IAES International Journal of Robotics and Automation (IJRA)
Vol. 13, No. 1, March 2024, pp. 96~104
ISSN: 2722-2586, DOI: 10.11591/ijra.v13i1.pp96-104

Script late injection: a framework to introduce JavaScript into web pages

Bhanu Prakash, Sandhya Sampangiramaiah
Department of Computer Science and Engineering, RV College of Engineering, Bengaluru, India

Article Info

Article history:

Received Jul 17, 2022
Revised Aug 10, 2023
Accepted Nov 27, 2023

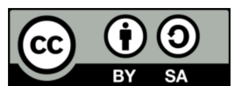
Keywords:

Hypertext markup language tags
Injection parsers
Proxy servers
Script injection
Web security

ABSTRACT

Script injection is one type of fault present in web, which mostly utilizes user data to execute code without applying any type of filters. Script injection can impact both client and server making exposing them to vulnerabilities. Security and related products may need to execute logic on the client-side generally in a browser. In order to achieve this, proxy servers inject appropriate JavaScript code into the responses they proxy. Typically, the injection point is at the end of the body element. The framework introduced in this paper rather uses a stack-based approach to determine the injection point in the web page. Ten kilobytes from the end of a web page are given as a string input to the framework, after tokenization and construction of the vector of tokens. A stack is used to determine the injection point. Along with the position of the injection point, a warning flag is also estimated indicating the correctness of the injection point. Different types of web pages were considered for running the unit tests and fuzzy tests on the framework. These classes of pages are determined by crawling most used web pages. The injected scripts are executed once the body content is completely loaded. Hence, it can retrieve maximum information without affecting end-user performance. It also does the job at a low cost.

This is an open access article under the CC BY-SA license.



Corresponding Author:

Bhanu Prakash
Department of Computer Science and Engineering, RV College of Engineering
Mysore Road, RV Vidyaniketan post, Bengaluru, Karnataka, 560059, India
Email: bhanuprakashshettigar@gmail.com

1. INTRODUCTION

Asynchronous JavaScript and XML (AJAX) has gained major attention in the current day with Web2.0 [1]. One major concern in software development is accessibility and accessibility of any website can be improved with JavaScript being used suitably with utmost care. JavaScript injection is the inclusion of one's own code in a webpage by checking the cross-site scripting (XSS) vulnerability in any website. These changes can only be seen at the client end since it is a 'client-side' language. Some common methods to inject are, employ developers console for inserting script, inclusion of the script within the address bar, inclusion of scripts into comment field or other forms – XSS.

JavaScript is enabled and supported by default in most of the major browsers. It is a widespread tool employed to develop browser interfaces since it is simple and is fully integrated with HTML/CSS. Communication over internet to the server can be done with ease regarding the origin of the current page. But the ability to obtain data from other sites is crippled. Even though this should be possible, it becomes essential to enter into an explicit agreement from the remote side which is a safety limitation.

Web security recommends defensive measures and protocols that organization must embrace in order to protect themselves from cyber-attacks and dangers of utilizing web channels. To implement web security, it is important to get essential information from the end user. This is where proxy servers come in.

They inject script into the responses and then forward it to the end user. This entire transaction is shown in Figure 1. A proxy server is a computer framework or switch that facilitates a transfer between client and server. A caching hypertext transfer protocol (or HTTP) proxy handles the HTTP/HTTPS requests that arrive at the server to serve content to the end user. The proxy also has web application firewall capabilities, where the traffic is inspected, and security policies are enforced.

