A Targeted Assessment of Cross-Site Scripting Detection Tools

Bruno Pala[1,†], Lorenzo Pisu[1,*,†], Silvia Lucia Sanna[1,†], Davide Maiorca[1,†] and Giorgio Giacinto[1,†]

[1]*University Of Cagliari, Italy*

Abstract

Cross-Site Scripting (XSS) attacks are among the most exploited vulnerabilities in web applications. As a countermeasure, various open-source XSS detectors have been released over the years, but none of such tools has been significantly tested to verify their effectiveness. In this paper, we propose an assessment of five of the most employed XSS detectors in the wild. The purpose of this analysis is two-folded: (i) to understand their efficacy in well-known and customized vulnerable environments; (ii) to provide a better comprehension of their detection mechanisms. We performed our evaluation by testing the detectors against one publicly available test bench. Additionally, we created two customized test benches that contain less trivial XSS vulnerabilities. The attained results show how, while most detectors show good accuracy at detecting trivial XSS vulnerabilities, they could fail as the XSS complexity increases.

Keywords

Web Security, Cross-Site Scripting, Exploitation

1. Introduction

Websites are becoming increasingly used in our day-to-day life, providing important services with common domains such as banking, social networking, health services, and administration. Recent reports showed that attacks on web applications are rising by an average of 22% per quarter, resulting in 4.7 million web app-related cybersecurity exploitations [1]. When developing web applications, we typically distinguish between backend and frontend. While backends are written in various languages (e.g., Python, Java, C++, Go) and executed on the host's web server, frontends are almost always developed in JavaScript and executed directly in the user's browser. This allows browsers to store sensitive information (session tokens, user preferences) in a very flexible way. However, at the same time, the use of JavaScript to read and manipulate this kind of data raises a lot of security vulnerabilities specific to clients.

Cross-Site Scripting (XSS) represents one of the most frequent attacks targeting frontends. Malicious JavaScript code is sent to the victim's browser for malicious purposes using trusted

ITASEC 2023: The Italian Conference on CyberSecurity, May 03–05, 2023, Bari, Italy

*Corresponding author.

†These authors contributed equally.

brunopala123@gmail.com (B. Pala); lorenzo.pisu@unica.it (L. Pisu); silvial.sanna@unica.it (S. L. Sanna); davide.maiorca@unica.it (D. Maiorca); giorgio.giacinto@unica.it (G. Giacinto)

0009-0005-1688-1476 (B. Pala); 0009-0001-0129-1976 (L. Pisu); 0009-0002-8269-9777 (S. L. Sanna); 0000-0003-2640-4663 (D. Maiorca); 0000-0002-5759-3017 (G. Giacinto)

websites with vulnerable functionalities. This way, attackers can exfiltrate cookies, session tokens, or sensitive information. According to the Invicti AppSec Indicator [2], 25% of web targets are vulnerable to XSS, a worrying trend considering the large variety of web-based vulnerabilities. One way to detect XSS is using black-box tools that automatically detect vulnerabilities. They can be fast, cheaper than other techniques like code review, and efficient because they provide an immediate result. However, despite the various tools available in the wild, there is no assessment of how they behave in real environments and how effective they are. In this paper, we provide a comprehensive assessment of the most popular XSS open-source detection tools in the wild. We define a testing methodology by leveraging performance metrics that allow us to evaluate the behavior of such tools. We propose our evaluation against a well-known vulnerable test bench and against a customized test bench in which XSS is implemented in a non-trivial fashion. The attained results show that, while black-box tools perform well against trivial XSS implementations such as those in the selected vulnerable web app, they tend to fail when the complexity of the vulnerability rises.

The rest of the paper is structured as follows: in Section 2, we present some background notions on XSS; Section 3 contains a brief overview of the relevant literature; in Section 4, we illustrate the proposed methodology, presenting the scanners, the test-benches used and the employed evaluation metric; in Section 5, we discuss the attained results; Section 6 contains the concluding remarks.

2. Background

As described in the Introduction, Cross-Site Scripting is the execution of JavaScript code in the victim's browser to steal specific data. Although XSS was first discussed in a CERT report in 2000, it is still one of the most common vulnerabilities in web applications, ranked respectively first and third in OWASP's top ten reports in 2017 and 2021. This vulnerability is mostly caused by developers failing to validate or sanitize the user's input, as described by the Common Weaknesses Enumeration (CWE) community [3]. Hence, untrusted data can be entered into the web app, which uses this data inside a page without proper validation. A malicious user can inject arbitrary code if the browser treats the data as HTML or JavaScript. Recent XSS research [4] categorizes the vulnerability using two articulations. The first describes where the XSS originates, namely server-side or client-side. The second describes if the payload is persistent or non-persistent. The previous two categories depend on how the malicious code is injected into the web application. However, the tools and test benches we considered in this work still use the following three categories: *(i) Reflected XSS* when the user's JavaScript input is parsed as part of a request and returned entirely or partially by a web application in the response; *(ii) Stored XSS* when the user's JavaScript input is stored on the target server, and as the victims connect to the web application, the JavaScript payload is executed; *(iii) DOM-based XSS*, similar to Reflected XSS, but in this case, the Document Object Model parses the user's JavaScript input to change the HTML page.

