Exploring Cross-Site Scripting (XSS): Attack Payloads, Prevention, and Mitigation Techniques

L. Joshua Crotts
Department of Computer Science
University of North Carolina at Greensboro

***Abstract*—Accelerated growth in the popularity of web applications has inadvertently prompted attackers to switch gears from compromising lower-level systems and applications. Indeed, certain web applications rely on security, e.g., banking websites and social media platforms. Correspondingly, such a reliance provides a great incentive to attackers to steal confidential information such as cookies, credit card detail, and private messages from victims who use these applications. In this paper, we will describe cross-site scripting (XSS) attacks: a modern plague against unknowing users and web developers alike. Additionally, we explain and survey state-of-the-art detection, prevention, and mitigation techniques. In particular, we will examine static, dynamic, and hybrid analysis, black and white-box testing, unit testing, and briefly mention newer approaches that delve into deep and machine learning. Finally, because of the continued threat and relevance of cross-site scripting attacks, we list some remaining open problems to further push software security research efforts.**

***Index Terms*—Cross-site scripting XSS attacks Web application vulnerability Web security**

I. Introduction

Web applications serve as a platform for increased productivity and flexibility for both consumers and businesses. Due to the relatively low-cost (monetary and expertise) entry barrier, the adoption of web-based applications over desktop counterparts in academia, medical care, and financial industries exploded in popularity [1]. With the sudden influx of incoming data over the Internet, attackers and hackers turned their efforts to stealing potentially confidential user information. Proportionally, this called for substantially increased web security research efforts. A prominent client-facing threat is known as cross-site scripting (XSS). The Open Web Application Security Project (OWASP) newly categorizes cross-site scripting as a type of injection vulnerability, taking the third-place spot in their "Top 10" project [2], indicating its prevalence in modern web applications. OWASP categorizes injection attacks into different language contexts, such as cross-site scripting with JavaScript, SQL injection with SQL, and OS injection with shell commands. A web application is vulnerable to an injection attack if it does not validate, verify, or sanitize user input, creating malicious data which is interpreted as executable code. This data may propagate through the program and infect other segments of code and databases. Cross-site scripting attacks occur when an attacker sends malicious scripts/code to a victim which is then executed in their browser, causing harmful side-effects including but not limited to cookie and data stealing. One reason cross-site scripting is as prevalent as it is stems from two factors: the ease of creating then injecting a payload, combined with the naiveté of end users. For instance, a popular payload example of XSS comes through a HTML `script` tag. An attacker injects the trivial payload `<script>alert('Message');</script >` which, when loaded by the victim, displays an alert message in their browser. Of course, an attack of this caliber does nothing other than serve as a minor annoyance to the user, but its scalability leads to much more dangerous attacks. This extendibility directly correlates to and implies the desire for improved research detection algorithms and prevention.

Because the internet is a pipeline of constantly-receiving data, attackers can upload cross-site scripting payloads in many capacities and contexts. The scale of said attacks, as previously suggested, is broad, leading to attacks that compromise/hijack user sessions, steal cookies/usernames/passwords, violate the integrity of files, redirect victims to other websites, etc. Consequently, security researchers and web developers try to find methods of attack prevention, detection, and, when necessary, mitigation.

II. Methodology and Previous Work

We will preface our literature review on cross-site scripting attack detection and prevention with brief descriptions and definitions of common attacks, payloads, and types. In subsequent sections, we explore these definitions in greater detail as they relate to state-of-the-art methodology and research experiments.

A. Attack Types and Payloads

Liu et al. [3] alongside several other researchers [4] [5] [6] [7] [1] [8] use three distinct classifications for cross-site scripting attacks, being DOM-based, stored or non-reflected, and non-persistent or reflected. DOM-based attacks, defined by Klein in [9], occur when a document contains potentially malicious code that the client-side webpage (and JavaScript interpreter) evaluates by a "sink", e.g., `eval()` or `document.write()`. Stored or non-reflected attacks come from attacker payloads that are sent to some type of data store intended for a recipient. The victim then, unknowingly, triggers the attack by opening the doctored message causing the browser to execute the payload. Finally, reflected or non-persistent are caused by the injection of malicious code into network packet requests sent to a server, e.g., GET/POST query parameters. Because the victim originates the request to the server (injected by the attacker via, for instance, a link), their browser views and executes tainted yet presumably trustworthy return messages from the server.

