we save the real-world function and its source link, if a hash match is found.

3.3 Exploitability verification

In this section, we provide details of our design and implementation of a static taint analysis method to verify the exploitability of the flagged vulnerable functions (see Section 3.2.2). Several existing tools offer taint analysis for JavaScript (see e.g., [21, 58, 77]); however as of writing, none of them support multi-file taint tracking—essential for real-world projects, which often consist of several JavaScript files. Our taint analyzer tracks an input variable, verifying that it has a path between a "source" and a "sink" across all the files in a given project. Our current prototype is tailored for vulnerable functions in NPM packages. Support for Chrome Web Extensions and websites can be added in future work, e.g., by leveraging the list of common sources for websites (such as URLs, Document.location, location.search, client-side storage) as defined by Kang et al. [84]. Note that capturing all input sources in extensions and websites comprehensively is non-trivial as there are numerous such options.

Note that we assume that the package is vulnerable if the vulnerable functionality of the package is reachable externally. The vulnerability can be exploited, only if a developer combines obtaining and processing a user's input with a vulnerable functionality in the NPM package. Here the term "user" refers to a person who interacts with the developer's project and enters some data in the context of this interaction. Despite the specific prerequisites for the exploitation, the vulnerabilities in the NPM packages are perceived seriously and often get the highest severity ratings by the National Vulnerability Database assessments(e.g., CVE-2022-21189, CVE-2021-23518).

NPM packages are meant to be included in projects as external dependencies. As such, they contain distinctly marked "exports", which are the only objects accessible to the user. These objects are potential "sources" of user-controlled inputs. An NPM package consists of one or more JavaScript files that interact with each other by exporting and importing functionality, and a "package.json" file that contains package metadata. Among other information, this file contains the top-level export location (or an "index file") that is made accessible to the developer when this package is included in the project. However, there are multiple competing standards and formats to specify export information, and any NPM package owner can choose to implement any number of them. An example of a typical implementation of "package.json" that contains three different export declaration standards is provided in Figure 3.8.

An example of file interaction using input and export statements is demonstrated in Figure 3.9. Given the absence of a single generalized data format in package.json we decided to apply another approach to detect top-level exports in NPM packages. We define the functionality that is exported but never imported in other files as a "source", as it may be accessible to the end-user (depending on how developers choose to use the given package). Therefore, on Figure 3.9 the top-exported function is "file3Func(...)", and hence it is our "source". The input data are all arguments of the function "file3Func()". The "sink" is the variable in the function that was earlier marked as vulnerable.

File Dependency Graph. To address the absence of a single generalized data format in package.json, as well as our need for a multi-file taint tracking, we introduce our File Dependency Graph (FDG) structure. Firstly, a multi-file project is represented in AST, preserving the file structure. From each file's AST we locate nodes that indicate an "import" or an "export" of a certain functionality. For this, our algorithm uses a predefined set of rules, specific to each type of the

```
19    "main": "underscore-umd.js",
20    "module": "modules/index-all.js",
21    "type": "commonjs",
22    "exports": {
23      ".": {
24        "import": {
25          "module": "./modules/index-all.js",
26          "browser": {
27            "production": "./underscore-esm-min.js",
28            "default": "./underscore-esm.js"
29          },
30          "node": "./underscore-node.mjs",
31          "default": "./underscore-esm.js"
32        },
33        "require": {
34          "browser": {
35            "production": "./underscore-umd-min.js",
36            "default": "./underscore-umd.js"
37          },
38          "node": "./underscore-node.cjs",
39          "default": "./underscore-umd.js"
40        },
41        "default": "./underscore-umd.js"
42      },
43      "./underscore*": "./underscore*",
44      "./modules/*": {
45        "require": "./cjs/*",
46        "default": "./modules/*"
47      },
48      "./amd/*": "./amd/*",
49      "./cjs/*": "./cjs/*",
50      "./package.json": "./package.json"
51    },
```

Figure 3.8: export declaration in package.json of “Underscore” NPM package.

import and export nodes (there are six import types, such as “ImportSpecifierNames” and “requireNames” and five export types, such as “ExportAllDeclaration” and “ModuleExportNames”). Each rule provides instructions to extract relevant information: the name of the imported/exported functionality, the origin file, and the node itself.

Secondly, after processing the nodes with the rules we iterate through the selected nodes to find and group the matching pairs: the functionality exported by one file, which is then imported by another file. Note that for each “import” node, there can be only one matching “export” node, whereas for the export nodes we include the array of all corresponding “imports”. If there is an export node,

Figure 3.9: Example of the import-export structure of a project. File file3.js contains top-level export, as it is not imported anywhere else.

for which no matching import is found, the node is marked as top-level, i.e., the exported functionality may be accessible to the user and therefore is our "source". This way we obtain a graph that indicates the relationship between project files, where each node has the information about its matching pair.

The taint approach implementation. The process to verify the exploitability of a vulnerable pattern in an NPM package consists of six steps (the corresponding pseudo-code is laid out in Algorithm 1):

(1) Download the project that contains the detected vulnerable function;

(2) Create an AST representation of the project;

(3) Create an FDG of the project based on its AST and find a "source";

(4) Get the location of the vulnerable "sink" from the Semgrep-based approach results, find the corresponding node in the AST;

(5) Build a "taint graph" that goes over the AST with FDG and taints nodes, which are affected by input data; and

(6) Look for a path in the "taint graph" that includes a "sink" variable. If there is such a path, then the user-controlled input can reach the vulnerable code.

Algorithm 1 Algorithm of taint analysis tool.

```
downloadProject(projectUrl, projectPath + "/original")
astgen.generate(projectPath + "/original", projectPath + "/ast")   ▷ create an AST representation of the project
asts ← []
for all file in (projectPath + "/ast") do   ▷ create an array of generated ASTs
    asts.push(parse(file))
end for   ▷ start: create FDG
allNodes ← []
importNodes ← []
exportNodes ← []
for all ast in asts do
    for all node in ast do
        allNodes.push(node)
        node.file = ast.file   ▷ add file information to each node
        if isImport(node) then
            node.isImport ← true   ▷ mark a node as an "import" node
            node.from ← resolveImportLocation(node)   ▷ add origin to the import node
            importNodes.push(node)
        end if
        if isExport(node) then
            node.isExport ← true   ▷ mark node as an "export" node
            exportNodes.push(node)
        end if
    end for
end for
sources ← []
for all export in exportNodes do
    export.importedBy ← []
    for all import in importNodes do   ▷ using import origin connect export with import
        if import.from = export.file AND import.name = export.name then
            export.importedBy.push(import)
            import.importedFrom ← export
        end if
    end for
    if export.importedBy is empty then   ▷ if found export but no import, mark as "source"
        sources.push(export)
    end if
end for   ▷ end: create FDG
taints ← []
currentLeaves ← sources   ▷ define found sources as an initial "taintor"
while currentLeaves is not empty do   ▷ do while new values are tainted each iteration
    currentLeaves ← propagateTaints(currentLeaves, taints, allNodes)   ▷ this step is described in Algorithm 2
    taints ← taints + currentLeaves   ▷ add tainted nodes to an array
end while
```

Our algorithm starts by taking the resulting dataset of the functions marked as vulnerable by the previous step of our experiment. Each entry of the dataset has a URL link to a file within a project on GitHub. By processing this link we get the information on the name of the project, as well as a specific commit ID. We craft a new link with extracted information and clone a project in our temporary folder using the `git clone` command.

