On Measuring Vulnerable JavaScript Functions in the Wild

Maryna Kluban
maryna.kluban@mail.concordia.ca
Concordia University
Montreal, Canada

Mohammad Mannan
m.mannan@concordia.ca
Concordia University
Montreal, Canada

Amr Youssef
youssef@ciise.concordia.ca
Concordia University
Montreal, Canada

ABSTRACT

JavaScript is often rated as the most popular programming language for the development of both client-side and server-side applications, and is currently used in almost all websites. 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 work mostly deals with package-level vulnerability tracking and measurements. However this approach is limited to detecting usage of already known vulnerabilities. In this paper we develop a vulnerability detection framework that uses vulnerable pattern recognition and textual similarity methods to detect vulnerable functions in real-world projects. We build our framework with the help of a comprehensive dataset of 1,360 verified vulnerable JavaScript functions that we compose based on Snyk vulnerability database and the VulnCode-DB project. Using our framework, we identify 11,148 vulnerable functions in three environments: NPM packages, Chrome web extensions and popular websites. In addition,we conduct an in-depth contextual analysis of the findings in several popular/critical projects and confirm the security exposure of 15 cases. As evident from the results, our approach can shift JavaScript vulnerability detection from the coarse package/library level to function level, and thus improve accuracy of detection and aid timely patching.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

JavaScript security; vulnerability detection; vulnerable functions

ACM Reference Format:
Maryna Kluban, Mohammad Mannan, and Amr Youssef. 2022. 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. https://doi.org/10.1145/3488932.3497769

1 INTRODUCTION

JavaScript is a programming language used in 97.5% of all web applications [72]. Such applications provide users with direct access to the source code, and are therefore a very common target for attackers. According to GitHub [4] and StackOverflow [67], JavaScript has been a dominating language for software projects development for at least the last 7 years. Moreover, JavaScript language can be used in both client-side and server-side applications, and has a large amount of frequently used development frameworks, such as React (reactjs.org), Angular (angular.io), Vue.js (vuejs.org) (client-side) and Express.js (expressjs.com), Koa (koajs.com), Nest.js (nestjs.com) (server-side).

The consequences of exploiting vulnerabilities in JavaScript source code can be information theft or forgery [48], malicious code injection [65], redirection to attacker-controlled sources [51], disruption of an application functionality [52] and much more.

Previous work [1, 13, 14, 17, 19, 78, 79, 82] on JavaScript vulnerabilities mostly cover 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 flagging of projects as vulnerable due to their use of vulnerable dependencies, when in reality, many such projects (73.3% according to Zapata et al. [78]) 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. [23] use static code metrics, generated for each function by static analysis tools (OpenStaticAnalyzer [47] and escomplex [20]), as features for machine learning (ML) algorithms that predict the probability of the function being vulnerable. Mosolygó et al. [41] use generalized representations of code lines, and calculate vulnerability likelihood based on cosisne distance between vulnerable and analyzed code lines. However, results from both approaches are not very encouraging (F1-measure of 0.7 for [23], and 97.3% false positives for [41]).

The issue of insufficient vulnerable code datasets also hinders research in this area. Ferenc et al. [23] compiled the first publicly accessible dataset of JavaScript functions which includes 1,496 functions marked as vulnerable. In a following study, Mosolygó et al. [41] reduced the dataset from [23] 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 filtered dataset.

The objective of our study 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 vulnerability detection framework based on the following approaches:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIA CCS '22, May 30–June 3, 2022, Nagasaki, Japan.

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9140-5/22/05…$15.00
https://doi.org/10.1145/3488932.3497769

(i) vulnerable pattern recognition, which uses our manually developed patterns to perform a semantics-based search in code; and (ii) search based on textual similarity, which uses content-sensitive and cryptographic hash comparison. For the first approach, we utilize a static analysis tool Semgrep [53], 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 [58] algorithm along with a SHA-1 cryptographic hash value. For each hash value from real-world dataset, we then search for matches in the vulnerable function dataset.

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 [59] and VulnCode-DB project [26] 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.

Contributions. Our contributions can be summarized as follows:

1. We automatically crawl for vulnerable JavaScript functions from Snyk [59] and VulnCode-DB [26], 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 vulnerability 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 11,148 vulnerable functions with an estimated average precision of 94.5% (based on manual verification of a small subset). We perform a case study on 10 popular/critical projects that contain 15 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.

