Assessment of Dynamic Open-source Cross-site Scripting Filters for Web Application

Nurul Atiqah Abu Talib[1], and Kyung-Goo Doh[2*]
[1] Computer Science and Engineering, Hanyang University ERICA
Gyeonggi-do, South Korea
[e-mail: atiqah@hanyang.ac.kr]
[2] Computer Science and Engineering, Hanyang University ERICA
Gyeonggi-do, South Korea
[e-mail: doh@hanyang.ac.kr]
[*]Corresponding author: Kyung-Goo Doh

Received May 11, 2020; revised August 27, 2020; revised March 23, 2021; accepted September 1, 2021; published October 31, 2021

Abstract

This study investigates open-source dynamic XSS filters used as security devices in web applications to account for the effectiveness of filters in protecting against XSS attacks. The experiment involves twelve representative filters, which are examined individually by placing them into the final output function of a custom-built single-input-form web application. To assess the effectiveness of the filters in their tasks of sanitizing XSS payloads and in preserving benign payloads, a black-box testing method is applied using an automated XSS testing framework. The result in working with malicious and benign payloads shows an important trade-off in the filters' tasks. Because the filters that only check for dangerous or safe elements, they seem to neglect to validate their values. As some safe values are mistreated as dangerous elements, their benign payload function is lost in the way. For the filters to be more effective, it is suggested that they should be able to validate the respective values of malicious and benign payloads; thus, minimizing the trade-off. This particular assessment of XSS filters provides important insight regarding the filters that can be used to mitigate threats, including the possible configurations to improve them in handling both malicious and benign payloads.

Keywords: Cross-site scripting, filters, open-source, web application, security, assessment.

1. Introduction

Cross-site scripting (XSS) is one of the most prominent web attacks and continues to plague web applications, appearing in the OWASP Top 10 Most Critical Web Application Security Risks for over a decade [1][2]. It is a type of attack that targets client-side scripts to cause unintended execution of scripts in the client's web browser.

Recent studies have shown that the number of web applications potentially vulnerable to XSS attacks has increased significantly between the years 2016 and 2018 [3]. The Hacker News is still reporting incidents of rampant online website attacks [4]. Web developers sometimes experience time crunches when they are working, which can cause them to have less regard for secure coding practices, thus risking their applications to XSS. They would seem to rely on already available XSS filters for internet security. However, this raises the question of whether these filters are sufficiently reliable to perform their expected task. It is this concern that creates our strong motivation to carry out a more scrupulous examination of their operation in internet systems in a manner that would enable us (and web developers) to detect the reasons for these failures.

A good filter is the one that would not only tackle harmful inputs efficiently but also handle benign inputs correctly. That is to say, some applications allow users to customize a web page's content, such as by defining custom (yet legitimate) HTML. If these inputs were to be filtered out, the application then no longer serves its intended purpose.

This study examines currently available dynamic open-source filters commonly used as security devices in web applications. We assess their performance in defending against XSS attacks and in handling benign payloads. Our main concern is to appraise common filtering techniques used by the filters to see if they sustain correct sanitized payloads. We also seek to underline their benefits based on the filtering ability. By extension, this study aims to assist web developers (particularly non-experts who require additional security input) with their security needs, and to help them consolidate security measures in their applications.

Specifically, this study systematically analyzes 12 readily available open-source XSS filters. Our overall objective in this paper is to (1) review the sanitization ability of each filter according to their success or failure, (2) investigate the extent of the filter's ability to handle benign payloads so that it does not destroy the payloads' intended meaning in any way, (3) make improvements on the filters' sanitization capabilities within certain conditions, and (4) survey each of the filtering techniques used and their ability in handling payloads. With the view that a properly detailed assessment on the capabilities is still forthcoming, we initiate this experiment to examine the benefits of web application open-source XSS filters as a security device.

In order to answer these questions, we test the 12 filters, some of which are recommended by the InfoSec Institute [5]. We create a simple vulnerable web application with a single input form and place each of the 12 filters (one at a time) in the final output function of the application. The final output function is where the HTML string is supplied to the browser. We assess the ability of filters to prevent XSS attacks by testing them against commonly reported attack payloads. We identify successful and unsuccessful attacks based on the execution of scripts on the browser.

This report comprises seven sections. Having stated our research problems as an introduction in Section 1, we next proceed to Section 2 and provide basic descriptions of the various forms of XSS attacks and vulnerable applications with examples. In Section 3, we explain the procedures and processes of our experiments. Section 4 presents our findings and results. A summary of our findings is presented in Section 5 and our discussions of related works are given in Section 6. Section 7 provides our final conclusions and possible directions for future works.

2. Preliminaries

This section provides an overview of XSS attacks, our description of vulnerability contexts in web applications, and a basic explanation of open-source XSS filters.

2.1 Types of Cross-site Scripting

XSS is a code injection attack, commonly associated with HTML injection. XSS can be described as an attacker injecting malicious scripts into web applications, thereby causing unintended execution by a client's web browser. This exploitation is possible due to poor filtering and sanitization routines for user inputs. The consequences of XSS attacks include information theft, session hijacking, web defacement, and loss of system integrity. Generally, XSS attacks are divided into three different types: reflected, stored, and DOM-based.

Reflected (or non-persistent) XSS attacks occur when malicious payloads are sent back to the client without any modification in the output of the web-page response, hence their name. Stored (or persistent) XSS attacks occur when user inputs containing injected code are stored in a data storage system. These inputs are later referenced and executed on a web page. Similar to reflected XSS, stored XSS also results from insufficient sanitization of user inputs in server-side code. DOM-based XSS occurs when client-side scripts execute malicious payloads as a result of modifying the document object model (DOM) structure of a web page. It functions quite similarly to reflected XSS, but exploits the vulnerability on the client-side code instead. Stored DOM-based XSS can also occur if the malicious scripts are stored in client-side storage (e.g., a cache).