We can now explain and categorize various attack payloads. Liu's paper [3] provides an overview of the following as well.

Phishing is a technique where attackers try to trick victims into providing confidential information, e.g., login usernames and passwords, without realizing the nefarious intentions. One way to exploit this via cross-site scripting is to create a dummy yet similar-looking webpage, e.g., `www.amazom.com` instead of `www.amazon.com`. Then, create a login form and field which sends a JavaScript command containing the confidential user data to both the real `amazon.com` and an attacker's server, followed by a redirect to `www.amazon.com`. Thus, to the victim, it looks as if they logged into the correct website, and unbeknownst to them, the attacker now has sensitive login data. Interestingly, with this specific example, Amazon must have the rights to both domains because typing `www.amazom.com` redirects users to the real Amazon website—a solid and secure idea on their part.

Because HTTP is a stateless protocol, browsers employ cookies to preserve data in between requests so the server communicates and remembers information about the connected client. As a result, though, cookies may contain sensitive data such as session identifiers, and other confidential information. Cookies have associated properties, e.g., name, value, lifetime, permission flags, etc. A *cookie-stealer* is a form of cross-site scripting where the payload has `document.cookie` embedded somewhere within, which when opened by the victim, executes via JavaScript and sends cookies to a listening attacker. We will discuss preventatives in section II-C.

Key loggers discretely record user keyboard input, sending the data to a log on an attacker's server (or anywhere the attacker has access). An attacker can set up a script and an event listener, e.g., JavaScript `document.onkeypress` events, that triggers/starts when the victim connects to their web page via a reflected cross-site scripting attack. From there, any time the victim types/presses a key, it is POSTed to the attacker.

Denial of service is an attack on availability, where an attacker sends a script to the victim that overloads the server with copious invalid cookie data. Because the server cannot understand such a large request, it denies access to the user. Distributed denial of service attacks may occur along a similar vein where an attacker injects a script that, when victims click, continuously sends phony (or even legitimate) requests to a server.

Now that we have provided definitions on cross-site scripting and several of its derivatives, we will review work on the detection, prevention, and mitigation of such attacks.

B. Detection Techniques

1) Static Analysis: Static analysis is a non-execution based technique of examining source code for vulnerabilities. Many years have gone into the development of quality static analysis tools with each having advantages and disadvantages compared to the competition.

In [10], Algaith et al. decided to tackle the problem of evaluating the efficacy of static analysis tools for detecting SQL injection and cross-site scripting vulnerabilities. They collected a dataset of 134 WordPress PHP based plugins where each contained at least one of the two sought-after vulnerabilities. In their tabulated results, they discuss that some tools are better than others due to their capabilities of code analysis (e.g., some only work on procedural code and do not support object-oriented programming), but ultimately, using one and only one system is not as effective as combining multiple. So, they created a majority voting paradigm in which, if m systems all discover a vulnerability out of n where $m \leq n$ and $m \geq \lceil \frac{n}{2} \rceil$, it is classified as "found". Eventually, they conclude that as they combine system detectio results, the overall sensitivity rate increases.

In [11], Melicher et al. focused on DOM-based cross-site scripting attacks. Part of this work is the evaluation of three popular static analysis tools against a test suite of DOM cross-site scripting vulnerabilities, in which they listed the advantages and disadvantages of each tool. ScanJS[1] is a now-deprecated open-source Mozilla tool for static analysis (they now encourage the use of ESLint[2]). Melicher et al. saw many false-positives since ScanJS only scans whether a point of injection could be vulnerable, and not whether it is in the provided context. Esflow[3] is another open-source JavaScript static analysis tool they examined, though we suspect it to be similarly deprecated as it has not received an update in over six years according to npm. Finally, they analyzed Burp Suite Pro[4]—an odd choice as even though it is an incredibly powerful tool, it is substantially more costly compared to the others. Their report concludes with the statement that each static analysis tool found different vulnerabilities; the two free tools (ScanJS and Esflow) consistently under-performed, while Burp Suite, unsurprisingly, performed better than their solution.