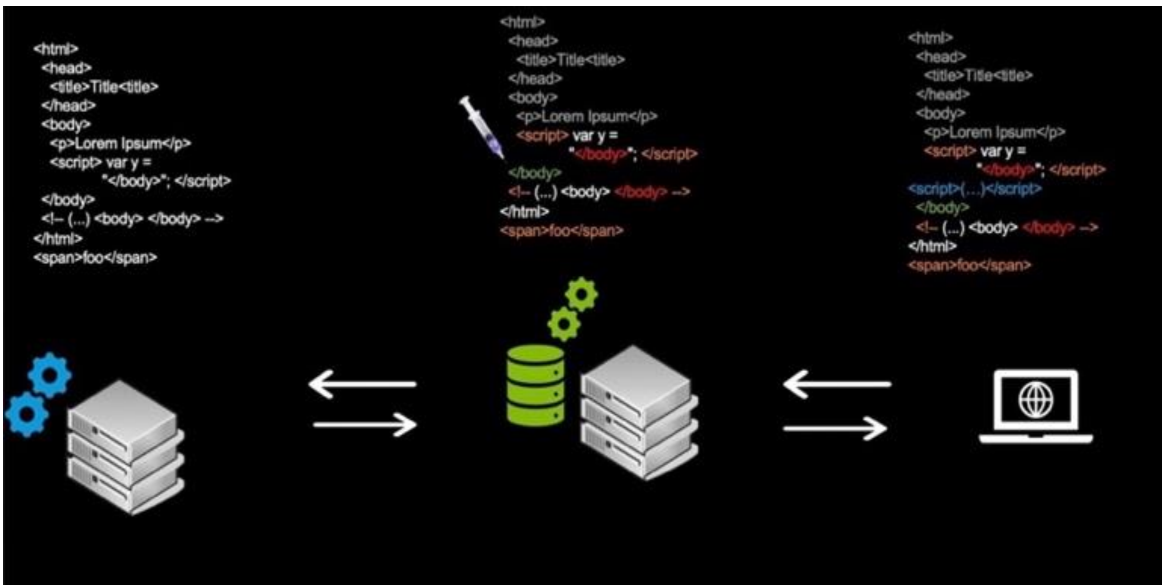


Figure 1. A request response exchange with reference to script late injection

Script late injection (SLI) is a framework which can be used by the proxy servers to inject JavaScript at appropriate position in the page based on the last N kilobytes, generally 10 kilobytes of chunk from the hypertext markup language (HTML) page. Most common injection point is at the end of body element (at the ending </body> tag). Among other things, this helps to achieve low impact to end user's performance. Two approaches were used so far. First, Parse the whole page from the start using page modify parser, locating tags </body>, inject "<script>...</script>" before that. Second, Treat the page as plaintext, search string "</body>" and replace it with "<script>...</script></body>". Both have shortcomings in functionality and most importantly processing cost. Script late injection (SLI) solution is offered which is developed as standalone library and later integrated with the proxy server configurations.

Mao *et al.* [2] uses identification of function-level execution information model for application vulnerabilities. It is possible for attackers to inject the code and activate it under the same conditions as in the original app. In such cases, this approach may miss the injection within a target function. Li *et al.* [3] explains data object model analysis-based method to crawl the web applications using JavaScript injection and runtime analysis. This raises difficulties, for example, the parallel and cross-browser compatible crawling, specific domain websites crawling which demand an aide from domain information, focused website crawling and so on. Control of traffic through traffic analysis [4] is demonstrated for a given pattern of traffic request at a proxy server that suits the analytic model. An ideal situation is assumed with finite size of web proxy cache memory. The approach employed limited number of files to determine bandwidth requirement considering hit rate and hit misses however packet jitter is proposed to be included for matching up with the real-world scenario. Kishore *et al.* [5] uses drive by download defense model for preventing malicious JavaScript injection attacks. This is a browser extension and hence needs enabling of extensions in browser, which is not available in all scenarios. Extensions are not available for mobile devices, and all the browsers do not support it.

Barua *et al.* [6] utilizes code randomization model combined with a static analysis strategy to forestall JavaScript infusion attacks. This technique has no offline analysis, possibilities of incorrectly modeling the internal application programming interfaces (APIs) of the browser, runtime protection framework. The browsers when deliver the embedded non-HTML files that contain malicious JavaScript code in the form of HTML files, would result in the possibility of content sniffing attacks [7]. Though there are approaches to handle these kinds of attacks, the existing approaches face certain limitations. Therefore, an approach to simulate the behavior of browser via sample download testing and detection mechanism at server

side to detect content sniffing attack is addressed. The proposed approach is evaluated on real world PHP programs and obtained results indicate that sniffing attacks could be overcome by preventing the upload of malicious files.

Shahriar *et al.* [8] uses comment injection-based model for preventing JavaScript injection attacks. Randomness in JavaScript comments is not reliable parameter for detection of injected code. Pre-processing of JavaScript, for example, the expulsion of comment tags and deletion of return keywords in event handlers is computationally expensive and could take additional time. Several advanced features of browsers were initially extensions from third party. Though these features provided some of the powerful extensions to browser, these if employed inappropriately could lead to security threats. Hence few browsers restricted the extension capabilities. Therefore the work [9] proposes verified security for browser extensions in form of high level languages in contrast to API based approach. Comparison of attacks built using web content filtering policies with content-sniffing XSS attacks is presented with reference to a scenario of conference management system is presented. Content sniffing algorithms [10] considering 3 browsers for specific web applications is proposed that describes filter based models that block XSS attacks for content sniffing. Due to increased popularity of XSS in web developers who fail to validate user data, poses some crucial security threats to web applications. Secure web application proxy [11] is described that detects and prevents XSS attacks at the server side. The HTML responses are caught by a reverse proxy that detects and mitigates several vulnerabilities.