The best way for developers to prevent this vulnerability is to follow the best practices in coding and perform security reviews of the code along with regular auditing. However, these actions can be both money and time-consuming. Hence, automatic vulnerability scanners can

be crucial as an alternative to code review practices. In particular, these scanners are especially used by web application developers and system administrators to test web applications against vulnerabilities without accessing the source code. They mimic external attacks from malicious actors and provide cost-effective methods for detecting a wide range of vulnerabilities, provided they are selected and used correctly.

3. Related Work

Various studies have been presented in the field of web vulnerability scanners. According to the literature [5, 6], it is possible to employ specific test benches to assess scanners' performances. The experimental methodology usually consists of preparation, execution, and reporting phases. Many works in literature can be found that follow this kind of approach for testing [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20].

As described by Sarmah et al. [21], the detection approaches for XSS can be classified into three main categories: *(i) Client-Side detection*, which focuses on the browser and can be based on static [22, 23, 24, 25, 26], dynamic [27, 28, 29, 30], and hybrid analysis [31, 32, 33, 34]. *(ii) Server-Side detection*, which focuses on the web app server code and can also be based on static [35, 36, 37, 38], dynamic [39, 40, 41, 42], and hybrid analysis [43, 44].(iii) *Client-Server* detection, which controls what is sent to the client and what the server should respond [45].

Many automatic tools have been developed to detect and then mitigate the risk of XSS attacks. The two main approaches to test vulnerabilities are *white-box* and *black-box* testing. In white-box testing, the source code of the application is put under scrutiny. On the other side, black-box testing aims to simulate, without knowing the source code, how an attacker exploits the vulnerability to detect possible flaws in the target [46, 47, 48].

4. Methodology

This section defines and explains the methodology for evaluating scanners against selected test benches. This methodology can be articulated in the following steps, which will be explained further in the next subsections:

- **Scanners Selection**: we illustrate the scanners selected for the evaluation by discussing the rationale behind their choice.
- **Test Benches Selection**: the idea is to select test benches that employ various types of XSS vulnerabilities (reflected, stored, DOM-based) and increasing levels of complexity. With the term complexity, we refer to the presence of filters that complicate the exploitability of the vulnerability.
- **Definition of Metrics**: the performances of these scanners can be evaluated from multiple perspectives. Hence, it is crucial to choose metrics that can provide a clear overview of the efficacy and efficiency of the analyzed scanners.

4.1. Criteria for Scanner Selection

Specific criteria drove the choice of scanners, the most important being the availability of the source code, which is essential to fully understand the observed scanners' behavior. Moreover, we employed the following additional criteria for our selection: *(i) Popularity*, which could be established by the number of stars on GitHub (an overall evaluation of the repository considering usage, maintenance, and updates). High popularity means that the community extensively uses the scanners and can also feature a track record of good performance. *(ii) Support and updates*, typically ensured to guarantee active development and frequent improvements; *(iii) Documentation*, which is essential to make the analyst understand how the tool works and possible future improvements; *(iv) Platform and language support* so that the scanner can run on different operating systems; *(v) Easiness to use* which makes the scanner more accessible and more usable.

4.2. Scanners

We selected five open-source scanners for XSS vulnerabilities that match the criteria described above. In the following, we provide a more detailed description of these scanners, which are further summarized in Table 1. These scanners can be divided into two main groups: *active* (interaction-based) and *passive* (based on traffic monitoring). There are also tools whose scanning activity is based on fuzzing (use large amounts of payloads to discover vulnerabilities), mutation (change and adapt the payload also according to the context), and detection of specific patterns (response and payload execution analysis).

- **OWASP ZAP** [49] is the leading free scanner used in industrial environments [50], as it is available by default in the Kali Linux operating system. It is written in Java and maintained by the OWASP community, and is based on active and passive activity. Once a vulnerability is detected, the tool helps manage and prioritize risks by providing a detailed report.
- **XSStrike** [51] is a Python scanner that employs fuzzing, mutation, and Machine Learning techniques to detect vulnerabilities and evade filters. Its context analysis can also determine whether the payload was executed or filtered.
- **Dalfox** [52] is a tool written in Go that uses fuzzing, mutation, and detection based on context awareness to recognize XSS vulnerabilities. Users can choose injection points and payloads, and crawl websites to discover new endpoints and potentially vulnerable pages.
- **XSSer** [53] is a Python scanner using multiple payloads for detection by mutating them. It has an interesting feature called *exploitation*, through which the detected vulnerability can also be exploited.
- **Wapiti** [54] is a Python tool using active and passive scanning with a wide range of attack payloads, as well as a flexible and customizable scan engine supporting different analyses according to specific needs.

Scanner	GitHub Stars	Platform	Version	Type of XSS
OWASP ZAP	10349	Java	2.12.0	R,S,D
XSStrike	11081	Python	3.1.5	R,S
Dalfox	2264	Go	2.8.2	R,S,D
XSSer	854	Python	1.8.4	R,S,D
Wapiti	508	Python	3.1.3	R,S

Table 1
A comprehensive summary of the employed scanners. In the last column (Type of XSS), we denoted the three different types of XSS by using their first letter: hence R stands for Reflected, S for Stored, and D for DOM-based

4.3. Selection of the Test Benches

Test benches are specialized systems or setups to test and evaluate various software, systems, and devices before releasing them. Choosing such benchmarks properly is crucial to ensure a fair analysis of web scanners. We analyzed various benchmarks and evaluated them according to the number of citations in scientific papers, source code availability, updates, and identification of all XSS types. As a result, we selected the Damn Vulnerable Web Application (DVWA) as the primary benchmark of this work. Other benchmarks, such as OWASP benchmark [55] and WAVSEP [56], widely used in previous works, are no more updated (since 2016 and 2014, respectively), and the given documentation is insufficient.