Out of several JavaScript parsers that generate ASTs [98, 100, 101, 102] we found @joernio/astgen [103], created by Joern, to be the most applicable. ASTgen is a multi-language parser that uses Babel [101]. Babel parser efficiently deals with any JavaScript variations, bringing them to one general format when parsed. This parser is constantly adapting to the updates in ECMAScript standards, implementing changes even from *stage 0* proposals [104]. In comparison, Acorn [102] and esprima [100] only support basic JavaScript syntax that is already in the standard and are not adjustable. ASTgen provides additional convenience by preserving the file structure of the project when parsing scripts to AST.

Next, our algorithm reads all AST files into memory and extracts import and export nodes. In this step, we iterate over all nodes and locate imports and exports using predefined rules. For each type of import or export node (the list of such types, as well as examples of their usage in the scripts, is provided in Figure 3.10) we develop a rule to extract relevant information: the name ("var0", "importVar", "exportVar"), origin ("file0.js") and the node itself.

```
import var0 from "./file0.js" //ImportDefaultSpecifier
import * as var0 from "./file0.js" //ImportNamespaceSpecifier
import {var0 as importVar} from "file0"; //ImportSpecifier
const var0 = require("./file0.js") // CallExpression "require"
```

```
export const var0 = 1; //ExportNamedDeclaration
export function var0(Arg) { //ExportNamedDeclaration
export {var0 as exportVar} //ExportSpecifier + from
export default function var0(file3Arg) { //ExportDefaultDeclaration
export * as exportVar from "./file0.js"; //ExportNamespaceSpecifier
export * from "./file0.js"; //ExportAllDeclaration
```

Figure 3.10: Import and export statements and the types of AST nodes they use.

Next, we start an iterative process of data tainting. In the first iteration, we find all nodes in the AST that are tainted by defined "sources" and add them to the accumulator. The newly tainted nodes are then marked as new "sources" for the next iteration. This process is repeated until no new nodes are tainted. Additionally, each tainted node in the accumulator has information about its

"taintor" (a node that caused tainting of the current node).

During the node tainting process, the algorithm first finds all instances of already tainted "source" identifiers. Since every node type in the AST has a different influence on the related nodes, a set of predefined taint rules is required, which regulates taint propagation based on the node type. For example, when a function is called with a tainted argument (e.g., `myFunc(tainted_arg)`), the corresponding argument is tainted in the function declaration. Similarly, if a tainted value is assigned to a variable, the variable becomes tainted as well. The pseudo-code of a single iteration is displayed in Algorithm 2.

Finally, we obtain the resulting accumulator that contains the list of the tainted nodes. We locate the "sink" node in the AST based on the information retrieved from Semgrep or a textual similarity algorithm. Then, to determine if the vulnerability is exploitable, we check if this node is present in the accumulator. If it is found, we can further obtain a full taint path by traversing the "taintor" properties. Later we use the taint path to provide the following information on the exploitability: where the vulnerability is introduced, which input is potentially controlled by the attacker, and where the functionality is exported.

Algorithm 2 Algorithm of one iteration of taint propagation.

```
1: currentLeaves ← arguments[0]                                   ▷ "taintors"
2: allTaints ← arguments[1]                          ▷ array with previously tainted nodes
3: allNodes ← arguments[2]                                        ▷ all AST
4: nodesToPropagate ← currentLeaves
5: for all node in currentLeaves do
6:     sameIdNodes ← findSameId(allNodes, node)                   ▷ nodes with the same name
7:     for all sameIdNode in sameIdNodes do
8:         sameIdNode.taintedBy.push(node)          ▷ add the information about the "taintor"
9:         if sameIdNode not in allTaints then
10:            allTaints.push(sameIdNode)
11:            nodesToPropagate.push(sameIdNode)
12:        end if
13:    end for
14: end for
15: newTaints ← []
16: for all node in nodesToPropagate do
17:    taints ← propagateTaint(node)          ▷ with propagation rules for different nodes
18:    for all taint in taints do
19:        taint.taintedBy ← node                    ▷ add the information about "taintor"
20:        if taint not in allTaints and taint not in newTaints then
21:            newTaints.push(taint)
22:        end if
23:    end for
24: end forreturn newTaints
```

Chapter 4

Results

In this chapter, we first present the results of running our vulnerable function detection approaches. Then we manually validate the findings and provide precision measurements. Next, we provide the results of identifying the exploitable vulnerabilities in NPM packages (including our findings from manual validation). Furthermore, we describe our semi-automated vulnerability reporting process and discuss the overall response dynamics. Section 4.4 is dedicated to case studies.

4.1 Vulnerable function detection results

We perform our textual-similarity tests and pattern recognition on the whole real-world dataset of 9,205,654 functions. First we run Semgrep algorithm. Out of all functions, our Semgrep rules flagged 284,413 potential prototype pollution and 22,496 potential ReDoS vulnerable functions. The distribution of the detected functions among all tested environments is presented in Table 4.1.

Note that sometimes vulnerable pattern search matched multiple functions from the same file to one vulnerable equivalent function. This mostly happened because of the nested functions that have the same vulnerable code. To find out the amount of uniquely occurring vulnerable code that was detected, we create a script, which finds all nested functions and keeps only the child function. As a result, we get 117,601 unique detected functions for prototype pollution pattern and 7,333 functions for ReDoS.

The SimHash algorithm matched 1,320 functions from the whole real-world dataset (Table 4.2).

	# sources	# functions	# prototype pollution		# ReDoS	
			total	unique	total	unique
NPM packages	3,000	413,774	10,338	4,896	1,325	493
Websites	1,892	5,739,271	179,626	77,504	14,586	4,648
Chrome extensions	557	2,659,649	94,449	35,201	6,585	2,192
Total	5,449	9,205,654	284,413	117,601	22,496	7,333

Table 4.1: Results of vulnerable function detection in real-world projects by the Semgrep-based approach.

The vulnerability types of the detected functions are distributed as follows: cross-site scripting (307), ReDoS (306), prototype pollution (138), command injection (133), directory traversal (72), SQL injection (68), denial of service (59), and others (such as arbitrary script injection (16), directory traversal (12)). Similar to the vulnerable pattern search algorithm, SimHash also detected several nested functions. We apply the same filtering script to all detected functions and as a result, we get 965 unique vulnerabilities.