Mao et al. in [12] created a theoretical model for analyzing the behavior of JavaScript-based Android applications, namely a quintuple of: program states, input language, initial state, a transition function, and a final state set. This model is identical to a finite state automaton, both from its concept/definition to its use in practice. Their idea is to dynamically test programs and generate a (correct) behavioral model for all function calls and event listeners. From there, they construct an anomalous input sequence which, if led to an incorrect state specified by the transition function of the behavioral model, is labeled as abnormal and therefore anomalous. As a test suite, they ran their model on two Android applications where it was discovered that the malicious inputs they generated were correctly captured by the behavioral model. We, however, question its effectiveness on larger, increasingly complex, and newer Android applications. As an example, we could

[1]https://github.com/mozilla/scanjs
[2]https://eslint.org/
[3]https://github.com/skepticfx/esflow
[4]https://portswigger.net/burp

find neither RewardingYourself[5] nor PhoneGapMega[6] on the Google Play Store, perhaps a tell-tale sign of their age and relevancy, or lack thereof.

In Mohammadi et al. [13], they explored automated unit testing using language grammars for applications in Java and Java Server Pages (JSP). Attack payloads are automatically generated based on the grammars, meaning developers can adapt and reconstruct the grammars as needed for future vulnerability detection experiments.

Kronjee et al. [14] developed a tool for detecting both cross-site scripting and SQLi vulnerabilities in PHP source code. They first extract data from the National Vulnerability Database (NVD)[7] and Software Assurance Metrics And Tool Evaluation (SAMATE)[8] database for cross-site scripting and SQLi payloads. From there, using machine learning techniques, they train models based on the extracted and classified data. In the end, they discover that their methodology is more successful on SQLi compared to cross-site scripting due to the wide variety of payload types and possibilities. Even still, they test their tool, WIRECAML, against four open-source static analysis tools and determine that WIRECAML produces the best overall results (i.e., precision and recall) when examining open-source PHP projects for vulnerabilities.

2) Dynamic Analysis: Contrary to static analysis techniques and tools, dynamic analysis does not require access to the source code, implying that all derived conclusions come via program execution. As Liu et al. [3] mention, a black-box approach significantly lowers the likelihood of false positives since attacks are caught in the act rather than speculating based on source code and points of injection. At the same time, however, dynamic analysis generally comes with non-negligible overhead not seen with static analysis. Furthermore, dynamic analysis often has lower code coverage because, as stated, to catch a vulnerability, it has to execute said vulnerability—a not-so trivial task.

In Lv et al. [5], they explored adaptive random testing versus traditional fuzzing in hopes of improving the efficiency of payload and website testing.

Rathore et al. [15] proposed a machine-learning approach to automatically detecting cross-site scripting attacks in social networking services (SNS). An approach pipeline is described, where web page and social networking service features are identified from relevant web pages, extracted and filtered into a dataset, trained, then classified using ten well-known classifiers (the classification categories were "XSS" and "not-XSS"). Their training dataset included one thousand web pages. Results indicated that all classifiers have low false-positive and high true-positive rates.

In [16], Wang et al. created a framework for detecting DOM-based cross-site scripting attacks using taint tracing. In this work, they list all possible source and sink contamination functions and control points in the DOM, and rewrite said functions so that data flow analysis is trivial. Though, we question the efficacy of their results as they only test their tool, TT-XSS, against one other research project with a very limited test suite.

Very recently in [17], Lee et al. created Link: a black-box framework for detecting reflected cross-site scripting vulnerabilities via reinforcement learning. They preface their discussion by explaining the downfalls of other black-box alternatives, namely their poor scalability and adaptability of different attack payloads. They tested this approach against Wapiti[9], Burp Suite Pro, the OWASP Zed Attack Proxy[10], and Black Widow [18]: four popular and powerful web scanners in twelve applications with cross-site scripting vulnerabilities. Their results indicate that reinforcement learning with Link is the better alternative due to its high true-positive rate and incredibly low false-negative rate: an existing issue among other state-of-the-art black-box testing tools.