DVWA is a widely used and well-established platform that has been cited in previous research studies [15, 19]. It is an open-source PHP web application that is designed to be vulnerable to various attacks (including XSS). It runs a variety of platforms and environments with a user-friendly web interface and a MySQL built-in database for easy configuration and management. Moreover, it features a range of security levels spanning from no security measures (low) to no vulnerabilities (impossible) to tailor the test to different scenarios and test the scanner's limitations. To ensure the code's isolation, portability, reproducibility, and immutability, we have executed DVWA via Docker.

However, the XSS vulnerabilities embedded in DVWA are widely known and can be considered trivial (even for high-difficulty settings). Hence, we developed two custom test benches to mimic some scenarios where the complexity of the vulnerability increases. We decided to use standard PHP functions that automatically escape special characters. The reason behind this is to understand how the tools' detection mechanisms work when the payload they send is modified or escaped by the server. Hosting a custom test bench can provide several benefits when evaluating the effectiveness of automatic web scanners. In particular, it allows a complete environmental *control* to ensure consistency and repeatability of the tests. Moreover, we can employ *real-world* vulnerabilities to evaluate the scanner's performance in a tangible environment. Finally, it is crucial to ensure that the selected scanners were not overfitted to work only with standard test benches like DVWA. The two test benches, both vulnerable to reflected XSS, can be summarized as follows:

- **Low Security**. The first test bench is based on the `htmlspecialchars` PHP function, as shown in Listing 1. We used it to filter a user input appended to the `src` attribute of an `img` tag. The function converts special characters that can be interpreted as HTML code

to the relative HTML entity. Using the function properly can help prevent XSS attacks by ensuring that user-supplied input is properly encoded and escaped. However, this function could be bypassed. For example, before PHP 8.2 (released at the beginning of 2023 [57]), the function did not filter single quotes, which can be used in combination with other attributes (i.e. `onerror`) to execute malicious JavaScript code. To sum up, this function can be very useful in certain scenarios. For example, suppose the user input was reflected directly outside a tag attribute. In that case, an attacker can send a payload like `<script>alert(1)</script>`, but it would be escaped by the function and replaced with `&lt;script&gt;alert(1)&lt;/script&gt;` which wouldn't be interpreted as HTML. However, it is useless in the context of our test bench. In fact, we can exploit the XSS by sending a payload such as `nonexistent' onerror='alert(1)`.

```
$src = htmlspecialchars($_GET['src']);
$image = "<img src='".$src."' alt='Error' errorwidth=30% height=30%>";
echo $image;
```

Listing 1: An example of vulnerability involving `htmlspecialchars`

- **High Security**. The second test bench (shown in Listing 2) combines the `htmlspecialchars` with the `strtoupper()` PHP function. As the name suggests, this function converts a string to uppercase. Again, the vulnerability arises from `htmlspecialchars`, but the supplied JavaScript code will not run because of the uppercase conversion, preventing it from execution. Still, there is a way to bypass these restrictions: using an esoteric language like `JSFuck`, a programming language designed to test JavaScript boundaries where code is written with only six characters and can be run in any web browser or engine interpreting JavaScript. In this way, the attacker can bypass the uppercase transformation.

```
$src = strtoupper(htmlspecialchars($_GET['src']));
$image = "<img src='".$src."' alt='Error' errorwidth=30% height=30%>";
echo $image;
```

Listing 2: An example of vulnerability involving the `htmlspecialchars` and `strtoupper` functions

4.4. Evaluation Metrics

The easiest way to determine how a scanner performs is to look at how the Cross-Site Scripting vulnerability is detected and exploited, leading to True Positives (TP) or False Positives (FP). A True Positive is a correctly identified instance of an XSS, meaning that the scanner identified the vulnerability, confirmed with manual validation. A False Positive is when the scanner identifies a potential XSS in the target but the generated payload does not work. False positives can occur for various reasons, such as a lack of understanding of the application logic by the scanner, outdated payloads, or complex code that is difficult for the scanner to parse correctly.

Another metric to evaluate the scanner's efficiency is the number of requests sent to the tested web application. Each request injects various payloads into different application parameters to detect XSS vulnerabilities. A scanner that makes fewer requests to achieve the same level of coverage is considered more efficient because it places less load on the web application and

is less likely to disrupt its normal operation. On the other hand, a scanner that generates too many requests might trigger security mechanisms such as rate limiters, making the scanning process difficult or even impossible to complete.

5. Results

This section analyzes and evaluates the five automated web scanners described in Section 4.2 against the Damn Vulnerable Web Application (DVWA) and the custom test benches. Regarding the DVWA test bench, all the scanners were tested against every type of XSS vulnerability (Reflected, Stored, and DOM-based) at low, medium, and high levels of security. On the other hand, the custom test benches feature two cases of reflected XSS, as described in Section 4.3.