Understanding the nature of XSS is crucial to defending against XSS attacks. That is to say, in order to successfully circumvent the consistent incursions of new attacks, a thoroughly updated knowledge on the type of XSS involved is critical. It is by only this knowledge can a good strategy for developing new approach to improve the existing defense mechanisms of internet security be initiated. This points to the fact that fighting these attacks is an ongoing challenge for security experts.

2.2 Vulnerability Contexts in Web Applications

In this sub-section, we discuss how web applications are vulnerable to XSS. Web applications are designed to be user interactive. For this interactivity to take place, applications dynamically construct an HTML page within the server based on inputs from the application's user. The page is initially an HTML *template* containing *data placeholders* that are completed by user inputs to form hard-coded strings. These strings will eventually be converted to code when they are presented to the browser.

The problem begins to occur when the untrusted user inputs manage to make their way into vulnerable contexts without proper sanitization. As these unsanitized inputs are presented back to the browser through vulnerable contexts, applications may become vulnerable to XSS. A *vulnerability context* is the environment surrounding the data placeholders in the template. These contexts, namely (1) HTML element content, (2) HTML attribute value, (3) URI query value, (4) CSS value, and (5) JavaScript value [6][7], are usually, but not necessarily, bound by quote delimiters. **Table 1** shows examples of the HTML and Attribute contexts with different delimiters, i.e., single, double, and no quotes [8]. The `input` represents the data placeholder in the template.

Table 1. Web Application Injection Context

	Examples		
Context	**Single Quoted**	**Double Quoted**	**Unquoted**
HTML	-	-	`<p>input</p>`
Attribute	`<p id='input'></p>`	`<p id="input"></p>`	`<p id=input></p>`

An XSS vulnerability *exploit* is successful when there is a syntactically correct insertion of the malicious payload in place of the data placeholder, where a part of the payload expands out of the vulnerable context. In other words, often the malicious payload will be executed in the browser only after it has broken out of the context it is currently in. This follows what Lekies et al. pointed out, i.e., that the structure of a payload often consists of a break-out sequence, an exploit string and an escape sequence [7]. The *break-out sequence* is the first part of the payload to escape the current context where script execution is possible. The *exploit string* is the part of the payload that is executable, while the *escape sequence* is the part that voids the trailing string after the placeholder in order to avoid interfering with the execution of the exploit string. To illustrate an example, let us look at the Attribute context in **Table 1** and consider its exploit string: `<script>alert(1)</script>`. We will see that if we are to execute the exploit string successfully in the output, all we need is to break out of a single, double, or unquoted Attribute context that is present before the data placeholder. To do this, we append the delimiter `'>`, `">`, or `>`, accordingly, in front of the exploit string to obtain an HTML output string, as follows:

Listing 1: Exploit Examples

```
<!-- Single Quote -->
<p id=''><script>alert(1)</script>'></p>

<!-- Double Quote -->
<p id=""><script>alert(1)</script>"></p>

<!-- No Quote -->
<p id=><script>alert(1)</script>></p>
```

Note that no escape sequences are required in this case, as the trailing `'>`, `">`, and `>` strings have now merely become plaintext that precedes the closing tag `</p>` in the HTML.

All in all, relevant solutions need to be applied at appropriate vulnerable contexts to nullify malicious strings in the untrusted input. This is especially true for strings that match the possible break-out sequence of the context and the subsequent exploit string.

2.3 The Critical Need for Open-source Cross-site Scripting Filters

In order to prevent XSS vulnerabilities from being exploited, XSS filters have become a necessary tool for web applications. These filters are applied to the server-side code when receiving user inputs or when supplying outgoing HTML to external processors. In general, XSS filters contain sanitization functions that either remove certain keywords (e.g., `script`, `javascript` and `eval`) or encode special characters (e.g., single and double quotes), each of which has a certain meaning to the server- or client-side code.

The task of developing filters as open-source software has since become imperative to the development of web applications. However, we believe that this task should involve a party apart from web application developers as it demands web security expertise. By making these filters available for free, a great service can be provided to web developers who would feel secure to run their applications and help web enterprises move forward.

3. Experimental Setup

The goal of this experiment is to evaluate the performance of filters in preventing well-known XSS attack payloads in web applications. Specifically, the experiment aims to help us (1) assess the effectiveness of each filter in preventing attacks and (2) verify that there is no impact on the benign payloads going through each filter. What follows is the description of our experimental setup, which includes our methods of (1) collecting payload test cases, (2) preparing the vulnerable web application, (3) setting up XSS filters, (4) automatic testing, (5) determining successful attacks, and (6) modifying filter settings in our experiment.

3.1 Payload Collection

The experiment requires a set of well-known XSS attack payloads and benign payloads as test cases. This sub-section describes how we obtain our payloads.

Table 2. Carriers in Malicious Payloads

Type	Payload Examples	Carrier	Percentage (%)
Event[1]	`<IMG SRC='image.jpg' onclick="alert('XSS')">`	`onclick`	60.3
Script[2]	`<script>alert('XSS')</script>`	`script`	27.7
URI[3]	`<a href="javascript:alert('XSS')">`	`href`	12.1

[1] Event Attributes, [2] Script Elements, [3] URI Atttributes

For the malicious payloads, we obtained the raw set by manually collecting from several security organization websites that supply common XSS attack payloads prepared for testing and research purposes [9][10][11][12]. We then ran them on the Mozilla Firefox browser and selected 382 successfully executed exploit strings in the HTML context (see Section 3.2 for more details) as our test set. We define the part of the payload that bears the exploit strings as the *carrier*. **Table 2** denotes the three types of carriers in HTML element payloads and provides examples. All the exploit strings include scripts to trigger an alert message box when executed.