Browser extensions based on JavaScript [12] are popular among browsers like Firefox and its related tools. Current attacks illustrate that JavaScript extensions (JSE) pose threats to browser security, therefore security architecture for browser extensions that tracks information in-flow and detect if malicious content is passed is described. The approach mainly dealt with explicit flow of information and cross-domain flows in Firefox browser. Further this was evaluated on JSE that contained both malicious and benign cases that also analyzed the runtime overheads. When malicious content is embedded in browser extension, the attacker would also modify the display mode to make the effect of malicious display transparent to users. Therefore abnormal visibility [13] that proposes display features to highlight embedded malicious code. This approach is observed to provide higher efficiency with reduced cost for maintenance.

Li *et al.* [14] uses chunk dependence graph-based model for real-time content transformation of web pages. Content transformation of streaming web data might increase the performance but there is no security aspects described. Byte-streaming, chunk-streaming, and whole-file buffering methods are susceptible to serious web attacks. In view of optimizing bandwidth and improving performance at the server side approaches such a proxy servers [15] or caching servers are introduced. The functionality of proxy servers was extended to match up the current web 2.0 trends. As compared to reusing of passive cache objects by the proxy servers functionalities such as prefetching content and distribution of cached content was researched and proposed as next generation proxy servers. As the demand of services catered on web is increasing, wide variety of services on web/web applications are gaining popularity. To match these demands good caching policies are being researched. Since traditional caching do not meet the requirements of web 2.0 due to varied reasons various cache replacement [16] algorithms are discussed. These cache replacement algorithms [17] are mainly dealt considering the key parameters to make them possible for adapting as policy for proxy servers. The main consideration here is for any sample of the workload, the approach needs to adjust dynamically the changes in the sample. The approaches thus employed for caching in proxy servers achieve reduced miss rates, increased hit rates that leads to reduced cost incurred while a miss happens in cached environment. The approach of caching suitably addresses the performance for mobile environments [18]. Mobile devices are constrained by battery and hence mostly get disconnected quite frequently for optimizing the battery usage. To add on these issues bandwidth is also limited in such environments. In order to address these issues an IR-based cache invalidation algorithm that effectively brings down the latency and bandwidth usage is proposed that is observed to significantly improve the throughput. When the web application accesses an object that is not in local cache, it is to be downloaded and transmitted from the server which incurs cost in terms of latency, bandwidth and power. The main aim of any approach is bring down these while fetching/prefetching the data item. Data prefetching [19] schemes that effective communicate data from server over wireless channel that experience fluctuation to mobile terminals is extensively researched. Such a scheme aims to dynamically prefetch data by choosing an appropriate power level for transmitting data and also exhibit potential trade-offs in terms of performance.

Hung *et al.* [20] uses adaptive proxy-based web content transformation model. Data-type specific content transformation, such as selective filtering and compression are not enough for effective security and performance. The transformations mentioned are data specific and does not involve security enhancement or performance enhancement. Although citations mentioned in the literature review analyzed the injected JavaScript, or the content transformations, they never posed a solution for determination of injection point for JavaScript, which is addressed in this paper.

Response time reduction at the proxy servers can be realized through caching. Reducing response time by means of an effective cache replacement algorithm forms crucial in optimizing the performance metric. Though these cache replacement algorithms use a subset of documents it becomes critical that they maintain consistency of these documents that are cached. Unified cache replacement algorithms [21] that integrates consistency in the replacement algorithms is discussed along with performance study is presented.

Multimedia access over WWW requires guarantee of quality of service (QoS) that transforms the content adaptively based on the agreement of bandwidth by estimating the network conditions. Hence the discussion on negotiating the bandwidth [22] for guaranteeing the web presentation delivery that ensures the requirements in a dynamic network for matching faster and reliable access is presented. The concept of spatial and temporal locality [23] is modeled for the requests that coming on to web servers is discussed. Self-similarity is employed for modeling the spatial locality of references. It was shown through the results that the self-similarity in traces of requests to web servers show correlation in the data set. This correlation in the distribution aid in achieving better cache performance. Performance gains for distributed information systems is proposed through two different approaches viz data dissemination mechanism and speculative service [24], while the former propagates the information from the producer to consumer which in turn reduces the traffic and balances the load, the later serves the other end with rest of the other documents along with the one that is requested based on the property of spatial location.

Based on desired grade of service, an approach for determining the bandwidth requirement [25] for both type of connections included the individual and multiplexed connections is described. Control procedures are invoked for optimizing the network usage along with using the link metrics. In order to minimize the amount of raw data accessible at various levels of IoT architecture, an approach the employs fog and edge computing [26] is described. Though this approach is shown to be efficient it is vulnerable to device level attacks.

2. RESEARCH METHOD

In this section, the approach undertaken in this paper is discussed comprehensively. The following activity diagram in Figure 2, shows the different processing stages in the framework after generation of proper HTML from the input. A string (interpreted as 8-bit byte stream) of size N kilobytes (by default 10 kilobytes) is given as input to the framework, which essentially is the last N kilobytes of the HTML page to which script needs to be injected. A token represented by the name of the tag, type of the tag (open/closed) and the position in the given string. The default *end_tag_html* is initialized as EOF which points to the end of file.