We manually tested the payloads generated by the scanners on two popular web browsers, Google Chrome (version 109.0.5414.74), and Firefox (version 108.0.2). As indicated in Section 4.4, the presence of a working payload is considered as a true positive. Since every test bench is vulnerable to XSS, we used the false positive case when the tool correctly detected an XSS, but the payload did not work. Notably, each tool was set up with the configuration parameters that, to the best of our knowledge, provided the best results possible.

5.1. Assessment against Test Benches

In the following, we first discuss the attained results for each scanner against the DVWA testbench, also highlighted in Figure 2 (left). Then, we provide an additional discussion on the results attained against the custom test benches.

- **OWASP ZAP** must be used from the browser web page. The attack must be launched from the Active Scan tab. If it detects an XSS, the tool provides the malicious payload. It has a success rate of 100% in detecting reflected and stored XSS, generating payloads without user interaction. The tool also captures all the DOM-based XSS vulnerabilities using a specific module called *DOM XSS Active Scan Rule* [58].
- **XSStrike** generates a different working payload reflected in the response body and asks the attacker if it is enough or if it should continue generating. It has a success rate of 100% for reflected and stored XSS, but with payloads requiring user interaction. This last point limits the scope and the impact of the attack as it can infect only users interacting with it. The tool is not able to generate any DOM-based XSS payload but only highlights potential JavaScript functions that can be used to inject malicious code.
- **Dalfox** generates different context-based payloads, marking those that are correctly reflected and working. It has a success rate of 100% in detecting reflected and stored XSS with payloads that do not require user interaction. However, it has difficulties in detecting DOM-Based XSS, and this defect was also confirmed by the author with an open issue in its GitHub page [59].
- **XSSer** tries different payloads from its database and reports those working correctly. However, it generates two false positives, as shown in Appendix B. This may significantly impact the detection of vulnerabilities, as the user needs to manually check the generated payloads, leading to a waste of time. Despite this, it can catch all of the DOM-based XSS.

- **Wapiti** provides a minimal report with a single payload working on the analyzed page. It had a success rate of 100% for reflected and stored XSS, generating payloads without user interaction. However, the tool does not support the detection of DOM-based XSS.

Overall, all the examined tools can correctly detect reflected XSS on DVWA at low and medium security levels. Most tools correctly identify stored XSS, and only two have failed to detect it on the high-security level of DVWA. DOM-based XSS is more difficult to detect because the page's JavaScript code needs to be executed.

On the right part of Figure 2, we show the performances achieved by the tools against our custom test benches. Only `Wapiti` is able to generate a working payload against the *Low Security* custom test bench. A clearer view of the behavior of the examined tools is provided in Table 2, which reports the payloads generated by each tool.

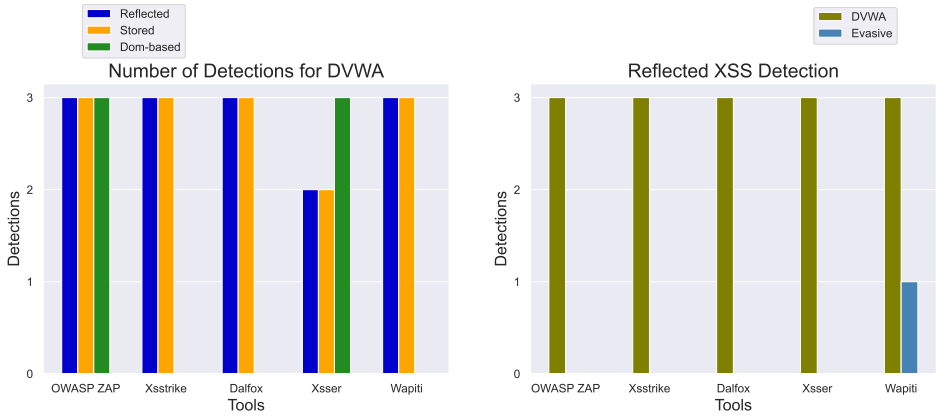

Figure 1: On the left, the histogram shows the number of detections (out of 3) for each tool on the DVWA. On the right, the histogram shows the number of detections (out of 3 for DVWA and out of 2 for the evasive test bench) for each tool.

Tool	Payload	Result
OWASP ZAP	javascript:alert(1);	FP
XSStrike	'AutofocUs onfocUS='(confirm)()	FP
Dalfox	'onload=prompt.apply(null,1) class=dalfox	FP
XSSer		FN
Wapiti	'onload='alert(/w024qjvkbf/)'onerror='alert(/w024qjvkbf/)	TP

Table 2: Payloads generated by each tool against the *Low Security* custom test bench. TP represents True Positives (the XSS is detected, and the payload is correct), FP represents False Positive (the XSS is detected, but the payload does not work), FN represents False Negatives (the vulnerability is not detected).

OWASP ZAP, XSStrike, and Dalfox are able to detect the XSS vulnerability but not to properly exploit it, generating syntactically correct payloads that are not executed. XSSer completely

fails at detecting the vulnerability, and Wapiti is the only scanner that is able to generate a working payload. The reasons why the payloads were not executed can be various. Since XSS is a client-side vulnerability, one of the reasons is that *the evolution of browsers* highly impacts whether a payload will work, thus filtering too simple ones. For example, ZAP's payload uses `javascript:`, which will not work on modern versions of Chrome and Firefox because an `img` tag cannot load anything that is not of image type.