2 RELATED WORK

2.1 JavaScript vulnerability detection

Most past work on JavaScript uses package-level vulnerability detection approaches that mostly target metadata on NPM packages [1, 13, 14, 17, 19, 79, 82]. 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 [78].

There are a few studies that leverage code-based vulnerability detection methods for JavaScript. Ferenc et al. [23] 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. [41] use code lines as targets, and develop a methodology to calculate 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 true 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 [49] and scripts from websites [8, 11]. 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 NPM packages, Chrome web extensions and JavaScript code from top websites, and instead of package/project level analysis we focus on actual vulnerable functions alone.

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: textual vs. semantic similarity-based methods.

Textual similarity methods. These methods search for matches between code instances in the vulnerability dataset and real-world code. To be able to detect slightly modified code (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 [33, 35, 55]. To make the search more efficient, some studies use cryptographic hash or vectorization (e.g., with Word2Vec) on the tokens [30, 34, 66]. Apart from tokenization, several other representations are also used in textual similarity approaches, such as Abstract Syntax Trees (AST) [3, 12, 31], Control Flow Graphs (CFG) [80], Program Dependency Graphs (PDG) [36], code binaries [29], or a combination of these representations [5]. In order to detect code that is more significantly modified (e.g., added/deleted/changed statements), some algorithms use locality-sensitive hashing [12, 31] on tokens, but unfortunately such approaches introduce 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-based approaches [21, 54, 69, 74], which create signatures for functions based on their syntactic features, and then compare the signatures between vulnerable and targeted functions.

There are several textual similarity tools that 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 [71], PDG and CFG [5, 34, 69, 74]. 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.

Semantic similarity methods. These methods detect semantic (functional) similarities 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. [75] 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 of the same authors uses AST to extrapolate vulnerabilities [76]. These methods are precise and produce few false positives. However, they only capture specific vulnerability types, for which the patterns were created.

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 [75]) is non-trivial, we apply another simpler, yet effective, approach. More precisely, we utilize Semgrep – a static analysis tool which allows 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 [21, 22, 38, 54, 57, 81]. While these tools aim to spare researchers time and effort, we did not choose to proceed with machine learning approaches: ML/DL 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.

2.3 Vulnerability datasets

To create vulnerability datasets, several past studies collect meta-information, packages, and functions/code-snippets from known/reported vulnerabilities. For example, VulData7 [32] uses the National Vulnerability Database (NVD) to extract information on vulnerabilities. However, it does not extract the code from files.

Ferenc et al. [23] use the GitHub repositories of the Snyk vulnerability database [60] and the Node Security Platform (NSP [46]) 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 no longer publicly available.

Mosolygó et al. [41] worked with vulnerable functions from dataset in [23]. 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 reliable vulnerability dataset is the primary motivation for our work. We use up-to-date and well-maintained vulnerability sources (Snyk [59] and VulnCode-DB project [26]), 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 key words in the commit message, such as “bug”, “fix”, “CVE” etc. and extract functions from commits [5, 34, 34, 35, 70]. 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.

3 APPROACH OVERVIEW

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 in three major steps: collection of JavaScript functions for creating the vulnerable function dataset, refinement and verification of the dataset, and vulnerable function detection in real-world projects. Figure 1 shows an overview of our approach.

First, we collect all details and functions from each vulnerability entry from selected sources and perform preliminary filtering to exclude irrelevant data. Second, we analyze the collected possibly vulnerable functions, and design a semi-automated function verification procedure. Finally, we develop a framework that uses several approaches and our verified vulnerable function dataset to search for vulnerabilities in the functions that we collected from real-world projects. 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 detect 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 vulnerable dataset, and we use them for completeness. Finally, we manually verify exploitability of a small subset of our findings in high-profile projects.

4 VULNERABLE FUNCTION DATASET PREPARATION

In this section, we describe the process of creating our vulnerable functions dataset. First, we collect possibly vulnerable functions