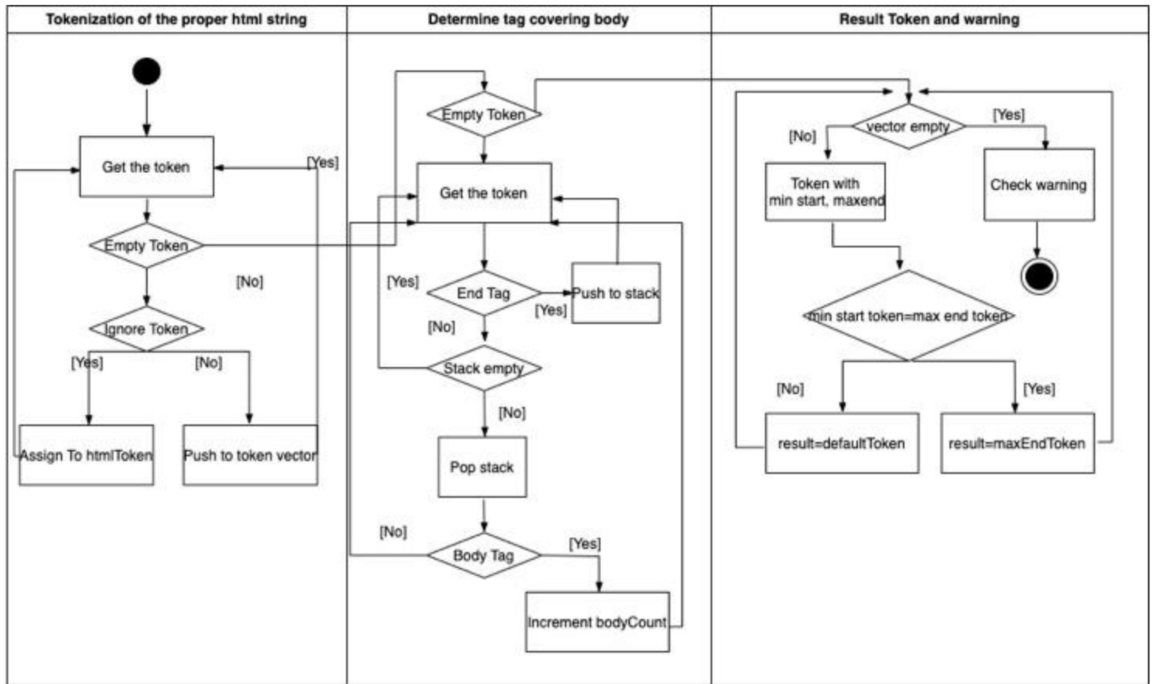


Figure 2. Different processing stages in the framework after generation of proper HTML from the input

2.1. Proper HTML string generation from input

To minimize the cost a fixed size (generally 10 kB) chunk from end of HTML file is considered to detect the injection point. Most of the pages are larger than 10 kB. So, construction of proper HTML is an important step in this algorithm. HTML string received is tokenized and required tokens are extracted. The input string is tokenized into HTML tokens. The loop is executed until all the tokens that are generated by the tokenizer is used.

The list shown in the Figure 3, includes those tags which are self-closing (empty element) because these tags obviously will not enclose the HTML content. Tags which appear inside head and other tags like script style will not contain any HTML. They are used for scripts/styling in the HTML Script, Canvas, NoScript, and Style. These tags either used for manipulating document object model (DOM) or for styling the page. Hence, inside they have CSS or JavaScript, or canvas scripting API code. Hence, these tags are ignored. Both the start and end tags are ignored as per the list. In each iteration, token obtained from tokenizer is checked if it is a token to be ignored (self-closing tags and tags in head section), if it is not, then the token is pushed to a vector for further processing. Then it goes for the next token obtained from tokenizer.

```
["area", "style", "noscript", "template", "base", "br", "col", "embed", "hr", "img", "input", "keygen",
"link", "meta", "param", "source", "track", "wbr", "script", "head", "title", "html", "canvas"]
```

Figure 3. List of self-closing tags

2.2. Determining the missing opening tags in the input string

Consider the input string as last N kilobytes of HTML shown in Figure 4. Tokenization of the input generates the set of vectors shown in Figure 5. The vector of tokens is reversed. For each token in the vector if it is an ending or closing tag it is pushed to the stack, if it is a starting or opening tag the token at the top of the stack is popped after checking if it has the same name as shown in Figure 6.