There were other highly-influential reports that we read on dynamic analysis, but due to their age, we omit their discussion in this report. Some research papers were also locked behind impassable paywalls, so we could not analyze their projects.

C. Prevention Techniques

In the previous section, we discussed experiments and reports on the state-of-the-art detection of cross-site scripting attacks. In this section, we will outline a few prevention methods that serve to intercept cross-site scripting attacks rather than waiting until they already exist in the system or code base [1].

Poor input validation and sanitization are the prime reasons and motivation for attackers to exploit cross-site scripting vulnerabilities. Web programmers often develop in the PHP language which comes with native database interaction functions, as well as string/input escaping procedures. To *escape* a string means to replace all potentially dangerous characters with their escaped counterparts, e.g., an opening quote ' is replaced by the escaped alternative \' to separate input data from code. Accordingly, we classify a dangerous character as one that may be mistakenly interpreted (or one that, otherwise, produces a side-effect when represented) as code.

PHP, likewise, has the function `htmlspecialchars` which converts all HTML tags and special characters from code into data equivalents, preventing the injection of, for instance, a `script` tag. `strip_tags` is a related function with the exception that it completely extracts HTML tags, leaving no data equivalents.

Another preventative is known as Content Security Policy—a server-side response header that manages and restricts content delivered to the browser (e.g., images, JavaScr
ipt, etc.). Unfortunately, a prevalent issue as noted by Melicher et al. [11] is that developers often misuse CSP due to its complexity or simply are unaware of its existence and benefits which directly results in vulnerable code.

An older example of preventative measures comes from Jim et al. [19] via the Browser-Enforced Embedded Policies, or BEEP project. In summary, they take a whitelist approach in

[5]https://apkpure.com/rewardingyourself/com.loyaltymatch.rewardingyourself
[6]https://apkpure.com/phonegap-mega/com.camden.phonegapmega
[7]https://nvd.nist.gov/
[8]https://www.nist.gov/itl/ssd/software-quality-group/samate
[9]https://wapiti-scanner.github.io/
[10]https://owasp.org/www-project-zap/

that they scrape the web page for existing script tags. They then compute its SHA-1 hash value and store it in a set of allowed hashes. Thus, whenever a script is to be executed, its SHA-1 hash is compared with the values in the whitelist, and rejected if it is not present. In addition, they developed a blacklist approach where malicious scripts/code are identified and stores in so-called "no-execute" divs or HTML blocks.

In [4], Niakanlahiji and Jafarian created WebMTD—a nonce-generating system that modifies the DOM server-side before being delivered to the client. The idea is to differentiate between existing/trustworthy executable JavaScript and malicious injected JavaScript by adding runtime nonces, or tokens, to each tag. Thus, when the page attempts to execute JavaScript, this nonce is validated, and if it is either not present or invalid, the script fails to execute, protecting the client. This idea is similar to modern protections against cross-site request forgery attacks, and it closely resembles the approach detailed in Jim et al. [19].

JSoup [20] is a Java-based HTML parsing library designed for navigating the document object-model (DOM), editing, cleansing, and validating input HTML content. While not strictly a security analysis software, due to its comprehensive tool set it is a good option to extract cross-site script vulnerabilities.

In Elhakeem et al. [21], they complement their work on the security framework ZEND with HTML Purifier: a simple and HTML standards-compliant PHP library designed to remove cross-site scripting vulnerabilities from input. Even though it is not as strenuously vetted as other academic-level projects, its practicality and ease-of-use should absolutely be considered in future evaluations of open-source cross-site script prevention tools, which we argue as a research possibility in section III.

Dabirsiaghi, later assisted by OWASP, developed Anti-Samy [22]—a Java API for removing weaknesses found in HTML/JavaScript input fields. It uses XML-based rules for whitelisting tags on a web page. Unfortunately, we feel that forcing developers to adopt and learn another standard, i.e., XML, and embed whitelisted rules is a bit cumbersome. We present a similar argument in section III.

