Detecting JavaScript libraries using identifiers and hashes

Saša Lončarević, Bruno Skendrović, Ivan Kovačević, Stjepan Groš
University of Zagreb Faculty of Electrical Engineering and Computing
Zagreb, Croatia
{sasa.loncarevic, bruno.skendrovic, ivan.kovacevic, stjepan.gros}@fer.hr

***Abstract*—The web is a widespread platform for data exchange and service delivery on which many depend. Due to the high demand for website creation, methods are offered that provide more time-efficient design and programming. One of them is using JavaScript libraries, which can introduce vulnerabilities into web applications. Because of the sheer number of websites, as well as libraries and their vulnerabilities, it is difficult to maintain website security. There is a need to create tools that will automatically and proactively search for vulnerable libraries and enable timely remediation. This paper deals with the aforementioned problem and provides an insight into the implemented solution. The paper describes how JavaScript libraries are obtained and used, and proposes multiple methods for detecting them. It also presents the technical side of the implementation, as well as the results obtained by detecting libraries on a set of web pages from the Croatian web space. Finally, we discuss the observed distribution of popular libraries and their vulnerabilities, as well as the limitations of the proposed implementation, and offer potential solutions with the aim of improving the project.**

Keywords*—*JavaScript libraries detection, vulnerabilities, Croatian web space, cybersecurity

I. Introduction

The World Wide Web represents a large collection of websites that facilitate the exchange of information and provide important services to both companies and individuals. To make a website more dynamic and highly functional, developers use client-side JavaScript. Advanced features are commonly implemented through the use of JavaScript libraries. It is a modern solution that speeds up development of websites, but carelessness can give attackers an opportunity to abuse them.

For example, due to maintenance failures, websites can end up with out of date, and potentially vulnerable libraries. Vulnerabilities can be used to target both the websites and end users. Some of those vulnerabilities include cross-site scripting, input validation, stealing session data. Naturally, the need arises for a solution that will raise the level of security and prevent attackers from performing unwanted actions.

Therefore, the aim of this paper is to help and provide such tool, implemented in a way to detect JavaScript libraries using automated detection methods. Knowing the name and version of a library is valuable because it consequently gives the information about known vulnerabilities for the given library. Such tool would enable proactive action, so unsecure websites could be patched before potential attackers take advantage of them.

This paper is organized as follows. The introductory Section of this paper is followed by related research in Section II, where other papers and projects are discussed. Section III deals with the general understanding of using JavaScript libraries. In Section IV the theory of JavaScript library detection methods is presented. Section V describes the technical side of developing a system for detecting JavaScript libraries. In Section VI the conducted experiments and testing is described. Section VII contains the discussion about theoretical and practical limitations of implemented system. The paper ends with the conclusion of the topic and possible future work insights.

II. Related Work

While there are many prior studies about vulnerability detection, which focus on pure JavaScript code, such as Kluban et al. [1], Guarnieri et al. [2] and Shar et al. [3], there are not many that deal with detecting JavaScript libraries which are known to be vulnerable. Such approach could be of greater help in website security.

T. Lauinger et al. [4] conducted a comprehensive study of JavaScript library usage across popular web sites. While giving impressive insights about vulnerable libraries, some problems still arise during detection. Minimal modifications in library code, especially in minified versions, hinders detection. The source of the problem is attributed to many different, decentralized sources for obtaining JavaScript libraries. In comparison this paper demonstrates a way to handle minimal modifications in libraries and speed up detection.

Retire.js [5] is a project that deals with a similar matter. It is a tool that uses multiple static and dynamic methods to detect JavaScript libraries used in a website. It uses network traffic monitoring and regular expressions to detect libraries while this paper focuses on methods which are purely static. Another tool, in a form of a browser extension, is Library Detector For Chrome [6]. Strictly dynamic methods are used, detection is carried out by looking for known attributes of JavaScript libraries. This tool does not provide information about vulnerabilities. Content management system detector, Wappalyzer [7], can also detect JavaScript libraries by analyzing data from HTTP response or headers. Data gathered by aforementioned tools, as well as our proposed solution, can be of great essence in developing website vulnerability prediction models such as [8].