Finally, the cryptographic hash matching algorithm produced 131 matching cases from all real-world functions (Table 4.2). All of these matches were already detected by SimHash. However, since the findings of cryptographic hash matching are guaranteed to be identical (except for variable names) copies of vulnerable functions, there is no need to manually verify them, and we can automatically count them as true positives. The findings belonged to the following vulnerability types: ReDoS (29), cross-site scripting (26), command injection (25), prototype pollution (14), timing attack (5), denial of service (1), and others (such as arbitrary script injection (1), directory traversal (1)).

Note that all ReDoS and prototype pollution vulnerabilities detected by textual similarity methods were also detected by our Semgrep rules. This is expected behavior, as the rules are based on functions from our vulnerability dataset, and SimHash and cryptographic hash are detecting near-duplicate versions of those functions.

As a result, our experiment identifies 125,555 unique functions, detected by at least one method in our framework. 124,934 (by adding up the amount of unique detected functions by Semgrep) of the findings belong to either prototype pollution or ReDoS vulnerabilities, and the remaining 621 to other types of vulnerabilities that appeared in our vulnerable functions dataset.

	NPM packages	Extensions	Websites	**Total**	**Unique**
Fuzzy Hash	56	201	1,063	1,320	965
Crypto Hash	30	85	16	131	131

Table 4.2: Detected vulnerable functions by textual similarity methods (all vulnerability types).

4.2 Manual validation of the results

To evaluate the performance of the vulnerable pattern search, we randomly picked 100 functions for prototype pollution and 100 for ReDoS vulnerability from all the detected functions. For each finding, we examine three main features. Firstly, we confirm that the flagged pattern is detected correctly. Secondly, we verify that the input to the pattern comes from the outside of the function. Lastly, we make sure that there are no sufficient protection measures, missed by our Semgrep rules. As a result, we identified 8 false positives for prototype pollution and 3 for ReDoS, resulting in the precision rate of 92% and 97%, respectively.

Among the prototype pollution functions, patterns got false matches for mainly two reasons. Firstly, in JavaScript the following assignment syntax: `obj[key] = value` is valid for an object and an array (ordered collection of elements). In the case of the assignment to the array element, the "key" is an index of the element. However, the prototype pollution applies only to the object's properties modification. Since JavaScript does not have explicit data types, the patterns fail to distinct array and object assignments. The second reason for false positives for prototype pollution was the function obfuscation technique, which did not modify the vulnerable pattern but heavily obfuscated the protection measures that prevent the vulnerability.

In three false positive matches from ReDoS, the function contained multiple regular expressions. The input variable was matched with some of the regular expressions, but not with the ones, marked as dangerous in the previous steps of our experiment. Since the dangerous regular expression was not used with the user input we do not see a possibility of a ReDoS attack in this scenario.

Since the fuzzy hashing comparison approach may introduce false positives (as described in the previous section), we need to manually verify the findings as well. Therefore, for 90 out of 965 matches we checked the differences between the real-world function and the vulnerable function with similar content. As a result, we identified two false positive matches, which gives us a precision

rate of 98%. There were two reasons for false positives: one function was patched, but the fix was only in one line, so the function remained similar enough; the other flagged function matched ReDoS vulnerable function, but did not have a regular expression in its body (otherwise it was identical to the vulnerable function).

4.3 Exploitable project identification results

Next, we perform our static taint analysis on the findings from Section 4.1. We aim to identify the functions that make corresponding projects exploitable by an attacker. We run this experiment on 5,389 findings in 204 NPM packages flagged as vulnerable. Our tool identified taint paths for 259 prototype pollution findings and 52 ReDoS functions (from 134 NPM packages), reducing the number of flagged functions by 94%. After manual verification of 100 randomly-chosen findings in this experiment, we detected no false positives, meaning that all the identified paths correctly indicate a relation between the vulnerable "sink" and top-level export. We list top-20 NPM packages (as of April 2022) containing such exploitable findings in Table 4.3. The second column in the table shows the number of NPM packages that depend on the vulnerable one. The third column refers to the weekly downloads number of vulnerable packages. In the fourth column, we present the number of findings by Semgrep, and in the last column, we show the number of vulnerable findings that are reachable by the top-level exported functions. Note, that for the "Lodash" package we do not include values in the second and third column, as our findings belong to the master branch of the package, which is different from a branch that is located in the NPM package registry (for detailed explanation see Section 4.4 about Lodash).

4.3.1 Vulnerability reporting

From our taint analysis, we obtain vulnerability entries that contain the following details: the taint path, the detected vulnerability type, the "sink" variable, the link to the targeted project, and the code itself.

Before reporting the findings, we need to make sure that they were not disclosed earlier in the MITRE CVE database [26]. We develop a script that automatically compares project vendors and

NPM package	# depended projects	# weekly downloads	Identified vulnerable functions	Identified paths
Lodash (master)	unknown	unknown	86	56
Request	35,681	21,155,428	12	6
Async	22,704	46,511,068	72	4
Minimist	11,310	40,869,389	6	4
Body-parser	11,207	26,897,728	2	2
Ramda	5,016	8,958,095	48	20
Morgan	3,897	3,502,973	2	1
Qs	3,790	56,395,909	9	2
Loader-utils	3,777	41,352,810	2	1
Validator.js	2,445	5,671,275	2	2
Grunt	2,059	601,235	16	4
Js.merge	1,411	1,905,969	3	1
Js-beautify	1,313	2,246,662	6	1
Nodemon	1,277	4,895,728	1	1
Vinyl	1,234	4,077,423	1	1
Html-minifier	1,203	3,207,164	134	47
Clean-css	1,164	11,415,710	22	11
Reselect	1,089	3,960,959	1	1
Url-parse	858	9,114,628	1	1
Restify	754	188,600	9	4

Table 4.3: Top-20 NPM packages and vulnerable findings as of April 3rd, 2022.

vulnerability types from our findings with existing entries in the CVE database. If a match is found, the CVE ID is added to the finding, and afterward, we check manually if the existing vulnerability is the same as our finding. The script matched 11 findings as duplicates. We noticed that the naming of vendors and products is not regulated, and thus project vendor names in the CVE database may not match the actual names used in GitHub repositories. Therefore, we perform an additional search for vulnerability duplicates manually, and as a result, find additional 10 matches. At this point, our exploitable dataset contains 290 unique vulnerability reports.

Next, we use the information in our unique findings objects to automatically create reports in a readable format. Firstly, we extract three nodes from each taint path: the exposure of the vulnerability to the user (i.e., top-level function export), the point of entry for a user-controlled variable, and the vulnerable sink. We then use the GitHub link to the corresponding file, and for each node add a specific line reference to the link, so that it points directly to the location of the taint node in question. We also use GitHub APIs to extract information on the project name and

version. Finally, we write a readable template for the reports, and then we automatically insert all of the extracted data into placeholders to give the responsible developers easy-to-understand useful information.

