Yaroslav Chuiko, master

National Technical University “Kharkiv Polytechnic Institute”, Kharkiv

Viacheslav Karpenko, candidate of technical sciences

National Technical University “Kharkiv Polytechnic Institute”, Kharkiv

ORCID: 0000-0002-8378-129X Viacheslav Karpenko

**DETECTING THREATS IN THE JAVASCRIPT CODE OF WEB APPLICATIONS**

JavaScript is a dynamic programming language that is used by the vast majority of websites and supported by all modern web browsers. Its prevalence has become one of the factors that in recent years JavaScript has become the most common and successful language for building web attacks. Recent cyberattacks regularly exploit JavaScript weaknesses, and sometimes even mask their malicious intentions to avoid detection. Attackers can embed malicious JavaScript in a web page, and it will be automatically executed when the page is loaded in any browser. All this makes the task of detecting threats in JavaScript code very important.

There are different approaches to malware detection. In solving this problem, it is advisable to use static analysis (known as source code analysis), which tests and evaluates the program by examining the code without executing the program. It is usually used to analyze the code syntax. The goal is to check if there are any suspicious keywords or code fragments [1].

There are various methods for analyzing static source code for potential vulnerabilities, and after analyzing them, lexical analysis was chosen. Lexical analysis transforms the syntax of the source code into “tokens” of information in an attempt to abstract the source code and facilitate its manipulation. This analysis is aimed at recognizing patterns, anomalies, and suspicious content in the data.

To find threats in JavaScript code, the following five-step algorithm was proposed: the stage of obtaining a URL, the stage of loading an HTML page, the stage of searching for <script> HTML elements and extracting JavaScript code from them, the stage of searching for potentially dangerous JavaScript code, and the stage of classifying malicious code.

The first stage involves obtaining the URL of the page the user wants to analyze. A URL is nothing more than the address of a specific unique resource on the Internet. Such resources can be an HTML page, a CSS document, an image, etc.

The purpose of the second stage is to get the HTML page for further analysis. This process takes place using the HTTP GET request method to the previously received URL. HTTP defines a set of request methods that indicate the desired action to be performed on a particular resource. The GET method requests a representation of the specified resource. The result of a GET request is not necessarily an HTML page, so at this stage, the extension of the received file is also checked.

The next step is to search for all JavaScript scripts that are used on the downloaded HTML page. Since on all HTML pages both embedded JavaScript scripts and external JavaScript files can be contained only in special script tags. The search is carried out by parsing the entire HTML page without rendering the page and applying styles that are executed by browsers, as this significantly slows down the search process. After all the <script> elements are found, they are divided into two groups: those that contain embedded JavaScript code and those that contain links to external JavaScript files. This division is based on whether the <script> element contains a special src attribute that contains the URI of a link to external files. After that, for those <script> elements that do not contain the src attribute, the internal content, i.e. JavaScript code, is extracted without prior execution, and for those that do contain the src attribute, GET requests are made to obtain external JavaScript files for further analysis.

At the stage of searching for potentially unsafe JavaScript code, we search for standard JavaScript functions and functions available through the Web API, which can also be potentially dangerous. Potentially unsafe JavaScript functions and the vulnerabilities they can cause are listed in [2]. The search is performed by parsing JavaScript code using regular expressions. From the previous stage, the JavaScript code is presented as lines of text in which the code that may contain potential vulnerabilities is searched.

At the last stage, if potentially dangerous JavaScript functions are found, the algorithm proceeds to the malicious code classification stage. At this stage, the potentially dangerous code found earlier is classified according to the type of attack it can lead to and the overall level of danger. The malicious JavaScript code is classified by the level of danger in such a way that each individual part of the code has its own weight and level of danger. After analyzing the code, the presence of potentially dangerous functions and the level of potential vulnerability are determined.

After classification, the result is presented in the form of the overall level of danger on the site and a list of possible attacks.

Based on this algorithm, a corresponding software solution was developed. It consists of three main components: a server side responsible for the business logic of the application, a client side responsible for the user interface, and a database server responsible for storing data. The user on the client side enters the URL of the site he wants to check for danger. After entering the correct URL, this value will be sent to the server where the code scanning process will begin. After successful completion of the scan, the user will be redirected to a page where the scan results will be displayed.

Thus, the proposed algorithm and the corresponding software solution allow an ordinary user to significantly increase the security of using third-party web applications.

References

1. Dynamic Analysis vs. Static Analysis // https://www.intel.com/content/www/us/en/docs/inspector/user-guide-windows/2022/dynamic-analysis-vs-static-analysis.html, 02.10.2023.

2. CSSXC: Context-sensitive Sanitization Framework for Web Applications against XSS Vulnerabilities in Cloud Environments // https://www.researchgate.net/publication/303745888\_CSSXC\_Context-sensitive\_Sanitization\_Framework\_for\_Web\_Applications\_against\_XSS\_Vulnerabilities\_in\_Cloud\_Environments, 24.10.2023.