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

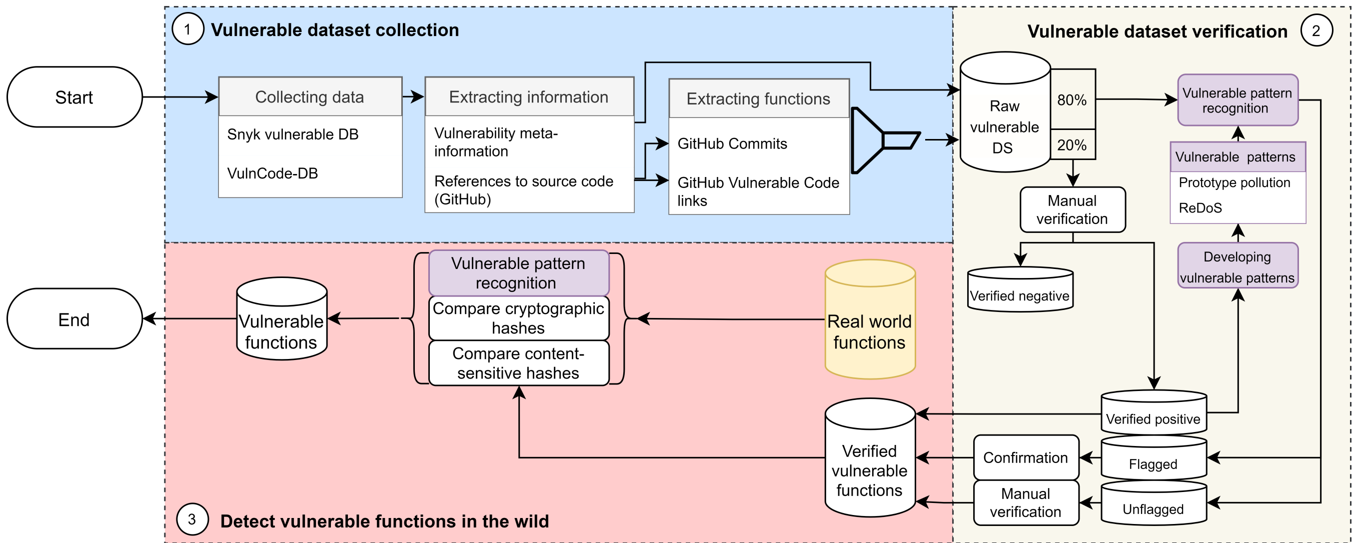

Figure 1: Overview of our approach

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.

4.1 Collecting Possibly Vulnerable Functions

4.1.1 Vulnerability sources. We use two sources for vulnerable functions collection: Snyk vulnerability database [59] and Google VulnCode-DB project [26].

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 [59]. 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 [44]. While having a massive code base (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 also includes projects beyond NPM packages. VulnCode-DB also relies on the community to add relevant information for each vulnerability [26], 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 [39]) identifier; CWE (common weakness enumeration [40]) 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

Category	No. (percentage)
GitHub commits	1291 (44.43%)
Advisories	474 (16.31%)
GitHub issue	244 (8.40%)
Pull requests	201 (6.92%)
Vulnerable code	170 (5.85%)
Bug report	123 (4.23%)
Other links	403 (13.87%)
Total	2906

Table 1: Categories of collected links

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 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, where 201 of them are for JavaScript projects with at least one reference link.

4.1.2 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 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 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` [24], and receive the range of each function (first and last symbol positions in string). Lastly, we extract all functions with modified lines within their range. We also collect nested functions that include modified lines.

Links of GitHub “vulnerable code” type contain modified line range at the end, in a format “#Li-j”, where i is the fist 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 as for commit links.

The structure of each entry in our dataset is presented in Listing 1. For each vulnerability entry we place the affected files in the “files” property (see Listing 1). 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 vulnerable link to the “files” property, and vulnerable functions in the “affectedFunctions” property.

```
{
  "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"
}
```

Listing 1: Structure of an entry in our vulnerable function dataset

After collecting these possible vulnerability entries, we perform preliminary filtering to remove 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, tash, 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 entries), regular expression denial of service (ReDoS, 121 entries) and prototype pollution (101 entries); see Table 5 in the Appendix.

Following a similar approach, Ferenc et al. [23] produced a dataset of 1,456 vulnerable functions; however, at least 1,013 of these functions are in fact not vulnerable [41]. 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. 4.3).

4.2 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 5 in the Appendix). For each function the objective is to make a decision, 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.

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 versification process involves less scrutiny than for other functions. .

4.3 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 [53], 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.

[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.

4.3.1 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 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. At 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.