Another reason is *the HTML context* in which the payload is injected. XSStrike uses the *autofocus+onfocus* trick to automatically focus a certain tag on the page when loaded, triggering the payload execution automatically. However, this trick will not work on `img` tags because they are not focusable.

Finally, none of the tools could even detect the vulnerability embedded in the High Security test bench. This clearly shows that, when the sophistication of the vulnerability increases, automatic scanners dramatically drop their detection capabilities. In particular, the analyzed scanners heavily rely on searching a reflection of the payloads they send inside the server's response. Our evasive test bench performs changes in the payload (HTML characters filtering and uppercase transformation), making the reflection harder to detect. More detailed results for each tool can be found in Appendix B.

5.2. Detection Performances

In this section, we provide a more detailed overview of the performances of the examined scanners. First, we show a chart comparing the number of requests made by each analyzed tool. As said in Section 4.4, this is an important metric that can underline the capacity of a tool to generate context-specific payloads or just brute-force a list of payloads against the web application. Figure 2 provides an overview of the scanners' performances in terms of the number of requests and standard deviation, calculated over all test benches (DVWA and custom) and security levels. We considered only the true positives for each tool to make this calculation.

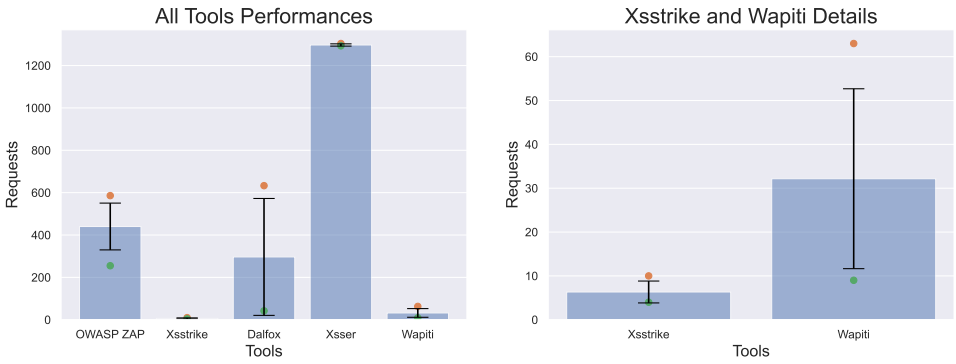

Figure 2: Average, standard deviation, minimum and maximum requests performed by each tool. On the left box plot all the tools are compared, and on the right a detail of XSStrike and Wapiti.

The Figure reports some interesting details. For example, XSSer tries all possible payloads in its list before providing the final result. On the contrary, XSStrike and Wapiti tend to generate only a few payloads to reach the goal. However, the great performances of XSStrike are also

related to the need for interaction by the user, who always has to confirm whether another payload should be generated or not.

Overall, a scanner that can detect XSS with fewer requests is better because it puts less stress on the website. However, a tool that can detect all kinds of XSS while making a very low number of requests has not emerged from this analysis. Therefore, it becomes crucial to find a trade-off between detection performances and efficiency.

Table 3 provides a general overview of the type of working payload generated by each tool during their detection (and exploitation) attempts against the low-security setting of the DVWA (reflected XSS). In particular, we can identify two main categories of payloads generated by the tools. The first category uses the *script tags* to execute JavaScript: this standard way works in the most basic examples of XSS. OWASP ZAP, XSSer, and Wapiti belong to this category. However, we can still notice some differences, such as the capitalization of the word `script` to evade WAFs (Web Application Firewalls) or using different arguments for the alert function.

The second category contains the *non-script tags* payloads used by XSStrike and Dalfox. In this case, instead of script tags, we can use any other tag with attributes that can be exploited to execute JavaScript code. For example, XSStike uses the `onmouseover` attribute, which triggers the execution of the payload when the user hovers on the tag with the mouse (in the case of the tag `html`, this is triggered whenever the user moves the mouse on the page). Dalfox uses a payload containing the `onload` attribute, which triggers the execution of JavaScript code when the tag loads. There are countless tags and attributes that can be exploited in this way.

Tool	Payload
OWASP ZAP	</pre><scrIpt>alert(1);</scRipt><pre>
XSStrike	<HtmL/+/onMOuseOVEr%09=%09[8].find(confirm)//
Dalfox	<sVg/onload=alert.bind()(1) class=dalfox>
XSSer	"><SCRIPT>alert('<random digits>')</SCRIPT>
Wapiti	<ScRiPt>alert('w4a815881u')</sCrIpT>

Table 3: Comparison between the different payloads generated by the tools for reflected XSS in low-security settings (DVWA).

6. Conclusions and Future Work

In this paper, we proposed an assessment of five popular XSS open-source scanners against various test benches. The attained results show that, while such scanners are very good at detecting (and in many cases, exploiting) XSS vulnerabilities, their accuracy dramatically decreases when the complexity of such vulnerabilities increases.

We are also aware that this study features several limitations, with the presented results being still preliminary. Despite the amount of diligence and study put into the configuration of the scanners, there may exist the possibility that some results can be affected by misconfiguration or lack of parameters in the commands used to launch the scanners. Moreover, the number of examined test benches can be significantly increased to evaluate the scanners further, and we plan to analyze more real-world scenarios in future work.