III. Background

JavaScript is one of the most popular programming languages for creating web applications. Research from 2022, conducted on more than 80 thousand respondents, tells how over 65% of software engineers actively use it [9]. Frequent use of client-side JavaScript led to creation of libraries. Libraries represent a set of ready-made code that provides certain functionality, it is also sufficiently generalised so it can be used on larger set of problems. Some of them are built to access the HTML DOM (Document Object Model) [10] and introduce changes to it dynamically in specific situations. For example one of the most popular JavaScript libraries is jQuery [11], it allows developers to manipulate DOM and handle user-triggered events in a simple and fast way. Alongside jQuery, popular libraries are Bootstrap, Modernizr, React, Vue [12]. Apart from libraries, their plugins are also important because they can introduce vulnerabilities to websites as well. Plugins provide an additional functionality without requiring to modify the library. Due to flexibility they offer, they are widely used.

JavaScript libraries are not organized nor available in a centralized repository. The reason for this is that the libraries are mostly open source and made by different vendors, so owners publish them on different platforms. But many libraries are available on CDNs (Content delivery networks) [13]. CDNs are groups of servers, spread out over many locations, which store duplicate copies of libraries so requests could be answered as fast as possible. Common way to implement JavaScript libraries in websites is by using HTML tags like `<script src="">` and referencing a link to CDN resource.

Despite their popularity and widespread use, JavaScript libraries are susceptible to vulnerabilities. JavaScript is not necessarily an unsecure programming language, but with careless organization of code and disregard for good security practices the code may contain flaws that attackers can exploit. Research conducted on a set of popular websites has shown that more than 50% of websites contain at least one known vulnerability due to the use of unpatched JavaScript libraries [14]. Open Web Application Security Project (OWASP) publishes 10 most common web vulnerabilities each year, and some of them are also represented by JavaScript libraries [15]. A real example can be found in the jQuery library up to version 1.9.0 which provides `load()` function without properly validating input, which means executing JavaScript code is possible by passing it in a script tag [16].

IV. Detecting JavaScript Libraries

JavaScript libraries can be used in multiple ways, in different forms and thus different approaches can be used to detect them. Alongside fetching libraries from CDNs, they can also be stored on the same server as a website. The library itself can be in a certain form, depending on the need. Apart from the differences in versions there are differences in the way the library is packaged, specifically there are uncompressed, slim and minified types. The types are based on the idea of compressing code to make transfers over the network faster and reduce loading times. In production, minified code is preferred, i.e. code from which comments, empty characters and unused code are deleted in an automated way, and names of variables and functions are shortened [17].

A rough division in detection refers to static and dynamic methods. In general, static analysis refers to the study of source code without its execution, so in this case detection takes place without code execution. The method is often used in malicious code analysis because it is considered to be a safer approach than executing the code. On the other hand dynamic detection refers to recognition of code by its execution. Certain objects that the program creates and functions that it runs can be used to identify the library. What makes this type of detection possible is the absence of namespaces in JavaScript. Consequently, objects and functions are available globally in the code and can be called at runtime. Combination of recognized elements can be used to determine which library it is.

The paper implements two approaches of static JavaScript library detection. Those are detection by cryptographic hashes and detection by identifiers. A hash function is any function that takes an input string of arbitrary length and maps it to fixed length value. The idea of using hash functions comes from the need of truncating the strings being compared, specifically, instead of comparing the entire code of the JavaScript libraries, hash digests can be compared to see if they match. A prerequisite for performing detection by cryptographic hashes is a collection that contains all required JavaScript libraries and their calculated hashes. Then, for an unknown library we can compute a hash and look for a match in the collection. An assumption for using this type of detection is that the libraries are used in their original form, without any modifications. Reason for this is that hash functions don't tolerate even slight changes. This might be seen as a limitation, but procedures are implemented for reducing libraries to their canonical form, so slight modifications are not taken into account.