4.3.2 Prototype pollution. This vulnerability occurs 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:

Semgrep rule

```yaml
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

```javascript
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;
  }
}
```

```javascript
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 2: Example of a Semgrep rule for prototype pollution

```
{"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 properties, the program performs the following 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 [18].

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`.

By understanding the vulnerability and its preventive measures, we can now 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 the Figure 2, along with an example of two vulnerable functions, targeted by the rule.

Semgrep rule

```
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 examples

```
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: Example of a Semgrep rule for ReDoS

4.3.3 Regular Expression Denial of Service. The second vulnerability type that we develop patterns for is ReDoS. It is a specific case of Denial of Service (DoS) attack that happens when a program runs a user's input through an evil regular expression [73], that takes exponential time to process specially-crafted complex strings. This usually happens when operators in 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!'` [73]. The protection measures for ReDoS are either sanitizing the input (i.e., limit the length or discard repetition patterns), or rewriting regular expressions without using "dangerous"combinations of operators (such as '+' and '*', '?' and '>', '?' and '='). There are several tools [16, 28, 45, 68] that can analyze regular expressions and flag the dangerous ones in a given code base. For our purpose, we choose to use the NPM module `safe-regex` [28]. 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 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 rules that account for four different ReDoS scenarios. Figure 3 shows an example of a ReDoS rule and two different targeted functions.

4.4 Our final vulnerable functions dataset

With two rule sets, developed for prototype pollution and ReDoS vulnerability types, we executed 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 2; 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

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

Table 2: Summary on the vulnerable pattern findings in our vulnerable function dataset

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 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 from 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 for 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 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 still 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 [41], and the biggest dataset of verified vulnerable JavaScript functions.

5 VULNERABLE FUNCTION DETECTION: EXPERIMENT AND RESULTS

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 vulnerable functions. We also present the summary of our findings.

5.1 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 [15]). In the NPM open source package registry, packages are mostly written

Dataset	No. entries	No. functions
NPM packages	3000	413,774
Chrome extensions	557	2,659,649
Top websites	1893	5,739,271
Total	5450	9,205,624

Table 3: Summary of the datasets of real world functions

in JavaScript. We extract JavaScript files from GitHub repositories of 3000 most popular NPM packages.[3] The majority of scripts in Chrome web extensions is 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.

From the Cisco Umbrella Popularity List we choose 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 18,107 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 1,893 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 3.

5.2 Implementation of search algorithms

5.2.1 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 4.3). 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 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.

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

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 a small batch of real-world functions (i.e. adding new conditions `pattern-not` and `pattern-not-inside`).

5.2.2 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 [41]. 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. See Figure 4 in the Appendix for an 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` [58] Python module. First, we create content-sensitive hashes for all functions of real-world and vulnerable datasets using `Simhash()` method. Then we utilize the `get_near_dups()` method, which uses the hamming distance to compare the hashes. Simhash allows to choose 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 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 real world, 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.

5.3 Results

We perform our textual-similarity tests, which are very efficient, on the whole real-world dataset (9,205,654 functions). On the other hand, we only processed a total of 795,912 real-world functions with Semgrep vulnerable pattern search, which is more computationally-intensive. With Semgrep, we analyzed 171,109 functions from NPM (1,300 packages), 325,978 functions from Chrome web extensions (31 extensions) and 298,825 functions from websites (122 websites). Out of these functions, our Semgrep rules flagged 18,362 potential prototype pollution and 1,720 potential ReDoS vulnerable functions. The distribution of the detected functions among all tested environments is presented in Table 4.