Nevertheless, we think we succeeded in highlighting some problems with popular XSS scanners that need to be addressed, thus opening intriguing lines of research toward a more accurate detection of XSS attacks. In this regard, a possible direction for future work is to update the scanners' code to improve their payload generation and detection mechanisms (for example, DOM-based XSS). This becomes especially important as new vulnerabilities like Prototype Pollution and Client-Side Template Injection (CSTI) are providing new ways to obtain XSS by abusing vulnerable JavaScript code.

Acknowledgement

This work has been carried out while Silvia Lucia Sanna was enrolled in the Italian National Doctorate on Artificial Intelligence run by Sapienza University of Rome in collaboration with University of Cagliari.

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

References

[1] Web app attacks grew 251% in two years. https://www.lifars.com/2022/01/web-app-attacks-grew-251-in-two-years/, 2022.

[2] Web application vulnerability report. https://www.acunetix.com/white-papers/acunetix-web-application-vulnerability-report-2019/, 2019.

[3] Cwe-79: Improper neutralization of input during web page generation ('cross-site scripting'). https://cwe.mitre.org/data/definitions/79.html.

[4] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. 2019.

[5] Nuno Antunes and Marco Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *2009 15th IEEE pacific rim international symposium on dependable computing*, pages 301–306. IEEE, 2009.

[6] Nuno Antunes and Marco Vieira. Benchmarking vulnerability detection tools for web services. In *2010 IEEE International Conference on Web Services*, pages 203–210. IEEE, 2010.

[7] SE Idrissi, N Berbiche, F Guerouate, and M Shibi. Performance evaluation of web application security scanners for prevention and protection against vulnerabilities. *International Journal of Applied Engineering Research*, 12(21):11068–11076, 2017.

[8] Azwar Al Anhar and Yohan Suryanto. Evaluation of web application vulnerability scanner for modern web application. In *2021 International Conference on Artificial Intelligence and Computer Science Technology (ICAICST)*, pages 200–204. IEEE, 2021.

[9] Malik Qasaimeh, A Shamlawi, and Tariq Khairallah. Black box evaluation of web application scanners: Standards mapping approach. *Journal of Theoretical and Applied Information Technology*, 96(14):4584–4596, 2018.

[10] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated

black-box web application vulnerability testing. In *2010 IEEE symposium on security and privacy*, pages 332–345. IEEE, 2010.

[11] Alexandre Miguel Ferreira and Harald Kleppe. Effectiveness of automated application penetration testing tools, 2011.

[12] Fakhreldeen Abbas Saeed and E ABED ELGABAR. Assessment of open source web application security scanners. *Journal of Theoretical and Applied Information Technology*, 61(2):281–287, 2014.

[13] Larry Suto. Analyzing the accuracy and time costs of web application security scanners. *San Francisco, February*, 2010.

[14] Shafi Alassmi, Pavol Zavarsky, Dale Lindskog, Ron Ruhl, Ahmed Alasiri, and Muteb Alzaidi. An analysis of the effectiveness of black-box web application scanners in detection of stored xssi vulnerabilities. *International Journal of Information Technology and Computer Science*, 4(1), 2012.

[15] Yuma Makino and Vitaly Klyuev. Evaluation of web vulnerability scanners. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 1, pages 399–402. IEEE, 2015.

[16] Mansour Alsaleh, Noura Alomar, Monirah Alshreef, Abdulrahman Alarifi, and AbdulMalik Al-Salman. Performance-based comparative assessment of open source web vulnerability scanners. *Security and Communication Networks*, 2017, 2017.

[17] Balume Mburano and Weisheng Si. Evaluation of web vulnerability scanners based on owasp benchmark. In *2018 26th International Conference on Systems Engineering (ICSEng)*, pages 1–6. IEEE, 2018.

[18] PA Sarpong, LS Larbi, DP Paa, IB Abdulai, R Amankwah, and A Amponsah. Performance evaluation of open source web application vulnerability scanners based on owasp benchmark. *Int. J. Comput. Appl.*, 2021.

[19] Richard Amankwah, Jinfu Chen, Patrick Kwaku Kudjo, and Dave Towey. An empirical comparison of commercial and open-source web vulnerability scanners. *Software: Practice and Experience*, 50(9):1842–1857, 2020.

[20] Natasa Suteva, Dragi Zlatkovski, and Aleksandra Mileva. Evaluation and testing of several free/open source web vulnerability scanners. 2013.

[21] Upasana Sarmah, DK Bhattacharyya, and Jugal K Kalita. A survey of detection methods for xss attacks. *Journal of Network and Computer Applications*, 118:113–143, 2018.

[22] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004.

[23] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, pages 91–100, 2010.

[24] Riccardo Pelizzi and R Sekar. Protection, usability and improvements in reflected xss filters. In *proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 5–5, 2012.

[25] Kanpata Sudhakara Rao, Naman Jain, Nikhil Limaje, Abhilash Gupta, Mridul Jain, and Bernard Menezes. Two for the price of one: A combined browser defense against xss and clickjacking. In *2016 International Conference on Computing, Networking and Communications (ICNC)*, pages 1–6. IEEE, 2016.

[26] Chih-Hung Wang and Yi-Shauin Zhou. A new cross-site scripting detection mechanism integrated with html5 and cors properties by using browser extensions. In *2016 International Computer Symposium (ICS)*, pages 264–269. IEEE, 2016.