pt></body>'</script> <h1></h1> </body> </html>

Figure 4. Input string to the SLI framework

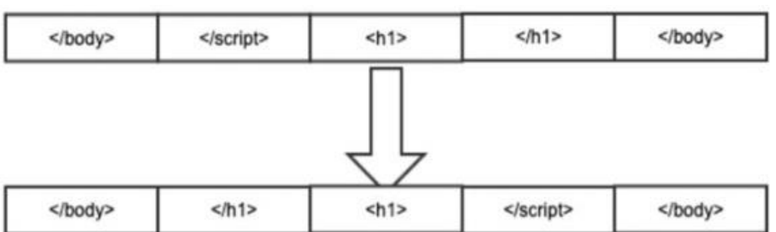


Figure 5. Vector of tokens

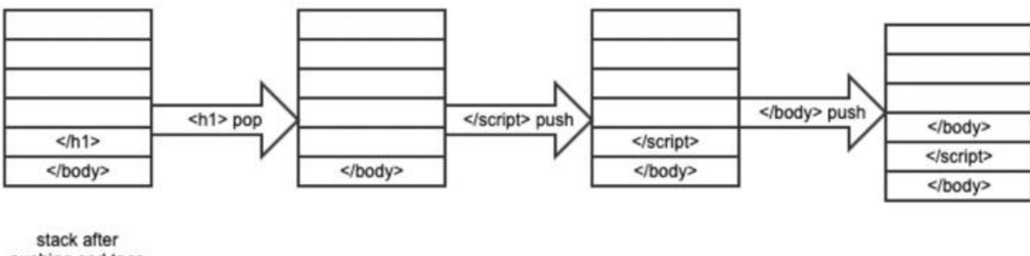


Figure 6. Stack after push and pop of tags

Now the tokens remaining in stack are those ending tags which do not have corresponding starting tags. Until the stack is empty, each token is popped, and corresponding starting tag is appended to the vector of tokens as shown in Figure 7. This forms the vector of tokens which contains “proper” HTML content. Now reverse these tokens again. An empty string is initialized. For each token in vector, if it is a starting tag string of *“<“+name+“>”* is appended to string, else, string of *“</”+name+”>”* is pushed to string which gives the preprocessed string. Finally, the pair of original vectors of tokens and the preprocessed string is returned as shown in Figure 8.

Figure 7. Vector of tokens

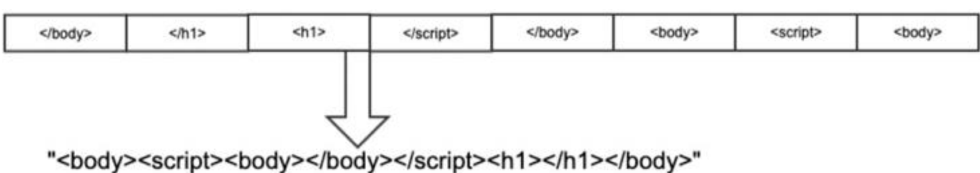


Figure 8. Proper HTML string

2.3. Determination of resultant token and warning flag

A variable for resultant token is initialized to end of the file if *</html>* is absent. Otherwise, it is initialized to *</html>*. Tokenize the tags again by the approach explained in previous step. Again, using the same approach pop closing tags for encountering its corresponding opening tags until the stack is empty, if the last token popped is *</body>* then it is initialized as resultant token. In other cases of resultant token warning is raised. Warning flag is raised if any of the conditions given in Table 1 is true. Position of result token and warning flag is returned as output from the algorithm. For the given example string the result is: (21, false).

Table 1. Conditions for raising warning flag

Sl. No.	Condition
1	If body token is not the result token.
2	If there are more than one body tags
3	If html end tag is absent
4	If there are trailing data (comments, html data) after end of body tag or html tag
5	If there are attributes in html or body end tags (which is uncommon)

3. RESULTS AND DISCUSSION

After the development of this framework, it was important to make sure it works not only against the standard HTML endings but also for the variety of classes of tags defined in Table 2. A repository classifying 500,000 most used websites into these categories was prepared in the preliminary stages of this project. The “*” indicates the position of injection and “(…)” indicates the rest of the data. Therefore, Google test was used to test the SLI framework. Unit tests were developed for all varieties of page endings found in the Table 2 which are verified against the result from the SLI framework. The position of injection as well as warning is checked.

Miscellaneous tags like nested body tags are also tested. Fuzzing tests are written with the help of Domato, an open-source DOM fuzzer from google which generates HTML string in run time. Custom HTML strings are also generated by Domato just by changing the configurations in the template file, and tested. Component tests were written for the methods defined in the framework which were responsible for preprocessing the HTML string, and to determine the trailing tags, attributes, or comments to raise the warning.