For the responsible disclosure of vulnerabilities, we choose to send emails to the developers responsible for NPM projects. Using GitHub APIs and file links for each finding, we query the repository owner's information and collect their email address. If the email of the owner is unavailable, we query the repository information and find the list of its contributors. Then we iterate through the list until one of the contributors has a publicly available email address. Then we map collected emails to the findings, and group readable reports by the same project, and then by the same email (sometimes one person is responsible for multiple repositories). Lastly, we use the "nodemailer" NPM package [105] to create and send emails to the responsible developers.

Using this vulnerability reporting algorithm we notified 112 responsible developers about 290 exploitable vulnerabilities in their NPM packages.

4.3.2 Ethical considerations during vulnerability disclosure

We performed our experiments in two batches: between May 2021 and October 2021, and between February 2022 and March 2022. After detecting and manually verifying vulnerabilities (during the first period of our experiments), we responsibly disclosed the findings to the owners by contacting them privately via email. In the statement, we included a detailed description of the finding, the corresponding proof-of-concept attack (in selected cases), and our suggested fix. We only received a response from 8/20 contacted owners, all of them are the developers of NPM packages.

Async's owner replied after 10 days, acknowledging the vulnerability, and notifying us about the fix applied in the next version. **Minimist**'s owner replied on the second day after the disclosure, promising to look into it soon. However, after more than 4 months the issue was not addressed. After the CVE ID was obtained, we contacted the owner again, notifying them about publicly disclosing the vulnerability as a GitHub issue. Since it was over 120 days since our first contact, we followed the best practices of ethical disclosure. After the issue was created, the developers of the package deployed a fix in two days.

QueryString cooperates with TideLift, a company that acts as an intermediary in the disclosure process. After several weeks of discussion between TideLift representatives and the package owner, they claimed that the developer who uses the affected functionality of the QS package must provide protection measures themselves. We provided further explanations, however, received no response. After 4 months of the last communication, the CVE ID on this issue was received with a 7.5/10 severity rating, and we contacted them again with a notification about publicly disclosing the vulnerability. We received no response. After some time we discovered that the owners sent a request to MITRE to withdraw the vulnerability. Their request was successful, and the CVE ID was revoked on April 11, 2022. **Ramda**'s owners asked us to disclose security vulnerabilities in GitHub issues and, after our statement, started a discussion with fellow developers as well as with an external security company, which advise the developers on related issues. After the CVE ID with severity rating 9.1/10 was issued, Ramda owners tried to withdraw it from MITRE, but did not succeed. The vulnerability is not addressed as of July 27, 2022.

SailsJS owners argued that the vulnerable functionality is not intended to be accessible to the attacker ("This code runs when Sails starts up, not on a per request basis"). We provided them with a proof-of-concept attack video but received no response. Then we obtained a CVE ID for this issue and made it public. Instead of fixing the vulnerability, the owner then provided a way for the developers to work around this problem. The **AngularJS** team considered our findings and decided not to fix them ("the reported problems don't pass the threshold of a dangerous vulnerability that would require changes to AngularJS"). The requested CVE IDs are being processed by MITRE as of writing.

Finally, **SheetJS** did not acknowledge the disclosed vulnerability as significant, as it does not belong to their main functionality, and **Highland**'s owner mentioned that this package was no longer maintained, despite being downloaded by over 44 thousand users weekly.

After the second period of our experiments, we used our semi-automated vulnerability reporting process. As of writing, we received 22 feedback emails. 13 project owners claimed that either there is no vulnerability, or the vulnerability is insignificant and will not cause any damage. Three owners requested additional information on the vulnerabilities. Another three acknowledged the vulnerabilities and have started to address them. One project owner notified us, that the project contains

the preventive measures for the vulnerability. The applied protection measures are specific to the project functionality and hence were not detected by our approach. Finally, two of the contributors provided the wrong contact information, so the emails bounced back.

4.4 Case studies

After analyzing manually the results of our vulnerability detection and verification framework we conduct several case studies on some of them to understand how these vulnerabilities affect the projects containing them, and what threats they pose to users. We choose the projects from the top, middle, and bottom of the popularity list to understand the vulnerabilities in different project types. Additionally, we select two JavaScript frameworks (both client-side and server-side) to explore possible implications of the found vulnerabilities. All rankings, as well as the usage numbers in this section, are dated by April 2022.

We isolate 20 findings from 17 different projects to describe their impact in detail in this section. 11 projects are from the NPM registry, while the remaining six are from popular websites. From the NPM projects, we select nine packages that were flagged by the taint analysis algorithm; additionally, we select two NPM projects that are JavaScript frameworks (AngularJS, SailsJS), rather than packages (libraries). Note that the vulnerable functions found in the frameworks cannot be detected by taint analysis due to the differences in the usage of such projects by the developers. More precisely, while the functionality from NPM libraries is just imported by the developer in their project, frameworks work as engines, which the developer uses to execute their project. Therefore, the top-level export concept that we utilize to define "sources" of user-controlled data is not applicable in the framework scenario.

By vulnerability types, we targeted 12 prototype pollution findings, six ReDoS findings, and one finding with both vulnerability types (in the SailsJS project). Note that after searching automatically through the MITRE CVE database [26] and manually checking the information available on the web on specific projects and the functions in question, we were unable to locate any reports on these vulnerabilities in publicly accessible sources.

Minimist [106].

This NPM package (40.8 million weekly downloads) is used to parse command-line arguments in Node.js applications. Minimist was previously reported for prototype pollution[4] and the authors issued an updated version with a fix. However, our framework flagged this code piece (with the fix) as vulnerable in the latest version of the package. After the manual analysis, we discovered that the fix does not cover all malicious scenarios. Therefore, we were able to implement PoC attacks that violate both integrity and availability of a JavaScript application.[5] This vulnerability is listed as CVE-2021-44906, and is ranked 9.8/10 (Critical) on the National Vulnerability Database platform (`nvd.nist.gov/vuln/detail/CVE-2021-44906`). It was fixed by the authors 4 months after the initial disclosure.

SailsJS. Sails.js is a model–view–controller web application framework written in JavaScript [107]. Currently, there are 8,415 active websites built on this framework [108]. At least 24 of these websites appear on the top 1 million Tranco list [109]. Our findings indicate that the `loadActionModules()` method is vulnerable to both ReDoS and prototype pollution, due to the absence of sanitization of the strings extracted from filenames. There is a conceivable scenario, where filenames are controlled by the end-user (e.g. dynamic creation of API endpoints). In this case, the method can be exploited in a form of a prototype pollution attack that leads to denial of service.[6]

This vulnerability is listed under CVE-2021-44908 ID, and is ranked 9.8/10 (Critical) on the National Vulnerability Database platform (`nvd.nist.gov/vuln/detail/CVE-2021-44908`). Additionally, certain filenames may also cause availability issues due to the usage of an "evil" regular expression in the method.