[27] Oystein Hallaraker and Giovanni Vigna. Detecting malicious javascript code in mozilla. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 85–94. IEEE, 2005.

[28] Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. {ZigZag}: Automatically hardening web applications against client-side validation vulnerabilities. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 737–752, 2015.

[29] Jinkun Pan and Xiaoguang Mao. Domxssmicro: A micro benchmark for evaluating dom-based cross-site scripting detection. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 208–215. IEEE, 2016.

[30] Dimitris Mitropoulos, Konstantinos Stroggylos, Diomidis Spinellis, and Angelos D Keromytis. How to train your browser: Preventing xss attacks using contextual script fingerprints. *ACM Transactions on Privacy and Security (TOPS)*, 19(1):1–31, 2016.

[31] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, volume 2007, page 12, 2007.

[32] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[33] Jinkun Pan and Xiaoguang Mao. Detecting dom-sourced cross-site scripting in browser extensions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 24–34. IEEE, 2017.

[34] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. {ZOZZLE}: Fast and precise {In-Browser}{JavaScript} malware detection. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[35] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[36] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

[37] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, volume 15, pages 179–192, 2006.

[38] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference*, pages 295–307. Springer, 2005.

[39] Martin Johns, Björn Engelmann, and Joachim Posegga. Xssds: Server-side detection of cross-site scripting attacks. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 335–344. IEEE, 2008.

[40] Peter Wurzinger, Christian Platzer, Christian Ludl, Engin Kirda, and Christopher Kruegel. Swap: Mitigating xss attacks using a reverse proxy. In *2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 33–39. IEEE, 2009.

[41] Swati Maurya. Positive security model based server-side solution for prevention of cross-

site scripting attacks. In *2015 Annual IEEE India Conference (INDICON)*, pages 1–5. IEEE, 2015.

[42] Xiaobing Guo, Shuyuan Jin, and Yaxing Zhang. Xss vulnerability detection using optimized attack vector repertory. In *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 29–36. IEEE, 2015.

[43] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52, 2004.

[44] Giuseppe A Di Lucca, Anna Rita Fasolino, M Mastoianni, and Porfirio Tramontana. Identifying cross site scripting vulnerabilities in web applications. In *Proceedings. Sixth IEEE International Workshop on Web Site Evolution*, pages 71–80. IEEE, 2004.

[45] Biswajit Panja, Tyler Gennarelli, and Priyanka Meharia. Handling cross site scripting attacks using cache check to reduce webpage rendering time with elimination of sanitization and filtering in light weight mobile web browser. In *2015 First Conference on Mobile and Secure Services (MOBISECSERV)*, pages 1–7. IEEE, 2015.

[46] Jose Fonseca, Marco Vieira, and Henrique Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *13th Pacific Rim international symposium on dependable computing (PRDC 2007)*, pages 365–372. IEEE, 2007.

[47] Lawrence A Gordon, Martin P Loeb, William Lucyshyn, and Robert Richardson. 2005 csi/fbi computer crime and security survey. *Computer Security Journal*, 21(3):1, 2005.

[48] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131. Springer, 2010.

[49] Owasp zap - github. https://github.com/zaproxy/zaproxy.

[50] Is zap the world's most popular web scanner? https://www.zaproxy.org/blog/2020-04-02-is-zap-the-worlds-most-popular-web-scanner/.

[51] Xsstrike - github. https://github.com/s0md3v/XSStrike.

[52] Dalfox - github. https://github.com/hahwul/dalfox.

[53] Xsser - github. https://github.com/epsylon/xsser.

[54] Wapiti - github. https://github.com/wapiti-scanner/wapiti.

[55] Owasp benchmark. https://owasp.org/www-project-benchmark/.

[56] Wavsep. https://github.com/sectooladdict/wavsep.

[57] Php 8.2 release. https://www.php.net/ChangeLog-8.php#PHP_8_2, 2023.

[58] Dom xss active scan rule. https://www.zaproxy.org/docs/desktop/addons/dom-xss-active-scan-rule/.

[59] Dalfox fails to detect dom xss - github issue. https://github.com/hahwul/dalfox/issues/412.

Appendices

A. Commands used to run the tests

For each tool used from a CLI we listed the used commands.

XSStrike:

```
python3 xsstrike.py -u http://192.168.1.14:80/vulnerabilities/xss_r/?name=XSS --skip-dom --
    headers <headers>
python3 xsstrike.py -u "http://192.168.1.14:80/vulnerabilities/xss_s/" --data "txtName=&
    mtxMessage=&btnSign=Sign+Guestbook" --skip-dom --headers <headers>
```

Dalfox:

```
./dalfox url http://192.168.1.14:80/vulnerabilities/xss_r/?name= -H "Cookie: security=<
    security_level>; PHPSESSID=<phpsessionid>"
./dalfox file raw_data.txt --rawdata --skip-mining-all --http
./dalfox url http://192.168.1.14:80/vulnerabilities/xss_d/?default= -H "Cookie: security=<
    security_level>; PHPSESSID=<phpsessionid>" --deep-domxss --mining-dom
```