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 9,858 unique detected functions for prototype pollution pattern and 669 functions for ReDoS.

The Simhash algorithm matched 1320 functions from the whole real-world dataset. 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 (237). Similar to vulnerable pattern search, Simhash also detected several nested functions. We apply the same filtering script on all detected functions and as a result we get 965 unique vulnerabilities.

Finally, cryptographic hash matching algorithm produced 131 matching cases from all real-world functions. All of the matches were already detected by the Simhash comparison. 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 (31).

Note that all ReDoS and prototype pollution vulnerabilities detected by textual similarity methods were also detected by our Semgrep rules. This is an 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 11,148 unique functions, detected by at least one method in our framework. 10,527 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	Ext.	Websites	Total	Unique
Vuln. Patterns	Proto. pollution	4,592	7,080	6,690	18,362	9,858
	ReDoS	552	542	626	1,720	669
	Total detected	5,144	7,622	7,316	20,082	10,527
	Fuzzy hash	56	201	1,063	1,320	965
	Crypto hash	30	85	16	131	131

Table 4: Results on detection of vulnerable functions in the real-world functions (for textual similarity methods results are presented for all detected vulnerability types)

5.4 Manual validation of the results

To evaluate the performance of 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 confim 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 into precision 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 case of the assignment to the array element, the “key” is an index of the element. However, the prototype pollution applies only to 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 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 precision rate of 98%. There were two reasons for a 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).

6 CASE STUDIES

After the manual verification of the 290 detected functions, we conduct case studies on some of them in order to understand how these vulnerabilities affect the projects containing them, and what threat do they pose to users. We isolate 15 findings from 10 different projects to describe their impact in detail in this section. By

vulnerability types we targeted eight prototype pollution findings, five ReDoS findings, and one finding with both vulnerability types (in SailsJS project). Note that after searching through multiple vulnerability databases (NVD, Snyk) and researching the available information on specific projects and the functions in question we were unable to locate any reports on these vulnerabilities in publicly accessible sources.

Ramda. Ramda [56] is an npm package (10.3 million weekly downloads), that provides utility functions with a focus on functional programming style. Our findings indicate that `mapObjIndexed()` method is vulnerable to object property injection. Due to insufficient protection measures, it is possible to pollute the prototype of the `Function` by supplying a crafted object, causing threats to integrity and/or availability of the JavaScript application.[4]

async. Async [9] is an npm package (40.4 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.[5]

Lodash. Lodash is a library for JavaScript with various utility functions, mainly for array and object manipulations [37]. It is used in approximately 3.6% of websites as of July 2021, and has 38.9 millions of weekly downloads [43]. Our findings indicate that the master branch of 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 [61–64]. 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 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,[6] 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 two from projects that contain local copies of the lodash library: the Highland NPM package [10] and accompany.com web domain; these projects use functionality, provided by lodash, but do not automatically receive security updates for it. As a result, both projects are exposed to the attacks, that can exploit both newly detected and previously found vulnerabilities that were reported and fixed in the original lodash. In particular, Highland package and accompany.com website contain an older version of the `baseAssignValue()` function from our findings, pre-dating even the partial fix.

AngularJS. AngularJS [25] is a client-side JavaScript framework for developing web applications. This version of Angular is going to be discontinued at the end of 2021, however there are still more than 1.2 millions live websites written with this framework [6]. We have identified three prototype pollution vulnerabilities in the source code of AngularJS:

(1) AngularJS routing system, implemented by `$routeProvider` service, contains vulnerable code that exposes certain AngularJS-based applications to the prototype pollution attack. It is possible in the scenario, when the paths in such applications are created dynamically based on user input. As a consequence, the attacker can disrupt the routing of the application.
(2) A misuse of AngularJS's `Select` directive can lead to prototype pollution. If the developer of an AngularJS-based application allows the user to dynamically add or modify options associated with `Select` directive, a special select value can be crafted to perform a prototype pollution attack.