Ramda [110]. This NPM package (8.9 million weekly downloads) provides utility functions with a focus on functional programming style. Our findings indicate that the method `mapObjIndexed()` is vulnerable to object property injection vulnerability. Due to insufficient

[4]Prototype pollution in Minimist, `https://security.snyk.io/vuln/SNYK-JS-MINIMIST-559764`

[5]The PoC Minimist attack is available at `github.com/Marynk/JavaScript-vulnerability-detection/blob/main/minimist%20PoC.zip`

[6]The PoC Sails.js attack is available at `github.com/Marynk/JavaScript-vulnerability-detection/blob/main/sailsJS%20PoC.zip`

protection measures, it is possible to pollute `Function.prototype` by supplying a crafted object, causing threats to the integrity and/or availability of the JavaScript application.[7] This vulnerability is listed under CVE-2021-42581 ID and is ranked 9.1/10 (Critical) on the National Vulnerability Database platform (nvd.nist.gov/vuln/detail/CVE-2021-42581).

Async. Async [111] is an NPM package (46.5 million weekly downloads) that provides functionality for working with asynchronous JavaScript. Similar to Ramda, the method `mapValues()` in async package is vulnerable to object property injection.[8] This vulnerability is listed under CVE-2021-43138 ID and is ranked 7.8/10(High) on the National Vulnerability Database platform (nvd.nist.gov/vuln/detail/CVE-2021-43138). It was fixed by the owner 10 days after disclosure in package version 3.2.2.

Lodash [112]. This is a library for JavaScript with various utility functions, mainly for array and object manipulations. It is used in approximately 3.6% of websites as of July 2021 and has 38.9 million weekly downloads [113]. Our findings indicate that the master branch of the lodash GitHub repository contains code exposed to a prototype pollution attack. Vulnerabilities of this type have been reported and subsequently fixed in multiple lodash functions [114, 115, 116, 117]. However, lodash does not apply security fixes to the source code in their master branch. While in the distributables that are supplied to the NPM registry, the found vulnerability is fixed, if developers decide to clone source code from lodash and use it locally, they can clone the master branch. Hence, the discovered vulnerability still bears a significant impact on some applications. More specifically, the internal function `baseAssignValue()` offers no protection measures against modifying `prototype` or `constructor` properties, which potentially exposes any project that uses specific methods from cloned lodash source code, such as `_.set()`, `_.copyObject`, `_.keyBy`, `_.countBy`, `_.groupBy`, to a prototype pollution attack. We implemented a proof-of-concept attack,[9] which targets `_.set()` method and successfully injects a custom property to `Object.prototype`, thus adding this property for all objects of the running application.

Additionally, among detected vulnerable functions we discovered that two of them are from projects that contain local copies of the lodash library: the Highland NPM package [118] and `ac`

[7] PoC ramda attack is available at jsfiddle.net/3pomzw5g/2/
[8] The PoC async attack is available at jsfiddle.net/oz5twjd9/
[9] The PoC lodash attack is available at jsfiddle.net/evmjxaq1/

company.com web domain; these projects use lodash functionalities, but do not automatically receive security updates for it. As a result, both projects are exposed to attacks that can exploit both newly detected and previously found vulnerabilities that were reported and fixed in the original lodash. In particular, the Highland package and accompany.com website contain an older version of the baseAssignValue() function from our findings, pre-dating even the partial fix.

AngularJS [119]. This is a client-side JavaScript framework for developing web applications. This version of Angular has been discontinued since the end of 2021, however, there are still more than 1 million live websites written with this framework [120]. We have identified three prototype pollution vulnerabilities in the source code of AngularJS:

(1) The AngularJS routing system, implemented by the $routeProvider service, contains vulnerable code that exposes certain AngularJS-based applications to a prototype pollution attack. It is possible in the scenario when the paths in such applications are created dynamically based on user inputs. As a consequence, the attacker can disrupt the routing of the application (e.g., navigating between UI components).

(2) A misuse of AngularJS's Select directive can lead to prototype pollution. If the developer of an AngularJS-based application allows the user to dynamically add or modify options associated with Select directive, a special select value can be crafted to perform a prototype pollution attack.

(3) Programmatic navigation within an AngularJS-based application can be performed via the $location service. Particularly, query parameters can be added to the current URL with the $location.search() function. This functionality can be abused to perform a prototype pollution attack if the query parameters in question are dependent on user inputs.

For the first case we have implemented a proof-of-concept attack.[10] If the AngularJS-based application allows the user to supply a custom payload to the $routeProvider.when() method, an attacker is able to manipulate the prototype of the routes object by providing __proto__ as the route path.

QueryString [121].

[10]The PoC AngularJS attack is available at jsfiddle.net/mspc3f8n/

This is an NPM package (56.4 million weekly downloads) that provides parsing and *stringifying* (i.e., transforming any data type into a string literal) functionality of query strings. We discover a prototype poisoning vulnerability in the internal `merge()` method. Due to insufficient input sanitation, attacks on the integrity and availability can be implemented by manipulating the input to `Qs.parse()`.[11].

Grunt-usemin [122]. This is an NPM package that creates a minified version of web files (HTML, CSS, JavaScript) in a project, and it has 29,673 weekly downloads. It can also automatically replace links to scripts in code with the minified versions of the same scripts. We found a potential ReDoS vulnerability in the function that searches for the internal file references in the project and replaces them. In terms of this functionality, the `getBlocks()` function aims to parse an HTML file line by line, and extracts specific information. During this process, a dangerous regular expression is applied to match patterns in every line. A maliciously crafted HTML file can cause ReDoS when processed by Grunt-usemin.

JSON.parse polyfill [123]. Four web domains were flagged due to the use of polyfill for a built-in JavaScript method `JSON.parse`. A polyfill is a piece of code (usually JavaScript on the web) used to provide modern functionality on older browsers that do not natively support it. The flagged domains and their rank in the Cisco Umbrella Popularity list [99] are: `acdc-direct.office.com` (#259), `ad.crwdcntrl.net` (#5806), `activedirectory.windowsazure.com` (#6050) and `360.cn` (#6291). The implementation of `JSON.parse` polyfill by these domains uses an unsafe regex to sanitize the input, which can lead to a ReDoS attack. Furthermore, it can be exploited to run arbitrary code because of the use of `eval()` on the parsed text as part of the functionality implementation (command injection). This attack can be exploited using older versions of Firefox (v.2-3), Opera (v.10.1), Safari (v.3.1-3.2), and Internet Explorer (v.6-7).

Gravatar.com [124]. This is a service for creating universal avatars. Gravatars (globally recognized avatars) are integrated into more than a million websites as of July 2021. On the `gravatar.com` website, there is a WordPress module called `cookie-banner`. The cookies are processed with a dangerous regular expression, and if the user modifies their cookies, such cookie processing implementation will lead to a ReDoS attack.

[11]The PoC QS attack is available at `jsfiddle.net/pb6an1dy/`

SheetJS [125].