XSSer:

```
./xsser -u 'http://192.168.1.14:80/vulnerabilities/xss_r/?name=XSS' --cookie="PHPSESSID=<
    phpsessionid>; security=<security_level>" --auto -s
./xsser -u 'http://192.168.1.14/vulnerabilities/xss_s/' -p "txtName=XSS&mtxMessage=XSS&
    btnSign=Sign+Guestbook" --cookie="PHPSESSID=<phpsessionid>; security=<security_level>"
     --auto -s
./xsser -u 'http://192.168.1.14:8001/vulnerabilities/xss_d/' -g "?default=XSS" --cookie="
    PHPSESSID=<phpsessionid>; security=<security_level>" --Dom --auto -s
```

Wapiti:

```
wapiti -u 'http://192.168.1.14/vulnerabilities/xss_r/?name=' -H 'Cookie: PHPSESSID=<
    phpsessionid>; security=<security_level>' -m xss
wapiti -u 'http://192.168.1.14/vulnerabilities/xss_s/?txtName=f&mtxMessage=f&btnSign=Sign+
    Guestbook' --skip "btnSign" -H 'Cookie: PHPSESSID=<phpsessionid>; security=<
    security_level>' -m xss,permanentxss:post
```

B. Detailed Results

Test bench	T	S	R	U	N	Payload
DVWA	R	L	TP	no	374	</pre><scrlpt>alert(1);</scRipt><pre>
		M	TP	no	374	</pre><scrlpt>alert(1);</scRipt><pre>
		H	TP	no	255	</pre><img src=x onerror=prompt()><pre>
	S	L	TP	no	581	<script>alert(1);</script>
		M	TP	no	586	<img src=x onerror=alert(1);>
		H	TP	no	586	<img src=x onerror=alert(1);>
	D	L	TP	no	402	#<script>alert(5397)</script>
		M	TP	no	402	#<script>alert(5397)</script>
		H	TP	no	402	#<script>alert(5397)</script>
Evasive	R	L	FP	no	315	javascript:alert(1);
		H	FN			

Table 4
OWASP ZAP Results

Test bench	T	S	R	U	N	Payload
DVWA	R	L	TP	yes	4	<HtmL/+/onMOuseOVEr%09=%09[8].find(confirm)//
		M	TP	yes	4	<html/+/ONMOUSEovEr+=+[8].find(confirm)%0dx//
		H	TP	yes	4	<hTML%0doNpoinTeRenTER%09=%09confirm()%0dx//
	S	L	TP	yes	10	<dETaiLs%0doNTOGgle%0a=%0aa=prompt,a()//
		M	TP	yes	9	<HTmL%09ONpoiNtERENteR%0a=%0a[8].find(confirm)//
		H	TP	yes	7	<hTML%0dONpOinteREnTeR%0d=%0dconfirm()//
Evasive	R	L	FP	no	29	'AutofocUs onfocUS='(confirm)()
		H	FN			

Table 5
XSStrike Results

Test bench	T	S	R	U	N	Payload
DVWA	R	L	TP	no	633	<sVg/onload=alert.bind()(1) class=dalfox>
		M	TP	no	633	<sVg/onload=alert.bind()(1) class=dalfox>
		H	TP	no	633	<sVg/onload=prompt(1)>iting headless
	S	L	TP	no	42	"><Svg/onload=alert(1) class=dalfox>
		M	TP	no	55	<xmp><p title="</xmp><svg/onload=print(1)>
		H	TP	no	120	<xmp><p title="</xmp><svg/onload=alert(1)>
	D	L	FN			
		M	FN			
		H	FN			
Evasive	R	L	FP	no	2284	'onload=prompt.apply(null,1) class=dalfox
		H	FN			

Table 6
Dalfox Results

Test bench	T	S	R	U	N	Payload
DVWA	R	L	TP	no	1293	"><SCRIPT>alert('<random digits>')</SCRIPT>
		M	TP	no	1293	"><SCRIPT>alert('<random digits>')</SCRIPT>
		H	FP	no	1293	<--<img/src= onerror=<random digits>>--!>
	S	L	TP	no	1293	"><SCRIPT>alert('<random digits>')</SCRIPT>
		M	TP	no	1293	"><SCRIPT>alert('<random digits>')</SCRIPT>
		H	FP	no	1293	<BODY BACKGROUND="<random digits>">
	D	L	TP	no	1304	Y#<script>alert('<random digits'>)</script>
		M	TP	no	1304	Y#<script>alert('<random digits'>)</script>
		H	TP	no	1304	Y#<script>alert('<random digits'>)</script>
Evasive	R	L	FN			
		H	FN			

Table 7
XSSer Results

Test bench	T	S	R	U	N	Payload
DVWA	R	L	TP	no	15	<ScRiPt>alert('w4a815881u')</sCrIpT>
		M	TP	no	31	<ScRiPt>alert('wl1rewj2d3')</sCrIpT>
		H	TP	no	63	<SvG/oNloAd=alert(/wvh1utxups/)>
	S	L	TP	no	9	<ScRiPt>alert('wkd6pjqju7')</sCrIpT>
		M	TP	no	19	<ScRiPt>alert('w7hazq4o4x')</sCrIpT>
		H	TP	no	79	<SvG/oNloAd=alert(/w64b26uhsj/)>
Evasive	R	L	TP	no	56	'onload='alert(/w024qjvkbf/)'onerror='alert(/w024qjvkbf/)
		H	FN			

Table 8
Wapiti Results