(3) Programmatic navigation within an AngularJS-based application can be performed via the `$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 input.

For the first case we have implemented a proof-of-concept attack.[7] 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.

Sails.js. Sails.js is a model–view–controller web application framework written in JavaScript [2]. Currently there are 8,265 active websites built on this framework [7]. At least 24 of these websites appear on Top 1 million Tranco list [50]. 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 for a form of a prototype pollution attack that leads to denial of service.[8]. Additionally, certain filenames may also cause availability issues due to the usage of an "evil" regular expression in the method.

Grunt-usemin. Grunt-usemin 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 [77]. It can also automatically replace links to scripts in code to 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, `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. 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 used to provide modern functionality

[4]The PoC Ramda attack is available at https://jsfiddle.net/3pomzw5g/2/
[5]The PoC async attack is available at https://jsfiddle.net/oz5twjd9/
[6]The PoC lodash attack is available at https://jsfiddle.net/evmjxaq1/
[7]The PoC AngularJS attack is available at https://jsfiddle.net/mspc3f8n/
[8]The PoC Sails.js attack is available at https://github.com/Marynk/JavaScript-vulnerability-detection/blob/main/sailsJS%20PoC.zip

on older browsers [42]. The flagged domains and their rank in the Cisco Umbrella Popularity list [15] 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 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. Gravatar is a global service for creating universal avatars [27]. 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 in case of the user being able to modify their cookies, such cookie processing implementation will lead to a ReDoS attack.

7 LIMITATIONS AND FUTURE WORK

Although we verified several of our findings as part of our case studies, we want to note that, when a function is flagged by our framework as vulnerable, this does not always imply that the project containing this function is vulnerable as well. Additional measures, applied from the outside of the function (such as input sanitation), may restrict an attack. On the other hand, it is still important to identify such functions, as they might be reused in an unsafe place, either in the same project or in a different one. Also, the code outside of the function might change, possibly eliminating the applied protection measures.

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

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.

8 CONCLUSIONS

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 a batch of 795,912 real-world JavaScript functions from popular NPM packages, Chrome web extensions, and websites, we detected 10,527 vulnerable functions with a precision of 94.5% (calculated based on a small randomly chosen dataset) by vulnerable pattern search. We then checked our whole real-world dataset (9,205,654 functions) using fuzzy and cryptographic hashes, and detected 131 and 965 vulnerable functions with the estimated precision of 100% and 98%, respectively. In addition, we conducted an in-depth analysis on 15 detected vulnerable functions from 10 projects, and described the attack vectors that can be exploited for these vulnerabilities. Moreover, we performed successful proof-of-concept attacks on two of the projects by exploiting the detected vulnerabilities. Finally, for reproducibility and further research in JavaScript security, all the outcomes of our work will be made publicly available, after our responsible disclosure to possibly affected developers and site operators.[9]

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments and suggestions to improve the paper's presentation. This research is partly supported by Mitacs (first author) and Natural Sciences and Engineering Research Council of Canada Discovery Grants (second and third authors).

REFERENCES

[1] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. 2021. On the Use of Dependabot Security Pull Requests. In *18th IEEE/ACM International Conference on Mining Software Repositories*. IEEE, random address, 254–265.

[2] Balderdash Design Co. 2012. Sails.js: The MVC framework for Node.js. https://sailsjs.com/.

[3] Ira Baxter, Andrew Yahin, Leonardo de Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. *Proc. of International Conference on Software Maintenance* 368-377 (01 1998), 368–377.

[4] Fabian Beuke. 2021. GitHub Language Statistics. https://madnight.github.io/githut/#/pull_requests/2021/1.

[5] Benjamin Bowman and H. Howie Huang. 2020. VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020*. IEEE, 53–69.

[6] BuiltWith.com. 2021. Angular JS Usage Statistics. https://trends.builtwith.com/javascript/Angular-JS.

[7] BuiltWith.com. 2021. SailsJS Usage Statistics. https://trends.builtwith.com/framework/SailsJS.

[8] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. 2004. Function Clone Detection in Web Applications: A Semiautomated Approach. *J. Web Eng.* 3, 1 (2004), 3–21.

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

[10] Caolan McMahon. 2014. HIGHLAND: The high-level streams library for Node.js and the browser. https://caolan.github.io/highland/.