This is an NPM package (1.4 million weekly downloads) that provides functionality for working with spreadsheets. It has both commercial and open-source versions, and the open-source version is used in more than 72,000 projects. The GitHub repository of the open-source SheetJS project includes source code for an online demo, where we found a function `deepset()` that is vulnerable to prototype pollution. This function is applied to a JSON representation of an `xlsx` file supplied by users. An attacker can supply a specially crafted file to overwrite properties of `Object.prototype`, thus exploiting the vulnerability. While this function is not in a library functionality itself, it is reasonable to expect it to be used by other developers as a foundation for actual projects, potentially retaining the vulnerability.

Highland [118].

This is an NPM package that provides functionality to work with data streams in JavaScript, and it has 47,759 weekly downloads. Highland exports a function `inspect()`, which we found to be vulnerable to a prototype pollution attack. This function assigns custom options to the context object. By supplying a specially crafted options object, a malicious actor can reassign the prototype of the inspection context, potentially affecting the execution of this function (e.g., causing denial of service, unexpected behavior, and procedure bypass).

Chapter 5

Conclusions and future work

In summary, we propose a framework for function-level detection of JavaScript vulnerabilities in the wild, shifting the focus from package-level vulnerability tracking/measurements considered in the past work. We also design a semi-automated vulnerable function collection mechanism to build a reliable dataset of known vulnerable JavaScript functions. Our dataset contains 1,360 verified vulnerable functions. By testing 9,204,654 real-world JavaScript functions from popular NPM packages, Chrome web extensions, and websites, we detected 124,934 potentially vulnerable functions with a precision of 94.5% (calculated based on a small randomly chosen dataset) by vulnerable pattern search. We then checked the dataset against fuzzy and cryptographic hashes and detected 131 and 965 vulnerable functions with the estimated precision of 100% and 98%, respectively.

Our static taint analysis on the 5,389 findings in NPM packages verified 301 cases with no false positives (19 of them already have CVE IDs). All cases were privately reported to the npm projects repository owners in terms of responsible disclosure. In addition, we conducted an in-depth analysis of 20 detected vulnerabilities from 17 projects and described the attack vectors that can be exploited for these vulnerabilities. Moreover, we performed successful proof-of-concept attacks on seven projects by exploiting the detected vulnerabilities. We obtained four CVE IDs from MITRE organization [26], which have "critical" and "high" severity ratings. Finally, for reproducibility and further research in JavaScript security, all the outcomes of our work are publicly available.[12]

As part of future work, our vulnerable function dataset can be extended by collecting more

[12] https://github.com/Marynk/JavaScript-vulnerability-detection

functions using other types of references from the Snyk database (e.g., GitHub pull requests and GitHub issues). Another way to collect vulnerable functions from NPM is to use the information on package version updates; we can look for the information on a "security patch" in the node advisory. The other approach may be to scan GitHub open-source projects; if developers are following certain rules in maintaining their version control, they might include useful flags in the commit messages. For example, for a vulnerability fixing commit, it is common to include a "fix" word in the commit message, along with a CVE identifier for the vulnerability. We can also perform a search for related keywords through GitHub commit messages.

Besides, in addition to NPM packages, Chrome web extensions, and websites, more environments can be analyzed (e.g., desktop and mobile applications, mini-apps, games, etc.).

Our static taint analysis tool currently verifies the exploitability of found vulnerabilities only in NPM projects. To include verification in other environments (websites, web extensions, other) it is necessary to develop a set of defined "sources" of user-controlled data. However, in terms of detecting different vulnerability types, our tool does not need to be adjusted.

In terms of Semgrep rules, other vulnerability types can be considered besides prototype pollution and ReDoS. The existing rule sets may also be improved to catch more specific patterns since our current pattern search method targets the most common implementations of the two vulnerability types we considered. In addition, Semgrep rules cannot detect heavily obfuscated JavaScript code, as its modified structure obscures the vulnerable patterns.

As part of future work, all detected and manually verified vulnerabilities that affect the whole project will be submitted to MITRE organization and reported to the project owners.

Bibliography

[1] W3Techs. Usage statistics of JavaScript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript, April 2021.

[2] Fabian Beuke. GitHub language statistics. https://madnight.github.io/githut/#/pull_requests/2021/1, March 2021.

[3] Stackoverflow.com. Developer survey. https://insights.stackoverflow.com/survey/2020#most-popular-technologies, February 2020.

[4] Durga Prasad Acharya. The 40 best JavaScript libraries and frameworks for 2022. https://kinsta.com/blog/javascript-libraries/, March 2022.

[5] Andrew Smith. Content spoofing. https://owasp.org/www-community/attacks/Content_Spoofing, 2021.

[6] Softwaretestinghelp.com. JavaScript injection tutorial: Test and prevent JS injection attacks on website. https://www.softwaretestinghelp.com/javascript-injection-tutorial/, 2021.

[7] PortSwigger.net. DOM-based Open Redirection. https://portswigger.net/web-security/dom-based/open-redirection, July 2021.

[8] Niels Provos. A JavaScript-based DDoS Attack as Seen by Safe Browsing. https://security.googleblog.com/2015/04/a-javascript-based-ddos-attack-as-seen.html, April 2015.

[9] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. Lags in the release, adoption, and propagation of NPM vulnerability fixes. *Empirical Software Engineering*, 26, May 2021.

[10] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *NDSS'21*, Virtual, January 2021.

[11] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages. In *IEEE ICSME'18*, pages 559–563, Madrid, Spain, September 2018.

[12] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Code-based vulnerability detection in node.js applications: How far are we? In *35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1199–1203, Los Alamitos, CA, USA, December 2020.

[13] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. On the use of dependabot security pull requests. In *MSR'21*, pages 254–265, Madrid, Spain, June 2021.

[14] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. On the impact of outdated and vulnerable JavaScript packages in docker images. In *IEEE Conference on Software Analysis, Evolution and Reengineering*, pages 619–623, Hangzhou, China, March 2019.

[15] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium*, pages 995–1010, Santa Clara, CA, February 2019.

[16] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the NPM package dependency network. In *Conference on Mining Software Repositories, MSR 2018*, pages 181–191. ACM, May 2018.

[17] Rudolf Ferenc, Péter Hegedüs, Péter Gyimesi, Gábor Antal, Dénes Bán, and Tibor Gyimóthy. Challenging machine learning algorithms in predicting vulnerable JavaScript functions. In *RAISE@ICSE'19*, pages 8–14, September 2019.

[18] OSA. OpenStaticAnalyzer. https://openstaticanalyzer.github.io/, April 2018.

[19] Escomplex. Escomplex: Software Complexity Analysis of JavaScript Abstract Syntax Trees. https://github.com/escomplex/escomplex, 2015.

[20] Balázs Mosolygó, Norbert Vándor, Gábor Antal, Péter Hegedűs, and Rudolf Ferenc. Towards a prototype based explainable JavaScript vulnerability prediction model. In *2021 International Conference on Code Quality (ICCQ)*, pages 15–25, Moscow, Russia, April 2021.

[21] Semgrep.dev. Semgrep–find bugs and enforce code standards. https://semgrep.dev/docs/, 2020.