An alternative approach, in static detection, is detection by identifiers. It refers to extracting keywords from JavaScript code, where the keywords are identifier tokens obtained by lexical analysis. While collecting identifiers of a library it is important to take into account only those identifiers that do not appear in other libraries, because then they would not be unique, neither would they identify a given library. The assumption is that the libraries are relatively long scripts and that there are many identifiers, therefore the detection will be proportionally reliable. More identifiers means more diversity, which in turn suggest less chance of falsely detecting libraries.

The same can not be said for library plugins, because their code is often relatively short. For this reason, only libraries and plugins with significant amount of identifiers are capable of being reliably detected. To prevent false positive detections, libraries and plugins with short source codes are discarded.

This method, by its nature, is not resistant to situations where arbitrary JavaScript code that contains a large number of identifiers of some known library, is falsely detected as a known library. The disadvantage of detection by identifiers is the computational complexity. For larger library collections with many identifiers that must be compared to a multitude of unknown library identifiers a combinatorial explosion occurs. A large number of seemingly simple operations can drastically slow down detection, so it was necessary to consider more advanced measures during implementation.

V. Developing System for Detection

Developed software solution collects JavaScript libraries from CDN servers and builds a collection which serves as a basis for detecting unknown libraries. The same collection is supplemented with computed hashes and additional information about the libraries. The program implements three static detection mechanisms which are optimized and run on multiple processor threads.

The program is divided into units according to the functions performed. The system architecture and communication stakeholders overview is shown in Fig. 1.

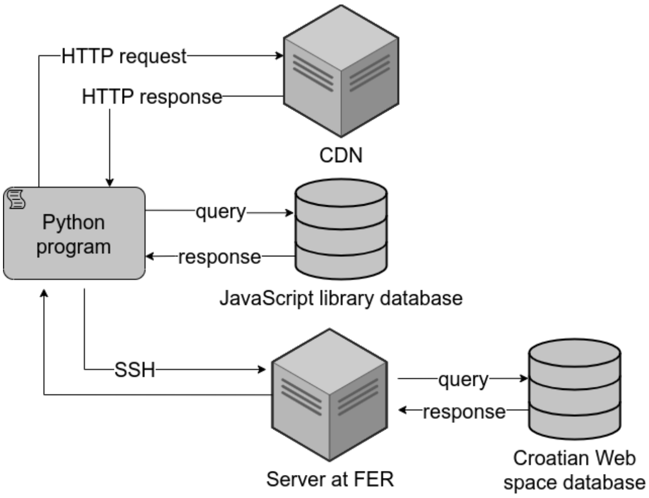
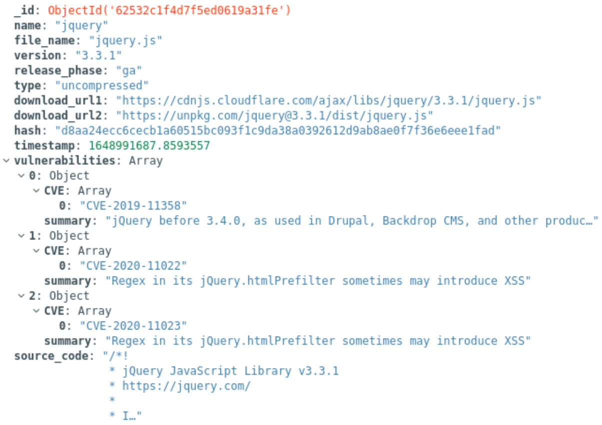


Fig. 1: System architecture components

While creating or updating the database, program communicates with CDN server and obtains library source codes, which are then stored and processed. The second part of communication is with FER server which provides access to a database of crawled web pages and used JavaScript libraries.