Some web applications, such as blogs and wikis, allow users to format input texts in a certain part of the HTML page and to share links from other resources among different users. We treat these forms of input texts and sharing of links as benign payloads. In our attempt to examine the extent of a filter's ability to correctly handle these types of payloads, we carefully select 175 benign payloads from several sources [13][14][15][16] to act as our second test set. To represent possible user inputs in general applications, the set consists of alphanumeric character texts, with or without special characters, in respect of the HTML input element types such as dates, time and files [17]. It also consists of syntactically inert HTML tags and attributes in respect of image and text-related HTML elements [18] to represent possible inputs

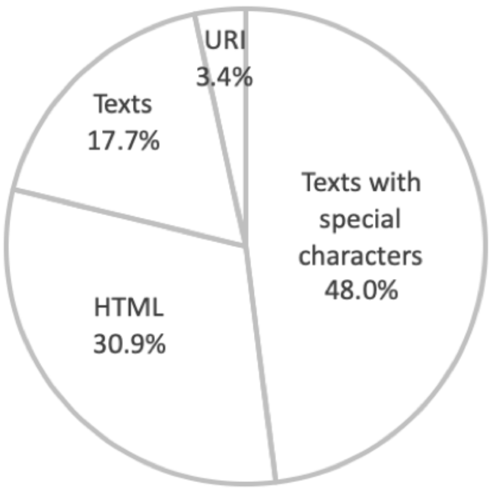

Fig. 1. Characteristics of Benign Payloads.

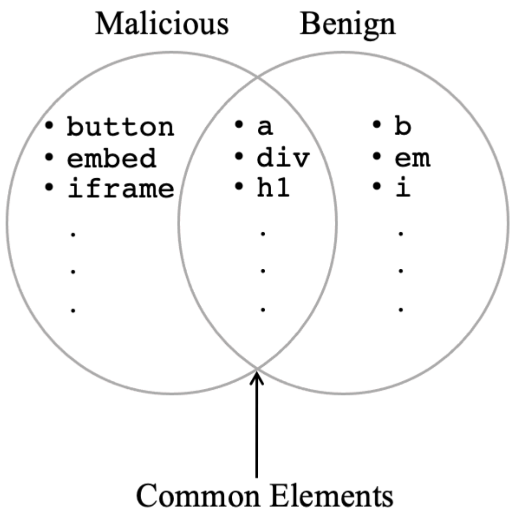

Fig. 2. Overview of Payload Test Set.

in user-customizable applications.

Finally, the characteristics of the benign payloads are shown in **Fig. 1** and an overview of the HTML elements in both test sets is shown in **Fig. 2**. We refer to the elements in the intersection as *common elements*. These elements are intended to enrich the functionality of an application, but they can also be abused while carrying malicious payloads.

3.2 Model of Vulnerable Web Application and Context

Our vulnerable web application has a single input form where a user submits an input payload. Upon submission, the server receiving the payload from the browser will use it to invoke the filter. After being sanitized, the payload appears in the browser.

As shown in the examples of Section 2.2, each payload can be embedded into multiple vulnerable contexts at the program output. However, we employ one vulnerable context in our application because, although some of the filters provide different sanitization routines for different contexts, the other filters do not. This suggests that either the filters are built only to protect a single context or that they generically perform sanitization on payloads. In other words, they would sanitize a single payload equally in different contexts, which implies that they are context-insensitive. Therefore, for the sake of consistency, we created our web application to be vulnerable only in one context; the HTML context.

We place the filters right before the final output (i.e., HTML output) in the application's server-side code. Additionally, because the filters in this study are implemented in different languages, we create three separate but similar vulnerable applications in JSP, Node.js and PHP.

3.3 Dynamic Open-source Cross-site Scripting Filters

We selected 12 available dynamic open-source XSS filters in current use: jsoup Java HTML parser or *jsoup* [19], *Lucy-XSS* [20], *XSS HTML Filter* [21], *xssprotect* [22], *sanitize-html* [23], *secure-filters* [24], *xss* [25], *xss-filter* [26], *HTML Purifier* [27], *PHP Anti-XSS* [28], *PHP-XSS-Filter* [29], and *xss_clean* [30]. These filters are designed to provide protection against well-known reflected, stored, and/or DOM-based XSS attacks.

Table 3. Cross-site Scripting Filters, Language and Filtering Types

Name of Filters	Version Used	Filter Language	Subject Policy		Input Form		Encoding or Escaping	HTML Validation
			WL	**BL**	**String**	**Parse**		
jsoup	1.12.1	Java	●			●		●
Lucy-XSS	1.6.3	Java	●			●	●	●
XSS HTML Filter	1.5	Java	●		●		●	●
xssprotect	0.1	Java		●		●		●
sanitize-html	1.20.0	Node.js	●	●		●	●	●
secure-filters	1.1.0	Node.js		●	●		●	
xss	1.0.6	Node.js	●	●		●	●	
xss-filter	0.5.3	Node.js		●	●			
HTML Purifier	4.11.0	PHP	●			●	●	●
PHP Anti-XSS	1.2b	PHP	●	●	●		●	
PHP-XSS-Filter	1.1	PHP		●	●		●	
xss_clean	-	PHP		●	●			

For the purpose of analysis, the 12 filters are classified, as shown in Table 3, into four categories based on their main filtering types: *subject policy*, *input form*, *encoding or escaping*, and *HTML validation*. The first category constitutes the subject policy of strings, or a set of strings via regular expression (regEx), used for sanitization. That is, a *whitelist* policy (WL) lists the allowed elements, while a *blacklist* policy (BL) lists the disallowed elements. The second category is related to the form of the input to be searched, as in *string* or *parse tree* (parse), to identify a set of whitelisted or blacklisted elements. The third category represents special character encoding or escaping (i.e., converting untrusted inputs into passive characters). Lastly, HTML validation filters payloads by validating their HTML structure.