[22] 1e0ng. Simhash. https://github.com/1e0ng/SimHash, 2020.

[23] Snyk.io. Snyk vulnerability database. https://snyk.io/product/vulnerability-database/, 2015.

[24] Google. The vulnerable code database (Vulncode-DB). https://www.vulncode-db.com, March 2019.

[25] GitHub Inc. GitHub advisory database. https://github.com/advisories, 2017.

[26] MITRE. Common vulnerabilities and exposures. https://cve.mitre.org/, 2021.

[27] Maryna Kluban, Mohamman Mannan, and Amr Youssef. On measuring vulnerable JavaScript functions in the wild. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30–June 3, 2022, Nagasaki, Japan*, May 2022.

[28] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. You've changed: Detecting malicious browser extensions through their update deltas. In *ACM CCS'20*, pages 477–491, November 2020.

[29] Wai Cheung, Sukyoung Ryu, and Sunghun Kim. Development nature matters: An empirical study of code clones in JavaScript applications. *Empirical Software Engineering*, 21:517–564, March 2015.

[30] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. Function clone detection in web applications: A semiautomated approach. *J. Web Eng.*, 3(1):3–21, May 2004.

[31] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528, Oakland, CA, USA, January 2010.

[32] Seokmo Kim, R. Kim, and Young Park. Software vulnerability detection methodology combined with static and dynamic analysis. *Wireless Personal Communications*, 89:1–17, August 2016.

[33] Cypress Data Defense. Differences between static code analysis and dynamic testing. https://www.cypressdatadefense.com/blog/static-and-dynamic-code-analysis/, April 2020.

[34] Cristina Lopes, Petr Maj, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages*, 1:1–28, October 2017.

[35] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *ICSE'07*, pages 96–105, Minneapolis, MN, USA, June 2007.

[36] Yusuke Sasaki, Tetsuo Yamamoto, Yasuhiro Hayase, and Katsuro Inoue. Finding file clones in freebsd ports collection. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 102–105, May 2010.

[37] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. In *MSR'18*, pages 542–553, Gothenburg, Sweden, May 2018.

[38] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. Functional code clone detection with syntax and semantics fusion learning. In *ACM Symposium on Software Testing and Analysis*, page 516–527, Online, July 2020.

[39] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. Modeling functional similarity in source code with graph-based siamese networks. *IEEE Transactions on Software Engineering*, August 2021.

[40] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. SourcererCC: Scaling code clone detection to big-code. In *ICSE'16*, pages 1157–1168, Austin, TX, USA, May 2016.

[41] Jiyong Jang, Maverick Woo, and David Brumley. ReDeBug: Finding unpatched code clones in entire OS distributions. *IEEE Symposium on Security and Privacy*, 37(6):48–62, May 2012.

[42] Benjamin Bowman and H. Howie Huang. VGRAPH: A robust vulnerable code clone detection system using code property triplets. In *IEEE EuroS&P 2020*, pages 53–69, Virtual, September 2020.

[43] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[44] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004.

[45] Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, and Heejo Lee. CLORIFI: software vulnerability discovery using code clone verification. *Concurrency and Computation: Practice and Experience*, 28(6):1900–1917, May 2016.

[46] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In *IEEE Symposium on Security and Privacy*, pages 595–614, May 2017.

[47] Xiaonan Song, Aimin Yu, Haibo Yu, Shirun Liu, Xin Bai, Lijun Cai, and Dan Meng. Program slice based vulnerable code clone detection. In *IEEE TrustCom'20*, pages 293–300, Guangzhou, China, February 2020.

[48] Ira Baxter, Andrew Yahin, Leonardo de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM'98*, pages 368–377, Bethesda, MD, USA, 1998.

[49] Minmin Zhou, Jinfu Chen, Yisong Liu, Hilary Ackah-Arthur, Shujie Chen, Qingchen Zhang, and Zhifeng Zeng. A method for software vulnerability detection based on improved control flow graph. *Wuhan University Journal of Natural Sciences*, 24:149–160, April 2019.

[50] Jingyue Li and Michael D. Ernst. Cbcd: Cloned buggy code detector. In *ICSE'12*, page 310–320, Zurich, Switzerland, 2012.

[51] Hajin Jang, Kyeongseok Yang, Geonwoo Lee, Yoonjong Na, Jeremy D. Seideman, Shoufu Luo, Heejo Lee, and Sven Dietrich. QuickBCC: Quick and scalable binary vulnerable code clone detection. In *ICT Systems Security and Privacy Protection - 36th IFIP TC 11 International Conference, SEC 2021*, volume 625 of *IFIP Advances in Information and Communication Technology*, pages 66–82. Springer, June 2021.

[52] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762, Orlando, FL, USA, December 2018.

[53] Tijana Vislavski, Gordana Rakic, Nicolás Cardozo, and Zoran Budimac. LICCA: A tool for cross-language clone detection. In *IEEE SANER'18*, pages 512–516, Campobasso, Italy, April 2018.

[54] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy*, pages 590–604, San Jose, CA, USA, May 2014.

[55] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *ACSAC'12*, pages 359–368, December 2012.

[56] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Detecting node.js prototype pollution vulnerabilities via object lookup analysis. In *ESEC/FSE'21*, page 268–279, Athens, Greece, August 2021.

[57] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium*, Boston, MA, USA, August 2022.

[58] GitHub. CodeQL, semantic code analysis engine. https://codeql.github.com/, 2019.

[59] Abdullah Sheneamer and Jugal Kalita. Semantic clone detection using machine learning. In *IEEE Conference on Machine Learning and Applications*, pages 1024–1028, Pasadena, CA, USA, February 2016.

[60] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Conference on Neural Information Processing Systems*, pages 10197–10207, Vancouver, Canada, June 2019.

[61] Chaojian Hu, Zhoujun Li, Jinxin Ma, Tao Guo, and Zhiwei Shi. File parsing vulnerability detection with symbolic execution. In *Theoretical Aspects of Software Engineering (TASE'12)*, pages 135–142, Washington DC, USA, August 2012.

[62] Nikolai Tillmann and Jonathan de Halleux. Pex–white box test generation for .net. In *Tests and Proofs (TAP'08)*, pages 134–153, Prato, Italy, April 2008.

[63] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In *ACM CCS'06*, pages 311–321, Alexandria, VA, USA, October 2006.

[64] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS Symposium*, San Diego, CA, USA, February 2016.

[65] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, pages 1–1, February 2015.

[66] Thanassis Avgerinos, Sang Cha, Brent Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Communications of the ACM*, volume 57, January 2011.

[67] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 689–701, July 2017.

[68] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 87–97, March 2015.

[69] Raimondas Sasnauskas and John Regehr. Intent fuzzer: Crafting intents of death. In *WODA+PERTEA'14*, pages 1–5, San Jose, CA, USA, July 2014.

[70] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, pages 1–16, San Diego, CA, USA, February 2016.