Microsoft (ASP.NET) wrote an API designed to remove cross-site scripting attacks called AntiXss [23]. The library comes with functions to encode incoming data from the client as data rather than code, e.g., `AntiXssEncoder`. Moreover, since ASP.NET version 1.1, there is a "Request Validation" feature [24] which denies HTML data to the server that has not been encoded. Consequently, this requires developers to add the encoding functionality and substantially reduces the risk of injection attacks.

In [8], Xu et al. designed JSCSP—a JavaScript-based solution to the CSP developer misuse problem. As they note, traditional Content-Security Policy is difficult to use and configure, often misunderstood, and lacks sufficient browser support other than Chrome. Another issue is the coarse-grained policies set by CSP. For instance, there exists the "deny-all" paradigm, e.g., `default-src: self;`, even though we may wish to only block certain scripts and not all external content. One of JSCSP's selling points is its dynamically-generated policies; unlike contemporary CSP in which all policies are manually written by a developer, JSCSP generates policies based on DOM contents: scripts, HTML elements, and sensitive data. Moreover, they describe that their framework defends against common CSP workarounds, e.g., universal cross-site scripting, URL redirection, code reuse, and DNS-Prefetch attacks. Testing JSCSP demonstrates that not only is it compatible with all modern browsers that support JavaScript, it also integrates well with third-party libraries, e.g., JQuery, AngularJS, and Bootstrap. Results also indicate, however, that JSCSP has similar issues with false positives in that it sometimes blocks benign elements. Comparatively, JSCSP has potential to overtake CSP largely due to its ease of use. Oddly, though, we could not find JSCSP on any extensions/add-on services. We suspect this is due to the recent publication date of the paper, so it may not yet be publicly available.

III. Discussion and Future Work

In the previous section, we discussed research experiments and papers related to cross-site scripting detection and prevention. In this section, we outline and provide potential solutions to research questions in this area.

Our first question is concerned with reducing the burden on developers so that vulnerabilities need not to be patched but rather are prevented in the first place. As we mentioned in section II-C, Content Security Policy is an excellent method of server-side protection against not only cross-site scripting attacks but against any unwanted or untrusted content to the browser. Its issue, though, is the lack of understanding from developers who misuse it then create vulnerable web applications. Perhaps better tutorials, documentation, or a hybrid would reinforce the security benefits of correctly-implemented CSP. Secondly, there is the issue of sanitization and validation of input from the user, or more so the absence thereof. Native functions such as `htmlspecialchars` and `strip_t ags` remove all HTML functionality from user input—an excellent idea for combating cross-site scripting attacks. The associated downside, though, is again, it removes all HTML tag behavior including stylistic tags, e.g., `<b>`, `<i>` that do not inject code. For developers working on social-driven websites or services, this is likely an undesired outcome. So, they either look into better parsing and sanitization techniques or remove them altogether. The latter attempt, of course, has the obvious risks. The former, on the other hand, takes time and developer-understanding of a library which may or may not help. As aforementioned, there exist functions like `strip_tags` which also allow a whitelist of allowed tags. The problem is that a developer may not know ahead of time what tags they wish to permit, meaning that any time they encounter a tag to whitelist, they must go into the code and add said tag which brings about the risk of breaking existing code.

Our second question involves defensive programming mentioned in [7]. Is there a way we can emphasize to all developers that security on both ends of the pipeline, i.e., client and server-side is paramount to thwarting attackers? We speculate that many developers are likely aware of weaknesses in their code base, yet fixing or removing them may introduce backwards compatibility issues that break their product. On the contrary, it is possible that self-taught developers or even experienced professionals merely take source code riddled with bugs and vulnerabilities found online and stick them into their web pages. Doing so introduces a multitude of problems, largely stemming from fixing incomprehensible and non-adaptable code down the road. Writing code that assumes the worst (i.e., taking a defensive standpoint) from the beginning takes time and skill, and of course requires a clean-slate code base; a luxury that many do not have.