In this experiment, we use the default settings of filters to closely match what would be employed by a naïve user. We use the latest version of each filter (at the time of this writing) and will consider newer versions of filters in our future work. Below, we briefly explain the settings we used for the experiments.

- *jsoup*. We use the available basic whitelist and add the attribute `id` to the allowed attribute list via the `addAttributes` function. We note that all the other filters are set to allow the attribute `id` in the default setting (see Section 3.5).
- *Lucy-XSS*. We use the default whitelist filter security policy of the `lucy-xss-default.xml` file.
- *XSS HTML Filter*. We use the available whitelist policy in the `HTMLFilter.java` file that consists of a list of safe entities, including tags (e.g., `a`, `em`, and `img`) and attributes (e.g., `href` and `src`).
- *xssprotect*. We use the available blacklist policy in the `XSSFilter.java` file for the default setting. This consists of a list of forbidden elements, such as tags (e.g., `script`, `embed`, and `object`), attributes (i.e., any attributes containing the `on` keyword), and attribute values or schemes (e.g., `javascript:`, `vbscript:`, and `mocha:`).
- *sanitize-html*. We use the available default option, which contains a whitelist of allowed tags (e.g., `h3`, `b`, and `strong`) and allowed attributes (e.g., `href`), as well as a blacklist of forbidden tag elements (i.e., `script`, `style` and `textarea`).

- *secure-filters*. As opposed to the other filters in our study, it is the only filter that also provides sanitization functions for contexts other than HTML such as JavaScript, URI, and CSS. However, we use the HTML context sanitization function (for reasons explained earlier in this section).
- *xss*. We use the available default whitelist that contains allowed tags (e.g., `a`, `br`, and `code`) and their respective attributes (e.g., `href` for the `a` tag and `src` for the `img` tag).
- *xss-filter*. We use the available blacklist, which contains prohibited event attributes such as `onclick` and `onfocus`.
- *HTML Purifier*. We use the default configuration of the `HTMLPurifier_Config::createDefault()` function.
- *PHP Anti-XSS*. We use the whitelist filter, which will return strings with patterns containing only alphanumeric characters (i.e., combinations of letters and numbers), successively with the blacklist filter.
- *PHP-XSS-Filter*. We use the available function, `filter_it()` to call the filter.
- *xss_clean*. We use the available function `clean_input()` to call the filter.

Overall, we found that all the filters provide checks on HTML tags. Additionally, all but *secure-filters* and *PHP Anti-XSS* provide checks for HTML attribute elements. Similarly, all but *Lucy-XSS*, *secure-filters*, *PHP Anti-XSS*, and *xss-filter* provide checks for URI schemes. Additionally, all filters, with the exception of *jsoup*, *XSS HTML Filter*, *secure-filters*, and *PHP Anti-XSS*, perform checks for CSS or style tags. Although, *jsoup* provide checks for URI values, no filter provides checks for JavaScript values.

3.4 Test Methodology

We apply the technique of black-box testing [31] to help us automatically probe and identify security vulnerabilities in a web application without having to access the application's internal structure (i.e., its source code).

In our experiment, we avoid using web application scanners because a study has shown that these are prone to false negatives [32]. Instead, we create our own automated XSS testing framework to assess the filtering capabilities of filters.

We first prepare the payloads, as a single batch for the testing framework. The framework then submits the payloads to the server, i.e., the web applications equipped with XSS filters. The server receives one payload at a time and uses this as an argument to invoke the filter. The filter provides the sanitized payload to the application, which is included in the HTTP response that is sent back to the browser. Eventually, the framework reports whether the sanitized payload has been correctly or incorrectly filtered by the filters (see Section 3.5).

We use a 32-bit Ubuntu 16.04 LTS machine with 3.8GB of memory to conduct all experiments and a 3.20 GHz Intel Core i5 650 processor to host the web application. For the PHP and Node.js filters, we use PHP version 7.0.33-0ubuntu0.16.04.1, Apache Server version 2.4.18, and Node.js version 4.4.2. To execute the Java filters on the Netbeans IDE 8.0.1, we use Glassfish server 4.1. The browser we use is Mozilla Firefox version 65.0.

3.5 Identification of Correct Filtering

Below, we explain how we identify whether the malicious and benign payloads are correctly-filtered in this experiment. For malicious payloads, by using Selenium WebDrive [33], we check to see if the scripts in malicious payloads are no longer executable by observing the absence of popup boxes in the web page response.

In certain cases, scripts may require an extra action before they can be executed. For example, the call to the alert function in the following payload is only executed when the user clicks on the image:

`<IMG SRC='image.jpg' onclick="alert('XSS')">`.

To counter this problem, we add the same identifier to each of the payloads via the HTML `id` attribute. Then, we add a jQuery `trigger` function in our vulnerable web application that calls the identifier from the payload and triggers any existing HTML events that are associated with it. To illustrate this, the identifier is added to the payload as follows:

`<IMG id='identifier' SRC='image.jpg' onclick="alert('XSS')">`,

which will be triggered by the following function:

```
('identifier').trigger(
     'onclick',
  // other HTML events
);
```
.

In the case of benign payloads, our checking for successful filtering is not done only by observing the changes on the payload in the web page response alone. This is because some filters perform tag balancing for tags that are missing or unbalanced. In another case, some filters, such as *Lucy-XSS*, provide information on what was removed from the input. Therefore, we determine the correctly-filtered benign payloads by comparing the HTML structure based on the abstract syntax tree of the payload with the output [34] and check whether there are any deletions (while ignoring additions) from the input. Then, we manually verify whether there are any changes to the payload in the output strings in the final check.

3.6 Modification of Filter Settings

In our attempt to identify possible issues in the sanitization of payloads in the as-is condition, we preemptively explore the performance of the filters in their default settings. That is to say, based on the idea that a filter's main goal is to prevent XSS, we reconfigure the settings of the filters so that they sanitize malicious payloads that were not correctly sanitized in the initial test. Meanwhile, to review their performance against benign payloads, we add our benign element set to their whitelist. If the filters achieve perfect results on malicious payloads, we can determine whether modifications influence their ability to handle benign payloads. Additionally, our modification of the configured settings involves the examination of the filtering techniques individually.