[71] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 523–538, August 2012.

[72] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, pages 497–512, Berkeley, CA, USA, July 2010.

[73] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. SoFi: Reflection-augmented fuzzing for JavaScript engines. In *ACM CCS'21*, page 2229–2242, November 2021.

[74] Han HyungSeok, Oh DongHyeon, and Kil Cha Sang. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *NDSS'19*, San Diego, CA, USA, February 2019.

[75] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1, December 2018.

[76] IBM. The t. j. watson libraries for analysis (wala). http://wala.sourceforge.net, 2015.

[77] Joern. Joern - The Bug Hunter's Workbench. joern.io, 2019.

[78] CodeQL. CodeQL: About data flow analysis. https://codeql.github.com/docs/writing-codeql-queries/about-data-flow-analysis/, 2019.

[79] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon J. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE'13*, page 488–498, Saint Petersburg, Russia, August 2013.

[80] Matt Zeunert. FromJS. https://www.fromjs.com/, 2016.

[81] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. Platform-independent dynamic taint analysis for JavaScript. *IEEE Transactions on Software Engineering*, 46(12): 1364–1379, October 2020.

[82] Ben Dickson. Prototype Pollution: The Dangerous and Underrated Vulnerability Impacting JavaScript Applications. https://portswigger.net/daily-swig/prototype-pollution-the-dangerous-and-underrated-vulnerability-impacting-javascript-applications, 2020.

[83] Hee Kim, Ji Kim, Ho Oh, Beom Lee, Si Mun, Jeong Shin, and Kyounggon Kim. DAPP: Automatic detection and analysis of prototype pollution vulnerability in node.js modules. *International Journal of Information Security*, 21:1–23, February 2022.

[84] Zifeng Kang, Song Li, and Yinzhi Cao. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *NDSS'22*, San Diego, CA, USA, April 2022.

[85] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent spring: Prototype pollution leads to remote code execution in node.js. In *USENIX Security Symposium*, Anaheim, CA, USA, 2023.

[86] Adar Weidman. Regular expression denial of service - ReDoS. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS, 2019.

[87] Jamie Davis. Detect vulnerable regexes in your project. https://github.com/davisjam/vuln-regex-detector, March 2018.

[88] Superhuman Labs. RXXR2 regular expression static analyzer. https://github.com/superhuman/rxxr2, 2016.

[89] James Halliday. saferegex. https://github.com/substack/safe-regex, July 2013.

[90] James Kirrage, Asiri Rathnayake Mudiyanselage, and Hayo Thielecke. Static analysis for regular expression Denial-of-Service attacks. In *Network and System Security (NSS'13)*, pages 135–148, June 2013.

[91] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *USENIX Security Symposium*, pages 361–376, Baltimore, MD, USA, August 2018.

[92] Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing-Chi Cheung. ReDoSHunter: A combined static and dynamic approach for regular expression DoS detection. In *30th USENIX Security Symposium*, pages 3847–3864, Virtual, August 2021.

[93] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. Enabling the continuous analysis of security vulnerabilities with VulData7. In *IEEE Source Code Analysis and Manipulation*, pages 56–61, Madrid, Spain, November 2018.

[94] Snyk.io. GitHub snyk vulnerability database. https://github.com/snyk/vulnerabilitydb, 2018.

[95] npm. The node security platform service is shutting down. https://blog.npmjs.org/post/175511531085/insert-title-here.html, July 2018.

[96] npm. Node package registry. https://www.npmjs.com/, 2010.

[97] MITRE. Common weakness enumeration. https://cwe.mitre.org/, 2006.

[98] OpenJS Foundation. Espree. https://www.npmjs.com/package/espree, 12 2014.

[99] Dan Hubbard. Cisco umbrella 1 million. https://umbrella.cisco.com/blog/cisco-umbrella-1-million, December 2016.

[100] Ariya Hidayat. ECMAScript parsing infrastructure for multipurpose analysis. https://esprima.org/, 2012.

[101] Sebastian McKenzie. Babel, the JavaScript compiler. https://babeljs.io/, 2014.

[102] AcornJS. Acorn: A tiny, fast JavaScript parser. https://github.com/acornjs/acorn, 2012.

[103] Joernio. AST generator. https://github.com/joernio/astgen, 2021.

[104] Babel. Babel progress on ECMAScript proposals. https://github.com/babel/proposals, April 2020.

[105] Andris Reinman. NODEMAILER, Send Emails from Node.js – Easy as Cake! https://nodemailer.com/, 2014.

[106] James Halliday. Minimist. https://www.npmjs.com/package/minimist, 2013.

[107] Balderdash Design Co. Sails.js: The MVC framework for node.js. https://sailsjs.com/, 2012.

[108] BuiltWith.com. SailsJS Usage Statistics. https://trends.builtwith.com/framework/SailsJS, April 2022.

[109] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco - a research-oriented top sites ranking hardened against manipulation. In *NDSS'19*, San Diego, CA, USA, February 2019.

[110] Scott Sauyet, Buzz de Cafe. Ramda. https://ramdajs.com/, 2020.

[111] Caolan McMahon. Async. https://caolan.github.io/async/v3/, 2011.

[112] John-David Dalton. Lodash: A modern JavaScript utility library. https://lodash.com/, 2009.

[113] John-David Dalton. Lodash. https://www.npmjs.com/package/lodash, 2009.

[114] Snyk.io. Prototype Pollution in zipObjectDeep, Lodash. https://snyk.io/vuln/SNYK-JS-LODASH-590103, July 2020.

[115] Snyk.io. Prototype Pollution in set/setWith, Lodash. https://snyk.io/vuln/SNYK-JS-LODASH-608086, August 2020.

[116] Snyk.io. Prototype Pollution in defaultsDeep, Lodash. https://snyk.io/vuln/SNYK-DOTNET-LODASH-540457, June 2019.

[117] Snyk.io. Prototype Pollution in merge, mergeWith, and defaultsDeep, Lodash. https://snyk.io/vuln/SNYK-DOTNET-LODASH-540455, August 2018.

[118] Caolan McMahon. Highland: The high-level streams library for node.js and the browser. https://caolan.github.io/highland/, 2014.

[119] Google. AngularJS. https://angularjs.org/, 2010.

[120] BuiltWith.com. AngularJS Usage Statistics. https://trends.builtwith.com/javascript/Angular-JS, April 2022.

[121] Jordan Harband. QS, a query string parsing and stringifying library. https://github.com/ljharb/qs, 2014.

[122] Yeoman team. grunt-usemin. https://www.npmjs.com/package/grunt-usemin/, 2012.

[123] Mozilla. Polyfill. https://developer.mozilla.org/en-US/docs/Glossary/Polyfill, January 2021.

[124] Gravatar.com. Gravatar. https://en.gravatar.com/, 2007.

[125] Sheetjs LLC. SheetJS community edition – spreadsheet data toolkit. https://github.com/SheetJS/sheetjs/, 2012.