Table 2. Varieties of HTML endings

Sl. No.	Type of Ending	Example
1	Basic	(…) *</body>/html>
2	Trailing data with html tags	(…) *</body>/html>XPM
3	Trailing data with comments	(…) *</body><!-- hi -->/html>><!-- hi -->
4	Implicit	(…) *</html>
5	Multiple body tags within and out of comments/scripts	(…) </body>(…)><!--(…<body> and </body>)(…) -->(…)*</body></html> (…)<script>document.write("(…)</body></html>")</script>*</body></html>

As mentioned in introduction section, standard approaches for script injection that were used so far are page modify parser and find and replace. Page modify Parser tokenizes entire HTML page into set of tokens then identifies the point of injection as specified by the metadata in the request received or the configurations in the proxy. Later, it injects the script at the identified position. For example, if the metadata indicates to inject before *</body>*, then *<script>...</script>* will be injected before that. Find and Replace approach considers the entire page as plain text then processes the page character by character to find the required tag to replace. Later, it will be replaced with concatenation of script that is to be injected and original tag. For example, if the metadata indicates to inject before *</body>*, then it will be replaced with *<script>...</script></body>*.

It is important for any computer algorithm, especially for the one that operates on the fly and has significant impact on user experience to be efficient and reliable. To measure the efficiency, web pages of size from 500 kB to 4 MB are considered. On each of these web pages, injection of JavaScript at the end of the body tag is performed using script late injection parser, page modify parser and find and replace parser respectively. Space efficiency and time efficiency of approaches to script injection is shown in Figure 9 and Figure 10 respectively. Each of the data point represents arithmetic mean over twenty iterations. For a web page of size 2 MB, script late injection takes 207872 bytes to complete the transformation, page modify parser takes 7261923 bytes to complete the transformation, while find and replace takes 4034205 bytes to complete the transformation. For a web page of size 2 MB, script late injection takes 0.004132 seconds to complete the transformation, page modify parser takes 0.01545 seconds to complete the transformation, while find and replace takes 0.03456 seconds to complete the transformation. From both point of analysis script late injection parser performs way better than other two parsers. From the above analysis, it is vividly justified that script late injection is better than other two approaches in both time and space efficiencies.

The limitation of this approach is, if the window picks last 10 kilobytes from the arithmetic '<' or '>' symbol in the JavaScript section of the HTML page, there might be problem to determine the tags. This is because the tokenizer considers '<' as start of a HTML tag. The subsequent stream of characters is treated as name of the tag until '>' symbol is discovered again. This might cause the tokenizer to crash or generate erroneous opening tag. Consider the following example *<script> if (x<y) console.log("hi") </script>*. In this case the tokenizer finds tags as *<script>, < y) console.log("hi") </script>* while the expected tags are *<script>,</script>*. This problem is handled by using a static map at present which contains all the existing HTML tags, but this cannot be a permanent solution as HTML tags can depreciate or new tags can be added over time which might be ignored by the tokenizer.