The database itself is based on 26 most used JavaScript libraries that have, or have had in past versions, publicly known vulnerabilities [5]. What enables automated library code fetching is a web scraper. It is programmed to retrieve all search results from CDNs for a given query, query being a library name. It collects all libraries and plugins related to the selected libraries and also all their versions and subtypes. All retrieved data is processed sequentially, for each request library name, type and version are extracted. Version semantics used is `major.minor.patch`, while types are organized by level of compression, so there are uncompressed, slim and minified types. Also, cryptographic hash digest is calcualted in advance to speed up the algorithm and remove redundant computation. If available, CVE references about library vulnerabilites are associated. This process generates a document as shown in Fig. 2.



```
 _id: ObjectId('62532c1f4d7f5ed0619a31fe')
 name: "jquery"
 file_name: "jquery.js"
 version: "3.3.1"
 release_phase: "ga"
 type: "uncompressed"
 download_url1: "https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.js"
 download_url2: "https://unpkg.com/jquery@3.3.1/dist/jquery.js"
 hash: "d8aa24ecc6cecb1a60515bc093f1c9da38a0392612d9ab8ae0f7f36e6eee1fad"
 timestamp: 1648991687.8593557
 vulnerabilities: Array
  0: Object
   CVE: Array
     0: "CVE-2019-11358"
   summary: "jQuery before 3.4.0, as used in Drupal, Backdrop CMS, and other produc…"
  1: Object
   CVE: Array
     0: "CVE-2020-11022"
   summary: "Regex in its jQuery.htmlPrefilter sometimes may introduce XSS"
  2: Object
   CVE: Array
     0: "CVE-2020-11023"
   summary: "Regex in its jQuery.htmlPrefilter sometimes may introduce XSS"
 source_code: "/*!
              * jQuery JavaScript Library v3.3.1
              * https://jquery.com/
              *
              * I…"
```

Fig. 2: Collection document example

The created collection is sufficient to perform a basic type of detection where only cryptographic hashes of libraries are compared, but for performing more advanced methods, it is necessary to further process the library data. To solve the problem of minimal changes in JavaScript libraries, they need to be reduced to a canonical form, i.e. parts of library code that are not crucial to the code itself are ignored. Precisely, all spaces, tabs, new lines and all comments are deleted. A new hash is computed from processed code and stored in the database. The same procedure is repeated on the unknown library during detection, for both sides to be equally reduced.

Furthermore, it is necessary to prepare data for detection by identifiers. The procedure includes tokenization of the library's source code, where identifiers are extracted by lexical analysis. These include all the names of variables, functions, classes and everything that identifies a certain object and is not specified by the syntax of the language itself. The resulting list of identifiers for each library must be compared to the lists of identifiers of other libraries in the database to remove all that appear in multiple different libraries. The goal is to keep only those identifiers that are truly unique to a given library. The entire described process of data collection and processing results in a database which contains 34 thousand documents. All available libraries, plugins, all their types, subtypes and version of multiple development stages are included.

The three static detection mechanisms extend each other and thus increase the number of detected libraries. The first method takes a cryptographic hash from an unknown library and searches for the same one in a collection of JavaScript libraries. Finding a match means that the unkown library was successfully identified, and has been used in its original form without any changes. This method guarantees absolutely accurate detection, that is, it guarantees that no false positive detections will occur. The amount of detected libraries depends on how complete the database is, meaning the more instances of libraries from different sources are contained the more libraries are going to be detected. The reason for this is that the various services which offer JavaScript libraries make slight changes, so the cryptographic hashes for virtually the same library will differ.

The second method refers to detecting libraries by using cryptographic hashes, but with implemented reduction to canonical form. It is an approach that alleviates the problem of minimal changes in the source code of the libraries. Unknown libraries go through the same procedure of reduction as libraries in the database. This method covers all the results generated by the first method, meaning, it will identify all the libraries as would the first method and more. The implementation and execution of this method comes with the price of complexity and longer runtime.