Theoretically, we assume a filter should produce highly correct filtering against both malicious and benign payloads. For the purpose of elaboration, the output results of each category are discussed with respect to the performance issues in subsequent sections.

4. Analysis and Results

To analyze the performance of the 12 filters on a total of 382 malicious XSS payloads and 175 benign payloads, we classify the filtered outputs according to three criteria, which we define as follows:

1. Correctly-filtered: Popup boxes do not appear in the response page, indicating that the malicious payloads are properly filtered. For example,
   (a) the malicious payload, `<script>alert('XSS');</script>` becomes `alert('XSS');` or,

(b) the benign payload, `<b>Safe Text</b>` remains as is after going through the filter.

2. Under-filtered: Popup boxes appear in the response page, indicating that the malicious payloads are improperly filtered, thus allowing unwarranted execution in the browser. For example,
   (a) the malicious payload, `<script>alert('XSS');</script>` remains as is even after being filtered or,
   (b) the malicious payload, `<scr<script>ipt>alert('XSS');</scr</script>ipt>` becomes `<script>alert('XSS');</script>` after being filtered.
3. Over-filtered: Payloads are changed in the response page, indicating that the benign payloads are improperly filtered. For example, the benign payload, `<b>Safe Text</b>` becomes `Safe Text` or `&lt;b&gt;Safe Text&lt;/b&gt;`.

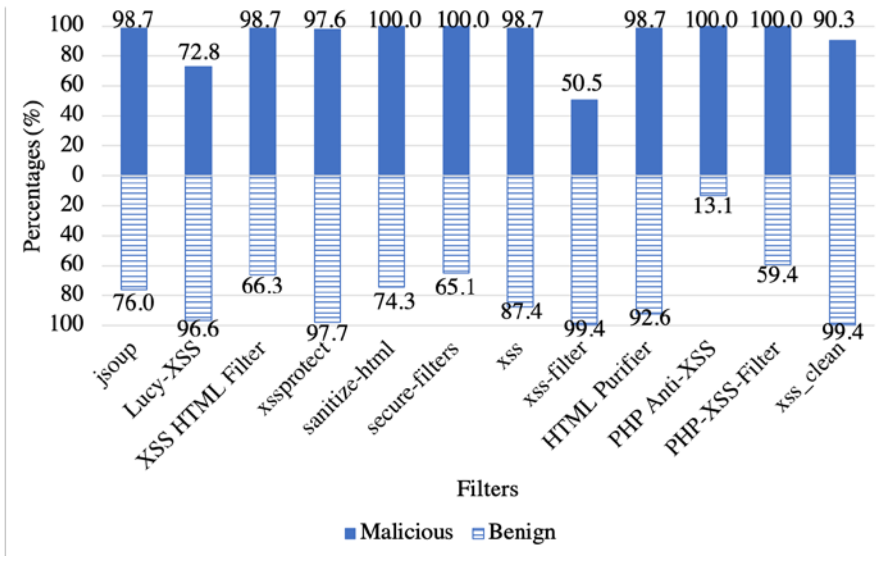

Fig. 3. Utilizability of XSS Filters Against 382 Malicious Payloads and 175 Benign Payloads.

4.1 Performance of Filters

Fig. 3 presents the utilizability of the 12 filters for 382 malicious payloads using the default settings of filters. Four filters, namely: *sanitize-html*, *secure-filters*, *PHP Anti-XSS* and *PHP XSS-Filter*, perform well by correctly filtering all malicious payloads. Based on this result, we are inclined to believe that these filters are utilizable in real-world applications to prevent well-known XSS payloads. The following five filters, namely: *jsoup*, *XSS HTML Filter*, *xssprotect*, *xss*, and *HTML Purifier*, perform correct filtering more than 97% of the time. While *xss_clean* performs correct filtering slightly more than 90% of the time, and *Lucy-XSS* more than 70% of the time, *xss-filter* manages to correctly filter only about half of all malicious payloads.

As for the utilizability of the filters on 175 benign payloads, none of the filters secure 100% success. *xss-filter* and *xss_clean* performed excellently but each had one non-success. *xssprotect*, *Lucy-XSS*, and *HTML Purifier* succeed acceptably with correct filtering more than 92% of the time, surpassing *xss*, *jsoup*, and *sanitize-html*, which succeeded in 74%- 87% of the payloads. Quite disappointingly, *XSS HTML Filter*, *secure-filters*, and *PHP-XSS-Filter* secured slightly more than half of the payloads. *PHP Anti-XSS* was successful 13% of the time.

In general, we can say that most of the filters are reasonably effective at preventing the majority of well-known XSS payloads via their default settings. However, they are not as effective in handling benign payloads. We will identify some of the issues of ineffective filters in particular cases in this study. We will also discuss these issues in reference to the two categories mentioned in the earlier part of this section, i.e., the under-filtering and over-filtering.

4.1.1 Issues on Under-filtering

Of the 12 filters we examined, we ascribe failure to secure 100% correctly-filtered malicious payloads in eight of them to under-filtering. That is to say, these filters are not sufficiently equipped to handle the malicious payloads, resulting in their unwarranted execution in the browser.

We contend that some of the common issues of under-filtering are related to payloads with URIs embedded in `href` attributes. We discover that the attacks are URIs that contain obfuscated scripts. However, even though *jsoup* and *HTML Purifier* check URI values, they were yet unable to check whether the values contain malicious scripts. We later learn that in the case of *jsoup*, it only performs to validate the payloads that contain only UTF-8 characters. This leads us to believe that the issue of under-filtering has something to do with the filters' ability to first identify whether the linked page is legitimate or malicious.