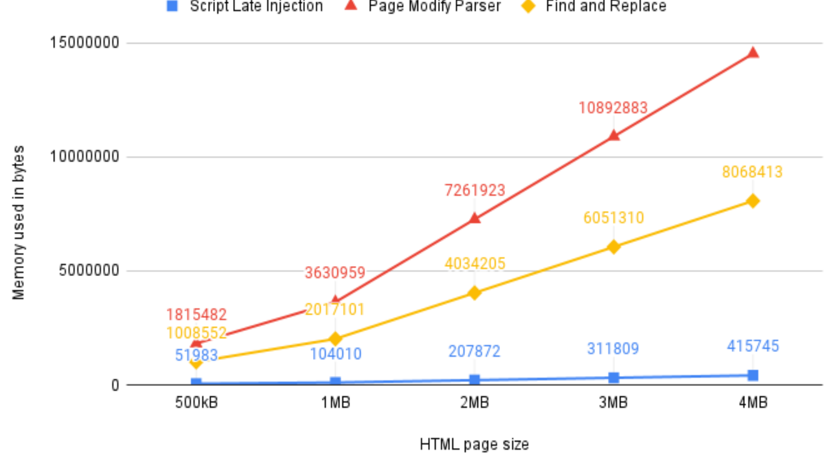


Figure 9. Space efficiency of approaches to script injection

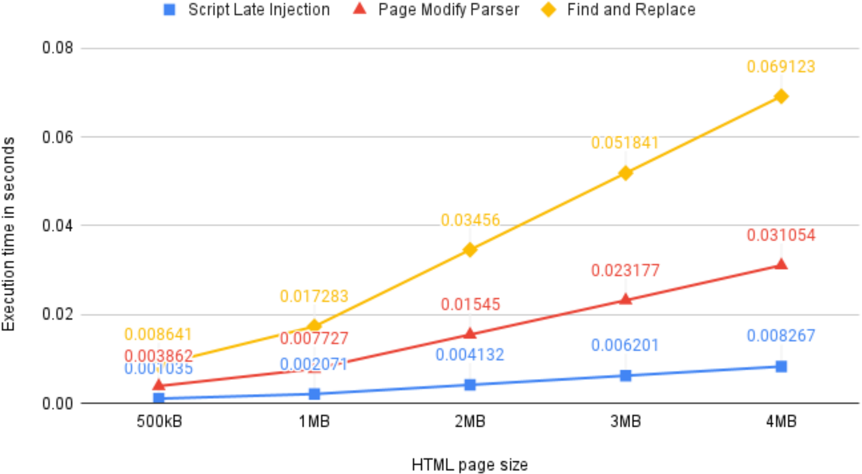


Figure 10. Time efficiency of approaches to script injection

4. CONCLUSION

In this research paper a novel approach was proposed for script injection to the web pages by proxy servers for security and other purposes. The framework designed is not only faster than other two approaches but also is cheaper in terms of computational cost. This approach of injection executes the JavaScript once the complete page is loaded and hence it retrieves the maximum information from the end user with minimal impact on their performance. The results conclusively show that the SLI parser performs better than the conventional parsers that are present in the industry. Future enhancements can be done to eliminate the limitation of this approach.

ACKNOWLEDGEMENTS

I would like to offer my profound gratitude to Mr. Krzysztof Baczkowski and Mrs. Ramya Naidu, supervisors of my work, for their valuable guidance, relentless support, and constructive criticism on this research project. I likewise want to thank Dr. Sandhya S, for her recommendation and help with keeping my project on track with time. My sincere gratitude also to Akamai Technologies who gave me the opportunity to work on this project.

REFERENCES

[1] T. O'Reilly, "What is web 2.0?: design patterns and business models for the next generation of software," *The Social Media Reader*, pp. 32–52, 2012.
[2] J. Mao *et al.*, "Detecting malicious behaviors in JavaScript applications," *IEEE Access*, vol. 6, pp. 12284–12294, 2018, doi: 10.1109/ACCESS.2018.2795383.
[3] Y. Li, P. Han, C. Liu, and B. Fang, "Automatically crawling dynamic web applications via proxy-based JavaScript injection and runtime analysis," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, Jun. 2018, pp. 242–249, doi: 10.1109/DSC.2018.00042.
[4] S. Kanrar and N. K. Mandal, "Traffic analysis and control at proxy server," in *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, Jun. 2017, pp. 164–167, doi: 10.1109/ICCONS.2017.8250702.
[5] K. R. Kishore, M. Mallesh, G. Jyostna, P. R. L. Eswari, and S. S. Sarma, "Browser JS Guard: detects and defends against Malicious JavaScript injection based drive by download attacks," in *The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014)*, Feb. 2014, pp. 92–100, doi: 10.1109/ICADIWT.2014.6814705.
[6] A. Barua, M. Zulkernine, and K. Weldemariam, "Protecting web browser extensions from JavaScript injection attacks," in *2013 18th International Conference on Engineering of Complex Computer Systems*, Jul. 2013, pp. 188–197, doi: 10.1109/ICECCS.2013.36.
[7] A. Barua, H. Shahriar, and M. Zulkernine, "Server side detection of content sniffing attacks," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, Nov. 2011, pp. 20–29, doi: 10.1109/ISSRE.2011.27.
[8] H. Shahriar and M. Zulkernine, "Injecting comments to detect JavaScript code injection attacks," in *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, Jul. 2011, pp. 104–109, doi: 10.1109/COMPSACW.2011.27.
[9] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy, "Verified security for browser extensions," in *2011 IEEE Symposium on Security and Privacy*, May 2011, pp. 115–130, doi: 10.1109/SP.2011.36.
[10] A. Barth, J. Caballero, and D. Song, "Secure content sniffing for web browsers, or how to stop papers from reviewing themselves," in *2009 30th IEEE Symposium on Security and Privacy*, May 2009, pp. 360–371, doi: 10.1109/SP.2009.3.
[11] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "SWAP: mitigating XSS attacks using a reverse proxy," in *2009 ICSE Workshop on Software Engineering for Secure Systems*, May 2009, pp. 33–39, doi: 10.1109/IWSESS.2009.5068456.
[12] M. Dhawan and V. Ganapathy, "Analyzing information flow in JavaScript-based browser extensions," in *2009 Annual Computer Security Applications Conference*, Dec. 2009, pp. 382–391, doi: 10.1109/ACSAC.2009.43.

[13] B. Liang, J. Huang, F. Liu, D. Wang, D. Dong, and Z. Liang, "Malicious web pages detection based on abnormal visibility recognition," in *2009 International Conference on E-Business and Information System Security*, May 2009, pp. 1–5, doi: 10.1109/EBISS.2009.5138008.