[11] Wai Cheung, Sukyoung Ryu, and Sunghun Kim. 2015. Development nature matters: An empirical study of code clones in JavaScript applications. *Empirical Software Engineering* 21 (03 2015).

[12] Wai Cheung, Sukyoung Ryu, and Sunghun Kim. 2015. Development nature matters: An empirical study of code clones in JavaScript applications. *Empirical Software Engineering* 21 (03 2015).

[13] Bodin Chinthanet, Raula Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* 26 (05 2021).

[14] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2020. Code-based Vulnerability Detection in Node.js Applications: How far are we? *CoRR* abs/2008.04568 (2020).

[15] Dan Hubbard. 2016. Cisco Umbrella 1 Million. https://umbrella.cisco.com/blog/cisco-umbrella-1-million.

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

- [16] Jamie Davis. 2018. vuln-regex-detector: Detect vulnerable regexes in your project. https://github.com/davisjam/vuln-regex-detector.
- [17] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Conference on Mining Software Repositories, MSR 2018*. ACM, 181–191.
- [18] Ben Dickson. 2020. 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.
- [19] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021*. The Internet Society.
- [20] escomplex 2015. Escomplex: Software complexity analysis of JavaScript abstract syntax trees. https://github.com/escomplex/escomplex.
- [21] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 516–527.
- [22] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional Code Clone Detection with Syntax and Semantics Fusion Learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, 516–527.
- [23] Rudolf Ferenc, Péter Hegedüs, Péter Gyimesi, Gábor Antal, Dénes Bán, and Tibor Gyimóthy. 2019. Challenging machine learning algorithms in predicting vulnerable JavaScript functions. In *Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE@ICSE 2019*. IEEE / ACM, 8–14.
- [24] OpenJS Foundation. 2014. Espree. https://www.npmjs.com/package/espree.
- [25] Google. 2010. AngularJS. https://angularjs.org/.
- [26] Google. 2019. The vulnerable code database (Vulncode-DB). https://www.vulncode-db.com.
- [27] Gravatar.com. 2007. Gravatar: One avatar for everything, everywhere. https://en.gravatar.com/.
- [28] James Halliday. 2013. safe-regex: detect possibly catastrophic, exponential-time regular expressions. https://github.com/substack/safe-regex.
- [29] Hajin Jang, Kyeongseok Yang, Geonwoo Lee, Yoonjong Na, Jeremy D. Seideman, Shoufu Luo, Heejo Lee, and Sven Dietrich. 2021. QuickBCC: Quick and Scalable Binary Vulnerable Code Clone Detection. In *ICT Systems Security and Privacy Protection - 36th IFIP TC 11 International Conference, SEC 2021 (IFIP Advances in Information and Communication Technology, Vol. 625)*. Springer, 66–82.
- [30] Jiyong Jang, Maverick Woo, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. *login Usenix Mag.* 37, 6 (2012).
- [31] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society, 96–105.
- [32] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. 2018. [Engineering Paper] Enabling the Continuous Analysis of Security Vulnerabilities with VulData7. In *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018*. IEEE Computer Society, 56–61.
- [33] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on* 28 (08 2002), 654– 670.
- [34] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy, SP 2017*. IEEE Computer Society, 595–614.
- [35] Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, and Heejo Lee. 2016. CLORIFI: software vulnerability discovery using code clone verification. *Concurr. Comput. Pract. Exp.* 28, 6 (2016), 1900–1917.
- [36] Jingyue Li and Michael Ernst. 2012. CBCD: Cloned buggy code detector. *Proceedings - International Conference on Software Engineering* (06 2012), 310–320.
- [37] Lodash Utilities. 2009. Lodash: A modern JavaScript utility library delivering modularity, performance & extras. https://lodash.com/.
- [38] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. 2020. Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks. *CoRR* abs/2011.11228 (2020).
- [39] MITRE. 2021. Common vulnerabilities and exposures. https://cve.mitre.org/.
- [40] MITRE. 2021. Common weakness enumeration. https://cwe.mitre.org/.
- [41] Balázs Mosolygó, Norbert Vándor, Gábor Antal, Péter Hegedűs, and Rudolf Ferenc. 2021. Towards a Prototype Based Explainable JavaScript Vulnerability Prediction Model. In *2021 International Conference on Code Quality (ICCQ)*. IEEE, 15–25.
- [42] Mozilla. 2021. Polyfill. https://developer.mozilla.org/en-US/docs/Glossary/Polyfill.
- [43] npm. 2009. Lodash NPM Library. https://www.npmjs.com/package/lodash.
- [44] npm. 2010. Node Package Registry. https://www.npmjs.com/.
- [45] npm. 2015. ESLint rules for Node Security. https://github.com/nodesecurity/eslint-plugin-security/blob/master/rules/detect-unsafe-regex.js.
- [46] npm. 2018. The Node Security Platform service is shutting down. https://blog.npmjs.org/post/175511531085/insert-title-here.html.
- [47] OSA 2018. OpenStaticAnalyzer. https://openstaticanalyzer.github.io/.
- [48] OWASP.org. 2021. Content Spoofing. https://owasp.org/www-community/attacks/Content_Spoofing.
- [49] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. 2020. You've Changed: Detecting Malicious Browser Extensions through their Update Deltas. In *ACM CCS'20*. 477–491.
- [50] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2018. Tranco - A Research-Oriented Top Sites Ranking Hardened Against Manipulation. https://tranco-list.eu/.