The third method is based on textual search with the aim of finding identifiers that determine individual libraries. By conducting lexical analysis on the unknown library identifiers are extracted and then the database is searched for matches. Whether the library is detected or not is dependent on the ratio of the number of detected identifiers to the total number of identifiers for a given library. Because of this, the method can not guarantee that false positive detection will not happen. In theory, it is possible that an unrelated JavaScript code contains all the identifiers of a library, and because of it, gets positively detected while it should not have been. During testing no such cases were observed. Results will show that this method works because developers follow good practices and assign identifiers that are meaningful and specific enough so false positives rarely occur. Again, due to the increase in complexity, this algorithm comes with longer runtime than the first two.

VI. Experiments

The experiment was conducted on 5 million documents collected by the web crawler from Croatian web space, which was part of previously conducted research [18]. The dataset consists of 93.03% of HTML documents, 3.41% JavaScript files, 1.98% CSS files, 1.58% other types of files. This file type distribution does not necessarily represent the actual distribution of file types on the web. It is crawler's fetching mechanism that causes it. Further testing is conducted only on JavaScript files, which there are over 170000 that make up the listed 3.41%.

While the developed tool is intended to analyze JavaScript libraries exclusively, besides the libraries, the dataset also contains a significant share of bespoke JavaScript code, i.e. code that is written for a specific website. There is currently no solution to differentiate libraries from other scripts, therefore the experiment is carried out over the entire dataset.

By performing the three detection methods, there were 966 libraries detected using the first method by comparing cryptographic hashes. Secondly, 2870 libraries were detected by reducing them to canonical form and then comparing cryptographic hashes. Thirdly, 16097 libraries were detected using detection by identifiers. As expected, the first method returns the least amount of results. Some of the reasons are minimal changes in the library code and the incompleteness of the database due to the large number of sources for obtaining libraries, where different providers make changes to the original libraries. On the other hand, second method with reduction to canonical form returns almost three times more detections, which proves that the assumed minimal changes are indeed present and frequent. A further assumption is that there are even more libraries with changes that could not be detected, however, there is no heruistic method which would provide information on their quantity. Lastly, the third method, relying on identifiers gives almost six times more positive detections than previous method. The advantage of this method is that minimal changes are irrelevant as long as identifiers are unchanged. The method has even higher potential for solving the problem of minimal changes. On the other hand, this method has the disadvantage that the exact versions of libraries can not be reliably determined, since not many identifiers are changed between versions.

The detection results show that certain libraries are more present than others. Fig. 3 shows the distribution of the 25 most represented libraries that were detected by using the second method with reduction to a common form.

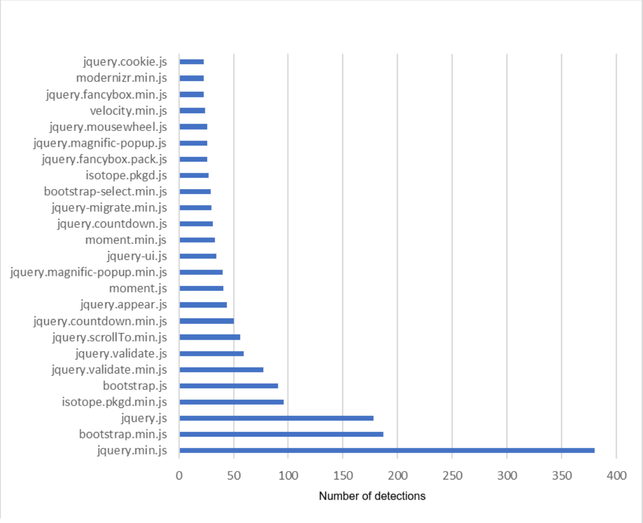


Fig. 3: Most represented libraries detected by reduction to common form and hash comparison

An important note is that minified libraries are used more often. This confirms the assumption from a paper on a similar topic [4] where it was the minified libraries that inhibited detection. Minification poses another problem in detection. Minification can be done by using different tools which will produce different minified versions of the same code, but even the same tool can, while shortening variable names, permute the names and thus brake detection.