[14] X. Li, J. Li, H. Xie, and C. Chi, "Modes of real-time content transformation for web intermediaries in active network," in *2005 IEEE 7th Workshop on Multimedia Signal Processing*, Oct. 2005, pp. 1–4, doi: 10.1109/MMSP.2005.248558.

[15] W. V. Wathsala, B. Siddhisena, and A. S. Athukorale, "Next generation proxy servers," in *2008 10th International Conference on Advanced Communication Technology*, Feb. 2008, pp. 2183–2187, doi: 10.1109/ICACT.2008.4494223.

[16] A. Balamash and M. Krunz, "An overview of web caching replacement algorithms," *IEEE Communications Surveys & Tutorials*, vol. 6, no. 2, pp. 44–56, 2004, doi: 10.1109/COMST.2004.5342239.

[17] H. Bahn, K. Koh, S. H. Noh, and S. M. Lyul, "Efficient replacement of nonuniform objects in Web caches," *Computer*, vol. 35, no. 6, pp. 65–73, Jun. 2002, doi: 10.1109/MC.2002.1009170.

[18] G. Cao, "A scalable low-latency cache invalidation strategy for mobile environments," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, pp. 1251–1265, Sep. 2003, doi: 10.1109/TKDE.2003.1232276.

[19] S. Gitzenis and N. Bambos, "Power-controlled data prefetching/caching in wireless packet networks," in *Proceedings Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, pp. 1405–1414, doi: 10.1109/INFCOM.2002.1019391.

[20] C. C. Hung and L. Y. Hong, "Adaptive proxy-based content transformation framework for the world-wide web," in *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, 2000, pp. 747–750, vol. 2, doi: 10.1109/HPC.2000.843537.

[21] J. Shim, P. Scheuermann and R. Vingralek, "Proxy cache algorithms: design, implementation, and performance," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 4, pp. 549-562, 1999, doi: 10.1109/69.790804.

[22] C. C. Hung and L. Yan-Hong, "Dynamic application-level bandwidth negotiations for WWW," *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, 2000, pp. 726-731 vol. 2, doi: 10.1109/HPC.2000.843534.

[23] V. Almeida, A. Bestavros, M. Crovella and A. de Oliveira, "Characterizing reference locality in the WWW," *Fourth International Conference on Parallel and Distributed Information Systems*, 1996, pp. 92-103, doi: 10.1109/PDIS.1996.568672.

[24] A. Bestavros, "Speculative data dissemination and service to reduce server load, network traffic and service time in distributed information systems," *Proceedings of the Twelfth International Conference on Data Engineering*, 1996, pp. 180-187, doi: 10.1109/ICDE.1996.492104.

[25] R. Guerin, H. Ahmadi and M. Naghshineh, "Equivalent capacity and its application to bandwidth allocation in high-speed networks," in *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 7, pp. 968-981, Sept. 1991, doi: 10.1109/49.103545.

[26] M. Hagan, F. Siddiqui and S. Sezer, "Enhancing security and privacy of next-generation edge computing technologies," *2019 17th International Conference on Privacy, Security and Trust (PST)*, 2019, pp. 1-5, doi:10.1109/PST47121.2019.8949052.

BIOGRAPHIES OF AUTHORS

Bhanu Prakash is working as a software engineer with the Security Ghost Team, in application security subdivision of Security Technology Group, Akamai Technologies. Bhanu holds a Bachelor of Engineering degree in computer science from Rashtreeya Vidyalaya College of Engineering, Bangalore. His research interests include application security, machine learning at edge, and threat analysis. Bhanu Prakash has significant contributions in development, verification, and continuous monitoring of security features of Akmai Edge. He has been awarded with Certificate of Excellence for completion of Project from Samsung PRISM during April 2021 to Feb 2022. He can be contacted at: bhprakas@akamai.com.

Sandhya Sampangiramaiah is working as an assistant professor (senior scale) in the Department of Computer Science and Engineering, RV College of Engineering. Her research interests include networking, genetic algorithm and optimization, security, deep learning, and high-performance computing. She has worked on consultancy projects funded by Cisco Pvt Ltd., Citrix R&D India Pvt., Samsung Pvt Ltd. She is currently working on a research project funded by the Government of Karnataka's K-Tech Centre of Excellence in Cybersecurity (CySecK). She has published more than 25 paper publications in both international journals and international conferences. She is working as member of editorial board in open access peer reviewed international journals and worked as reviewer for several IEEE international conferences and Scopus journals. She has worked as Advisory/Technical Program Committee Member and has been session chair for several IEEE international conferences. She has been Awarded Certificate of Excellence for completion of project from Samsung PRISM during January 2021 to February 2022. She can be contacted at: sandhya.sampangi@rvce.edu.in.