Our third question brings about the desire for a comparison of cross-site scripting preventatives. That is, can we effectively compare APIs, libraries, and frameworks that claim to protect against cross-site scripting attacks (and other vulnerabilities)? We know that HTML Purify[11], Microsoft's AntiXss [23], PHP's native functions[12], and similar libraries[13][14][15][16][17][18][19] aim to filter and sanitize user input. Perhaps there is a way to equally and fairly evaluate such frameworks via fuzz testing to determine if there exists a payload that breaks through the protections. This presents the obvious challenge of incorporating multiple language pipelines and writing n identical websites, where n is the number of libraries to test. It could be theoretically possible to write an automated program that creates a "template" website, then the programmer/security expert/researcher need only to insert the dependent library filtering/sanitization code.

Lastly, an increased public awareness effort would almost certainly have benefits. For example, many universities and businesses send emails en masse whenever a cyber-attack of some caliber occurs. Of course, this effort raises two questions: "How many people would participate in such training?", and "How many would still fall victim to (phishing-based) cross-site scripting attacks?" We, unfortunately, do not have ideal solutions to these problems.

IV. Conclusion

In this paper, we analyzed, explored, and defined the problem of cross-site scripting. We also examined existing prevention and detection algorithms/methods research, including static and dynamic analysis, machine learning techniques, and validation/sanitization libraries/frameworks. Finally, we discussed a few open questions in the research area with potential solutions. Cross-site scripting is a significant issue that plagues victims among the modern web. Despite great existing mitigation techniques, developers misuse or avoid them and, as a result, introduce vulnerabilities in applications much to the joy of attackers and hackers alike. While it may seem as a large price to pay at the start of development to learn and prevent cross-site scripting attacks, the long-term consequences repeatedly show that this upfront cost is worth the investment. Fixing the problems down the road, i.e., detection techniques, puts consumers/end-users at significant risk—an issue that may not exist in standards-conforming and, more importantly, competing websites.

Acknowledgements: This research paper was written for my graduate Software Security (CSC 693-02) course at the University of North Carolina Greensboro in the Spring 2022 semester.

[11]http://htmlpurifier.org/
[12]https://www.php.net/manual/en/function.htmlspecialchars.php
[13]https://github.com/voku/anti-xss
[14]https://github.com/cure53/DOMPurify
[15]https://github.com/mganss/HtmlSanitizer
[16]https://github.com/leizongmin/js-xss
[17]https://github.com/kalekarnn/xss-req-sanitizer
[18]https://github.com/rpalcolea/grails-xss-sanitizer
[19]https://github.com/Esri/arcgis-html-sanitizer

References

[1] G. Rodriguez, J. Torres, P. Flores, and E. Benavides, “Cross-Site Scripting (XSS) Attacks And Mitigation: A Survey,” *Computer Networks*, vol. 166, p. 106960, 11 2019.

[2] “OWASP Top 10,” 2021. [Online]. Available: https://owasp.org/www-project-top-ten/

[3] M. Liu, B. Zhang, W. Chen, and X. Zhang, “A Survey of Exploitation and Detection Methods of XSS Vulnerabilities,” *IEEE Access*, vol. 7, pp. 182 004–182 016, 2019.

[4] A. Niakanlahiji and J. H. Jafarian, “WebMTD: Defeating Cross-Site Scripting Attacks Using Moving Target Defense,” *Security and Communication Networks*, vol. 2019, p. 2156906, May 2019. [Online]. Available: https://doi.org/10.1155/2019/2156906

[5] C. Lv, L. Zhang, F. Zeng, and J. Zhang, “Adaptive Random Testing for XSS Vulnerability,” in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, 2019, pp. 63–69.