- [51] PortSwigger.net. 2021. DOM-based open redirection. https://portswigger.net/web-security/dom-based/open-redirection.
- [52] Niels Provos. 2015. A Javascript-based DDoS Attack as seen by Safe Browsing. https://security.googleblog.com/2015/04/a-javascript-based-ddos-attack-as-seen.html.
- [53] r2c. 2020. Semgrep. Static analysis at ludicrous speed. Find bugs and enforce code standards. https://semgrep.dev/docs/.
- [54] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *CoRR* abs/1807.04320 (2018).
- [55] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2015. SourcererCC: Scaling Code Clone Detection to Big Code. *CoRR* abs/1512.06448 (2015).
- [56] Scott Sauyet, Buzz de Cafe. 2020. Ramda. https://ramdajs.com/.
- [57] Abdullah Sheneamer and Jugal Kalita. 2016. Semantic Clone Detection Using Machine Learning. In *15th IEEE International Conference on Machine Learning and Applications, ICMLA 2016*. IEEE Computer Society, 1024–1028.
- [58] SIM 2020. SimHash. https://github.com/1e0ng/simhash.
- [59] Snyk.io. [n.d.]. Snyk Vulnerability DB. https://snyk.io/product/vulnerability-database/.
- [60] Snyk.io. 2018. GitHub Snyk Vulnerability Database. https://github.com/snyk/vulnerabilitydb.
- [61] Snyk.io. 2019. Prototype Pollution in defaultsDeep, Lodash. https://snyk.io/vuln/SNYK-DOTNET-LODASH-540457.
- [62] Snyk.io. 2019. Prototype Pollution in merge, mergeWith, and defaultsDeep, Lodash. https://snyk.io/vuln/SNYK-DOTNET-LODASH-540455.
- [63] Snyk.io. 2020. Prototype Pollution in set/setWith, Lodash. https://snyk.io/vuln/SNYK-JS-LODASH-608086.
- [64] Snyk.io. 2020. Prototype Pollution in zipObjectDeep, Lodash. https://snyk.io/vuln/SNYK-JS-LODASH-590103.
- [65] Softwaretestinghelp.com. 2021. JavaScript Injection Tutorial: Test and Prevent JS Injection Attacks On Website. https://www.softwaretestinghelp.com/javascript-injection-tutorial/.
- [66] Xiaonan Song, Aimin Yu, Haibo Yu, Shirun Liu, Xin Bai, Lijun Cai, and Dan Meng. 2020. Program Slice based Vulnerable Code Clone Detection. In *19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2020*. IEEE, 293–300.
- [67] Stackoverflow.com. 2020. 2020 Developer Survey. https://insights.stackoverflow.com/survey/2020#most-popular-technologies.
- [68] Superhuman Labs. 2016. RXXR2 regular expression static analyzer. https://github.com/superhuman/rxxr2.
- [69] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *Conference on Mining Software Repositories (MSR'18)*. 542–553.
- [70] Fabien Patrick Viertel, Wasja Brunotte, Daniel Strüber, and Kurt Schneider. 2019. Detecting Security Vulnerabilities using Clone Detection and Community Knowledge. In *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 245–324.
- [71] Tijana Vislavski, Gordana Rakic, Nicolás Cardozo, and Zoran Budimac. 2018. LICCA: A tool for cross-language clone detection. In *25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE Computer Society, 512–516.
- [72] W3Techs.com. 2021. Usage statistics of JavaScript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript.
- [73] Adar Weidman. 2019. ReDoS. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS.
- [74] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. 87–98.
- [75] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014*. 590–604.
- [76] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *ACSAC'12*. 359–368.

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

[78] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. 2018. Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*. IEEE Computer Society, 559–563.

[79] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. 2019. On the Impact of Outdated and Vulnerable Javascript Packages in Docker Images. In *IEEE Conference on Software Analysis, Evolution and Reengineering*. 619–623.

[80] Minmin Zhou, Jinfu Chen, Yisong Liu, Hilary Ackah-Arthur, Shujie Chen, Qingchen Zhang, and Zhifeng Zeng. 2019. A Method for Software Vulnerability Detection Based on Improved Control Flow Graph. *Wuhan University Journal of Natural Sciences* 24 (04 2019), 149–160.

[81] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *CoRR* abs/1909.03496 (2019).

[82] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. *CoRR* abs/1902.09217 (2019).

APPENDIX

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 5: Distribution of vulnerability types among all collected entries

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 4: Example of a function and its tokenized version

Manual filtering framework for vulnerable functions

PREV [0]Remote Memory Exposure(✓) NEXT IMPORT FILE SAVE FILE

EXPORT CONFIRMED FUNCTIONS ☐ Hide done

CVE-2021-23386

Remote Memory Exposure. dns-packet is an An abstract-encoding compliant module for encoding / decoding DNS packetsAffected versions of this package are vulnerable to Remote Memory Exposure. It creates buffers with allocUnsafe and does not always fill them before forming network packets. This can expose internal application memory over unencrypted network when querying crafted invalid domain names.

commit

INDEX.JS (✓)

ACCEPT ALL DECLINE ALL

FUNC 0(✓)

ACCEPT DECLINE UNMARK

```
1     function (n) {                       1     function (n) {
2       if (n === '.') return 1            2       if (n === '.') return 1
3  -    return Buffer.byteLength(n         3  +    return Buffer.byteLength(n
        ) + 2                                      .replace(/^\.|\.$/gm, '') )
                                                   + 2
4     }                                    4     }
```

Figure 5: GUI of the framework for manual vulnerable functions verification