Furthermore, detection by identifiers gives a slightly different distribution of detected libraries, the main reason for which is the limited set of data that serve as the basis for this type of detection. The method of extracting identifiers is suitable only for certain libraries, it is necessary that they have a considerable amount of the identifiers and that those are not generic names so that they can truly identify a given library. Fig. 4 shows a summary distribution of the most frequent libraries.

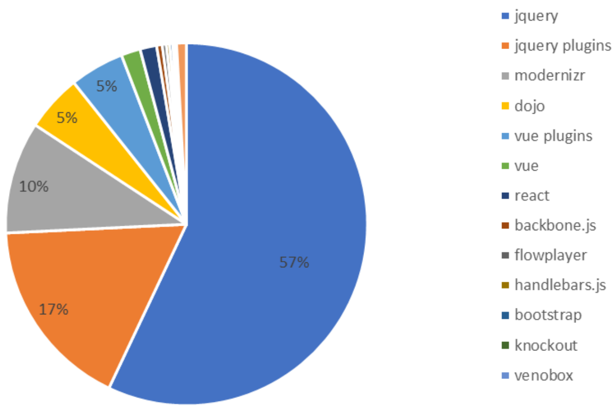


Fig. 4: Most frequent libraries detected by identifiers

The most recognized libraries throughout all methods of detection are jQuery and jQuery plugins. The accuracy of this distribution is also backed by research done on a selected set of websites [12]. It is confirmed that jQuery is certainly the most represented, followed by Bootstrap, Modernizr, React, Backbone. Unfortunately, past versions of these libraries have known vulnerabilities. Among the detected libraries, a non-negligible part of those has vulnerability records in the NVD database. Out of 966 libraries, detected using the first method, 165 (17.08%) of them are vulnerable. Out of 2870 libraries, detected using the second method, 636 (22.16%) is vulnerable. Third method does not provide reliable information about vulnerabilities because it can not determine the library version. Because of this the results rely on cryptographic hash detections. It turns out that jQuery is the most common vulnerable JavaScript library, specifically 80.82% of detected vulnerable libraries are versions of jQuery. A probable cause that lead to this result is that jQuery has over 80 publicly known vulnerabilities according to the NVD database [19], and only the last two versions of the library are considered secure, while all previous versions were proven to be vulnerable.

VII. Discussion

The implemented solution proves the validity of the theory in JavaScript library detection, but in practice certain limitations are imposed. To begin with, detection depends on how complete the database is, only those libraries that are contained in the database can be detected. The root of this limitation comes from the nature of hash functions that require identical inputs for detection to work. This would not be as big of a problem if JavaScript libraries were systematically organized on an official, centralized server. At the same time, all library instances would need to be identical, without minimal modifications. This is not the case, therefore the database should contain libraries collected from all platforms that are commonly used. Furthermore, the problem of minimal modifications introduced on the user side is algorithmically unsolvable. While some user based modifications follow certain patterns, there are plenty of unpredictable changes that are not easily extracted from library code. This problem passes on to detection by identifiers, even though this type of detection is more resilient to modifications, it still produced some false positive results. The solution was to declare libraries as detected only when they had a high percentage of matching identifiers. The percentage value was tuned until no false positive detections occurred, while giving the most results. Another limitation is the lack of organized information about vulnerabilities. Although many organizations invest a lot of effort to make vulnerabilities publicly known and available, the information is not as complete as such program would require. The implemented solution solves the mentioned limitations to a certain extent, but still leaves room for improvement.