We would also like to note that either 1) the exclusion of elements with possible carriers in the blacklist, as is the case for *xssprotect*, *xss-filter*, *xss_clean*, or 2) the inclusion of these elements in the whitelist, as is the case for *Lucy-XSS*, is the reason for under-filtering. For example, the `form` attribute is a possible carrier of the `button` element [10]. If the use of this element as an input to the applications is allowed, the whitelist policy of the filter should not include the `form` attribute, while the blacklist policy should. In a case where the application does not allow such an element, the element itself should be excluded from the whitelist and included in the blacklist.

Additionally, we also find that filters that process string input forms, either as the main routine or a part of its sanitization routine, may allow cases of malicious scripts with obfuscated characters in the payload to pass through unfiltered. In other words, when special characters are replaced with their respective encoded entities, there is a chance they will be automatically decoded upon reaching the browser. For instance, filters that prevent the `javascript:` scheme from being present in the payload would mistakenly allow an obfuscated version of the scheme, i.e., `javascript&colon;`, to pass through unfiltered.

4.1.2 Issues on Over-filtering

Here, we ascribe the lack of 100% correctly-filtered benign payloads in 10 filters to over-filtering. This implies that these filters have difficulties handling benign payloads. Common over-filtered payloads are those with elements belonging to HTML classes and texts with special characters. We observe that the inability of some filters to handle benign payloads successfully seems to be related to the following two possible reasons: (1) the use of API functions that blindly encode any special characters present in text payloads, and (2) the exclusion of syntactically inert HTML elements in the whitelist.

4.1.3 Summary of Analysis and Results

In summary, when using their default settings, filters are generally more able to handle under-filtering than over-filtering. The common critical issues in under-filtering are the handling of malicious URIs, the entries in the policy and the search on string-based input forms. Whereas in over-filtering, they are the use of encoding functions and the insufficient entries in whitelist.

In extension, we generalize the benefits of the filters. In their default setting, we may say that filters with low under-filtering and high over-filtering are utilizable for high-security risk applications, such as banking and electronic stocks, or applications that do not accept HTML strings as input. As for the filters with high under-filtering and low over-filtering, they are utilizable for applications that allow users to submit syntactically inert HTML, such as blogs and wikis, although this risk compromising the security of the application.

4.2 Improving Filter Performance

We attempt to find ways to improve the filter performance by observing two conditions. We aim to find out (1) whether there is another setting for each filter that can enhance the filter's ability to correctly filter malicious payloads and (2) whether using the reconfigured setting affects its performance against benign payloads. For this purpose, we reconfigure the default settings for the filters and measure their performance based on correct filtering. We do this either by utilizing other available functionalities provided by the filters, or by modifying the filter's policy. That is, in order to sanitize properly malicious elements that were previously under-filtered, we set the reconfigured settings for the following filters, listed based on the policy used, as such:

1. Whitelist-only filters:
   - *jsoup*, *XSS HTML Filter*, and *HTML Purifier*. We remove the `href` attribute of the `a` tag.
   - *Lucy-XSS*. We remove the elements `embed`, `form`, `iframe`, `input`, `meta`, `object`, and `script`, as well as the attribute `href` from the `a` element.
2. Blacklist-only filters:
   - *xssprotect*. We add `href` and `xlink:href` to the attributes list of the blacklist, remove the character `:` from the `javascript:` scheme in the attribute values list, and remove the element `form` from the element list.
   - *xss-filter*. We enable the `escape` option that encodes all `<` and `>` symbols.
   - *xss_clean*. To the blacklist, we add the tag `base`, the attribute `href`, and the `&colon;` for malicious payloads that use `javascript&colon` instead of `javascript:`. We also modify the regEx `on` to `\/?on` for payloads that use events starting with `/on` instead of `on`. Finally, we set the closing bracket of each tag `>` as optional.
3. Whitelist and blacklist filters:
   - *sanitize-html* and *PHP Anti-XSS*. We did not make changes for malicious payloads because there were no under-filtered payloads.
   - *xss*. We remove the attribute `href` of the `a` element from the whitelist.

Here, we consider common elements that were previously under-filtered as malicious. We also add all elements, if not already present, from our benign test set to filters with a whitelist policy (refer to **Fig. 2** for an overview). Note that two filters, i.e., *secure-filters* and *PHP-XSS-Filter*, are excluded from this experiment. While only utilizing a blacklist policy, these filters achieved 100% success with malicious payloads; thus, they cannot be reconfigured to improve their performance with benign payloads.

We presume that adding elements to a filter's whitelist will increase its correct filtering performance on benign payloads. In contrast, adding common elements to a filter's blacklist will decrease its performance. Our results show that the filters with reconfigured settings could filter all malicious payloads. Nevertheless, their handling of benign payloads showed mixed results.

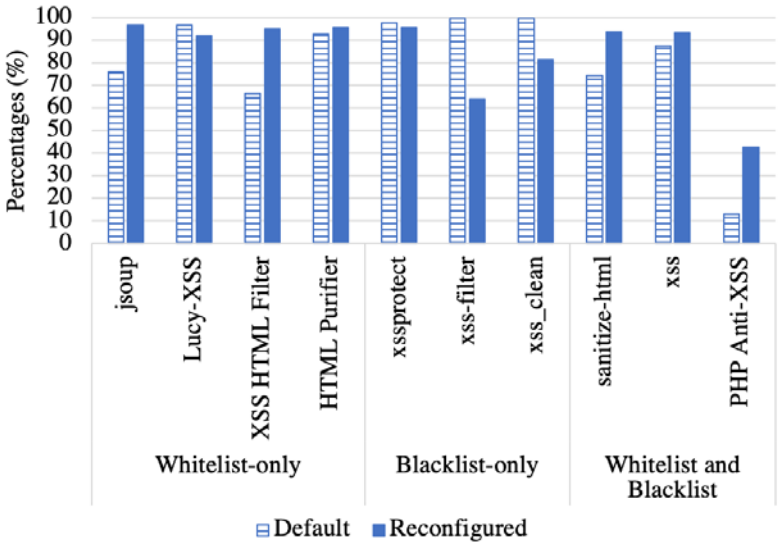

Fig. 4. The Performance Differences of XSS Filters Against 175 Benign Payloads for the Default and Reconfigured Settings.