[6] A. P. Aliga, A. M. John-Otumu, R. E. Imhanhahimi, and A. C. Akpe, “Cross Site Scripting Attacks in Web-Based Applications,” *Journal of Advances in Science and Engineering*, vol. 1, no. 2, pp. 25–35, Sep. 2018. [Online]. Available: https://www.sciengtexopen.org/index.php/jase/article/view/19

[7] A. W. Marashdih, Z. F. Zaaba, K. Suwais, and N. A. Mohd, “Web Application Security: An Investigation on Static Analysis with other Algorithms to Detect Cross Site Scripting,” *Procedia Computer Science*, vol. 161, pp. 1173–1181, 2019, the Fifth Information Systems International Conference, 23-24 July 2019, Surabaya, Indonesia. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050919319416

[8] G. Xu, X. Xie, S. Huang, J. Zhang, L. Pan, W. Lou, and K. Liang, “Jscsp: A novel policy-based xss defense mechanism for browsers,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 862–878, 2022.

[9] A. Klein, “DOM Based Cross Site Scripting or XSS of the Third Kind - A look at an overlooked flavor of XSS,” July 2005.

[10] A. Algaith, P. Nunes, F. Jose, I. Gashi, and M. Vieira, “Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools,” in *2018 14th European Dependable Computing Conference (EDCC)*, 2018, pp. 57–64.

[11] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, “Riding Out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting,” in *Proceedings of the 25th Network and Distributed System Security Symposium*, 2018.

[12] J. Mao, J. Bian, G. Bai, R. Wang, Y. Chen, Y. Xiao, and Z. Liang, “Detecting Malicious Behaviors in JavaScript Applications,” *IEEE Access*, vol. 6, pp. 12 284–12 294, 2018.

[13] M. Mohammadi, B. Chu, and H. Richter Lipford, “Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing,” 04 2018.

[14] J. Kronjee, A. Hommersom, and H. Vranken, “Discovering software vulnerabilities using data-flow analysis and machine learning,” in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3230833.3230856

[15] S. Rathore, P. K. Sharma, and J. H. Park, *Journal of Information Processing Systems*, vol. 13, no. 4, pp. 1014–1028, 08 2017.

[16] R. Wang, G. Xu, X. Zeng, X. Li, and Z. Feng, “TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting,” *Journal of Parallel and Distributed Computing*, vol. 118, pp. 100–106, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517302186

[17] S. Lee, S. Wi, and S. Son, “Link: Black-Box Detection of Cross-Site Scripting Vulnerabilities Using Reinforcement Learning,” in *Proceedings of the ACM Web Conference 2022*, ser. WWW ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 743–754. [Online]. Available: https://doi.org/10.1145/3485447.3512234

[18] B. Eriksson, G. Pellegrino, and A. Sabelfeld, “Black widow: Blackbox data-driven web scanning,” in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1125–1142.

[19] T. Jim, N. Swamy, and M. Hicks, “Defeating script injection attacks with browser-enforced embedded policies,” in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 601–610. [Online]. Available: https://doi.org/10.1145/1242572.1242654

[20] C. Saini and V. Arora, “Information retrieval in web crawling: A survey,” in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2016, pp. 2635–2643.

[21] Y. F. G. M. Elhakeem and B. I. A. Barry, “Developing a security model to protect websites from cross-site scripting attacks using zend framework application,” in *2013 INTERNATIONAL CONFERENCE ON COMPUTING, ELECTRICAL AND ELECTRONIC ENGINEERING (ICCEEE)*, 2013, pp. 624–629.

[22] “OWASP AntiSamy,” 2021. [Online]. Available: https://owasp.org/www-project-antisamy/

[23] Microsoft, “Antixss.” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.web.security.antixss?view=netframework-4.8

[24] ——, “Request Validation - Preventing Script Attacks,” February 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/whitepapers/request-validation

[25] O. J. Falana, I. O. Ebo, C. O. Tinubu, O. A. Adejimi, and A. Ntuk, “Detection of cross-site scripting attacks using dynamic analysis and fuzzy inference system,” in *2020 International Conference in Mathematics, Computer Engineering and Computer Science (ICMCECS)*, 2020, pp. 1–6.

[26] S. Gupta and B. B. Gupta, “Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art,” *International Journal of System Assurance Engineering and Management*, vol. 8, no. 1, pp. 512–530, Jan 2017. [Online]. Available: https://doi.org/10.1007/s13198-015-0376-0