VIII. Conclusion

The World Wide Web is of great importance in today's world and there is a need to raise its level of security. Vulnerabilities are proven to exist and are introduced in several ways, one of which is the use of vulnerable JavaScript libraries. A software solution that could detect libraries and determine whether they are vulnerable would be of great help. This paper deals with a static detection approach that detects JavaScript libraries based on comparing cryptographic hashes and identifiers. Experiments carried over the data from Croatian web space shows that many of the popular libraries are being used, with minified ones being preferred.

Detection results greatly improve when methods for resolving the problem of minimal modifications in JavaScript library code are implemented. First detection method is capable of detecting all the libraries that are used as-is. Second detection method implements reduction to a canonical form, and detects libraries despite minimal modifications. Third method is based on comparing identifiers and detects a larger share of libraries with greater amount of modifications. The implementation itself encounters certain limitations such as modifications in the library code and problems of obtaining all the necessary data, but to a certain extent it successfully solves them. To improve the solution, combining different detection methods and using hybrid approaches is necessary.

References

[1] M. Kluban, M. Mannan, and A. Youssef, "On measuring vulnerable javascript functions in the wild," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 917–930.

[2] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable javascript," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 177–187.

[3] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web application vulnerability prediction using hybrid program analysis and machine learning," *IEEE Transactions on dependable and secure computing*, vol. 12, no. 6, pp. 688–707, 2014.

[4] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," *Communications of the ACM*, vol. 61, no. 6, pp. 41–47, 2018.

[5] "GitHub - RetireJS/retire.js: scanner detecting the use of JavaScript libraries with known vulnerabilities." https://github.com/retirejs/retire.js/, [Accessed 13-Jan-2023].

[6] J. Michel, "Library detector for chrome," https://github.com/johnmichel/Library-Detector-for-Chrome, [Accessed 13-Jan-2023].

[7] "Wappalyzer - identify technologies on websites," https://www.wappalyzer.com/, [Accessed 13-Jan-2023].

[8] I. Kovacevic, M. Marovic, S. Gros, and M. Vukovic, "Predicting vulnerabilities in web applications based on website security model," in *2022 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, 2022, pp. 1–6.

[9] L. S. Vailshery, "Most used languages among software developers globally 2022," Aug 2022. [Online]. Available: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

[10] "Dom (document object model) - mdn web docs glossary," [Accessed 13-Jan-2023]. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/DOM

[11] "jquery," [Accessed 14-Jan-2023]. [Online]. Available: https://jquery.com/

[12] "Usage statistics of javascript libraries for websites," [Accessed 14-Jan-2023]. [Online]. Available: https://w3techs.com/technologies/overview/javascript_library

[13] "Cdn (content delivery network) - mdn web docs glossary," [Accessed 13-Jan-2023]. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/CDN

[14] T. Kadlec, "Open source security report - 77% of 433,000 sites use vulnerable javascript libraries," Nov 2017, [Accessed 13-Jan-2023]. [Online]. Available: https://snyk.io/blog/77-percent-of-sites-still-vulnerable/

[15] "Owasp top ten," [Accessed 14-Jan-2023]. [Online]. Available: https://owasp.org/www-project-top-ten/

[16] "Cve security vulnerability database - cve-2020-7656," [Accessed 13-Jan-2023]. [Online]. Available: https://www.cvedetails.com/cve/CVE-2020-7656/

[17] "Minification - mdn web docs glossary," [Accessed 13-Jan-2023]. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/minification

[18] E. Stambuk, S. Gros, and M. Vukovic, "Analyzing web security features using crawlers: Study of croatian web," in *2021 16th International Conference on Telecommunications (ConTEL)*. IEEE, 2021, pp. 142–145.

[19] NVD, "National vulnerability database jquery," [Accessed 17-Jan-2023]. [Online]. Available: https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=jquery&search_type=all&isCpeNameSearch=false