Fig. 4 presents the differences of the filters' performance on benign payloads with the default and reconfigured settings. We see that, when using whitelist-only filters, *jsoup* and *XSS HTML Filter* improve their performance against benign payloads by about 20%, while *HTML Purifier* is slightly improved by 3%. However, *Lucy-XSS* showed a slight decline of about 5%. Meanwhile all whitelist and blacklist filters improve their performance by 6% to 30%. In contrast, for blacklist-only filters, *xssprotect* shows a slight decline in performance of about 2%, while the performance of *xss-filter* and *xss_clean* drops significantly by 35% and 18%, respectively.

Next, we inspect the over-filtered payloads. As expected, all of the filters over-filtered payloads containing the common elements (e.g., the `href` attribute of the `a` tag). This is caused by the removal of those elements from the whitelist or the inclusion of them in the blacklist, which was done in an attempt to prevent malicious payloads. This is an indication that there is a conflict in deciding whether or not to allow elements that can both be benign and malicious.

We also found that the addition of benign elements to a filter's whitelist policy improved the performance on benign payloads for most filters, except *Lucy-XSS*. The slight increase in performance may be caused by the fact that most elements were already present in the default whitelist. Likewise, the reduced performance of *Lucy-XSS* may be due to the additional configuration required for nested elements. To illustrate, for the benign payload, `<p>This is an <i>italic</i> text</p>`, the `i` tag should be declared in the policy as an allowed nested element for the `p` tag. *HTML Purifier* also required this configuration; however, unlike *Lucy-XSS*, most of the rules for nested elements were already present in its default whitelist, which explains why its performance did not decline. This shows that, by adding rules for nested elements, *Lucy-XSS*'s performance can be improved, and our earlier presumption still holds. Additionally, the performance of *PHP Anti-XSS* declined significantly because of the encoding and upper- to lower-case character transformation function.

The performance of blacklist-only filters declined after reconfiguration. However, the performance reduction of *xssprotect* is lower than its counterparts; this may be because of its different input form. *xssprotect* performs a search on the parse tree rather than the string, which is the form used by *xss-filter* and *xss_clean*. In fact, the significant decline in performance for *xss_clean* may be caused by our modified regEx for the `on` keyword, which over-filters 17% of our benign test set containing the keyword. As for *xss-filter*, the reduced performance is due to the activation of the encoding function, which we also see for *PHP Anti-XSS*. This can have a negative effect on a filter's performance on benign payloads. It is noteworthy that *PHP Anti-XSS*, *xss-filter*, and *xss_clean*, also use blacklists and string-based input forms. As for whitelist and blacklist filters, the over-filtered payloads are similar to those of the whitelist-only filters, though their performance did not decline because there were more elements added to the whitelist, as compared to those that were removed from the blacklist.

One thing worth mentioning is that two of our benign payloads containing special characters happen to be similar to HTML tags. As the tag element is not included in any whitelist, it was incorrectly sanitized by the filters. In another case, our JSP web application includes the HTML charset attribute by default, resulting in the immediate encoding of two internationalized domain names of the same payload class upon reaching the browser. However, since there are only two cases of this type of payload, it does not significantly affect the filtering results.

Next, we would like to find out whether it is possible for a filter to achieve perfect results for both malicious and benign payloads. For this, we chose one filter, i.e., *jsoup*, which over-filtered only the `href` attributes of the `a` tag. This time, we allowed the attribute in the policy and then retested the filter with only the URI values of both the malicious and benign payloads. Indeed, the filter is capable of correctly filtering all payloads. This suggests that it is possible to achieve perfect results if the filter performs sanitization on the URI values. Since we analyzed the filter only in the context of HTML, we can say that it is possible for the filters to correctly handle both malicious and benign payloads within the context.

To recap, the executability of a malicious payload depends on the vulnerability context. In section 3.2, we mentioned that the filters are insensitive to context; therefore, they might not work as well in other contexts. To show that this is a problem, we tested our improved version of *jsoup* and placed it in the Attribute context [35] and the JavaScript context[1] in our web application as shown below:

Listing 2: Filter in Attribute value context

```
<img src='img.jpg'
width='Jsoup_filter(input)'>

// exploit string: '
onerror='alert(1);

// resulting HTML output:
<img src='img.jpg' width=''
onerror='alert(1);'>
```

Listing 3: Filter in JavaScript value context

```
eval(\"document.write('+
Jsoup_filter(input))+');\");

// exploit string: ');%20alert('1


// resulting HTML output:
eval("document.write('');
alert('1');");
```

[1] https://security.stackexchange.com/questions/52558/exploit-document-write-with-unsanitized-user-input

We then fed the application two malicious payloads that are executable in their respective contexts. The payloads do not contain HTML tags; therefore, they are not executable in the HTML context. We found that the filter under-filtered the payloads, which lead to the execution of malicious script. These examples further confirm that even if a filter is utilizable in the HTML context, it might not work the same in other contexts.

To summarize, our findings indicate that (1) there is certainly another setting for a filter that can improve its performance, (2) reconfiguring the filter's policy does effect the filter's performance on benign payloads, (3) it is possible that a filter can correctly handle both malicious and benign payloads in the HTML context, and (4) the performance of filters in the HTML context does not necessarily reflect their performance in other contexts.

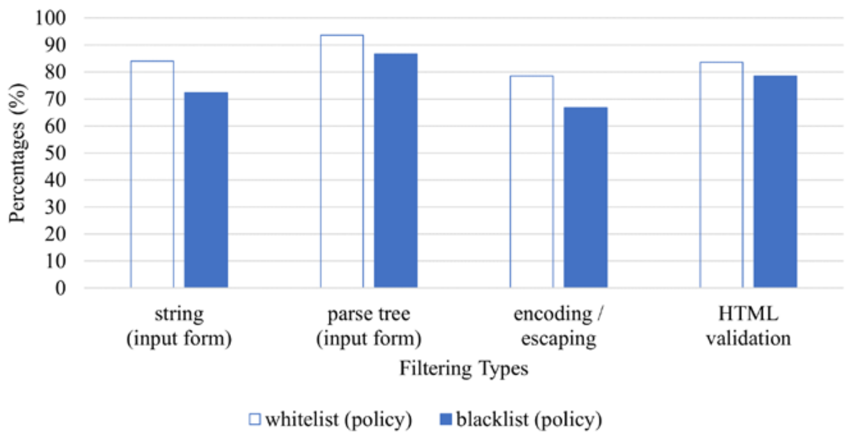

Fig. 5. Performance According to Filtering Types with Reconfigured Settings on Benign Payloads.

4.3 Performance Based on Filtering Techniques

We observe that the performance of filters is very likely related to their filtering types and their ability to recognize different HTML elements within the context of HTML. To validate this assumption, we examine the degree to which a filter can correctly filter benign payloads according to the filtering type. With the reconfigured settings, we enable each filtering type to run separately at a time. That is, within each filter, we alternately enable each function that works to sanitize payloads: (1) based on the policy and input form, (2) by performing encoding and/or escaping on the payloads, or (3) by structuring the payloads via HTML validation. As **Fig. 5** shows, filters of the same policy perform differently with different input forms. We see that when the input form is a parse tree, filters on the whitelist perform better (94%) than they do with a string (84%). The same goes with those using a blacklist policy, which perform more accurately with a parse tree (87%) than they do with a string (72%). Nevertheless, when comparing the performance on the basis of different policies with different input forms, we find that filters using whitelist perform better in either form.

Our examination of filters, using the same policy but with other filtering types, reveals that both the whitelist and the blacklist perform better with HTML validation (84% and 79%, respectively) than they do with encoding or escaping (79% and 67%, respectively). Lastly, when we compare the performance of filters of different filtering types working in combination with different policies, it appears that those combining encoding or escaping and HTML validation work better with the whitelist, than they do with the blacklist.

All in all, improvements on handling benign payloads are more likely when filters are made to run on filtering types based on the right combination of policy and input forms. The filter's performance would likely improve from using a whitelist policy with a parse-tree input form combined with encoding or escaping and HTML validation.

5. Recapitulation

A web application allows users to interact with it by generating a web response based on a user input. The dynamic nature of the application, while necessary, makes it vulnerable to web attacks and exposes it to security risks. While legitimate users submit valid inputs to obtain authorized information from the application, an attacker crafts malicious inputs or payloads to perform web attacks, such as XSS, which can cause the application to execute unauthorized actions. Thus, in order to prevent these actions, web developers have applied defense mechanisms to thwart attacks. One of such defense mechanisms is open-source XSS filters. An XSS filter with adequate capability sanitizes incoming payloads in the server prior to returning them to the browser via the web application's response. During this sanitization process, the filter should be able to disarm malicious payloads effectively and handle/facilitate benign payloads without destroying their original intention. However, we find that existing XSS filters tend to suffer, to various degrees, from issues of under-filtering malicious payloads and over-filtering benign payloads. We have analyzed 12 dynamic open-source XSS filters and assessed their filtering capabilities according to their degree of success or failure. First, we used their default settings, which is typically for this type of study, without making changes [36][37]. We found that only some filters are satisfactory in handling malicious payloads, although they are not equally capable of handling benign inputs. However, our experimental results using reconfigured settings indicate that each filter has at least one configuration that successfully sanitizes all of the well-known XSS payloads we used. However, users with little to no background knowledge in security may encounter difficulties finding the best filter configurations. We also discovered that it is more difficult for filters to manage benign payloads than malicious threats. Most filters tend to over-filter benign payloads, thereby limiting the intended interactive response of users. This is especially true for filters that utilize a blacklist and a string-based search on payloads. Our finding is consistent with the work of Bates et al., who explored the harmful use of regEx in preventing malicious payloads [38]. Although we found that it is possible for a filter to prevent malicious payloads while allowing benign ones, the solutions in this paper are contingent upon our way of testing. That is to say that we tried to solve a filter's issues only after obtaining information related to under- and over-filtered payloads. Therefore, a more robust and automated solution is required to prevent XSS in real-world applications. The effectiveness of a filter depends on its policy coverage. Unfortunately, the completeness of a policy can never be guaranteed. This is because attackers can find new ways to circumvent filters and the development of new technologies might introduce new attack payloads, thereby making it difficult for filter developers to maintain the policy. Additionally, the fact that some HTML elements can be used to carry both a benign and a malicious value causes a trade-off between under-filtering malicious payloads and over-filtering benign payloads. The trade-off between under- and over-filtering (in more general terms, false negatives and false positives), which effects the filter's performance in terms of precision and recall, is a prevalent issue in security engineering. This has been addressed in numerous studies [39][40][41]. Also, current filters are not context-sensitive, i.e., the filters do not consider the context in which the user inputs are used in the output HTML. Rather, the filters prevent XSS by searching for certain patterns or particular elements that are considered safe or malicious; this method can be error-prone and produce false results. This implies that XSS vulnerabilities are context-sensitive [6][7]. Therefore, a context-sensitive filter is needed. Normally, a context-sensitive filter would require the user of the filter to supply information about the vulnerable context as an argument to the filter. This would allow the filter to automatically apply appropriate sanitizers to act on malicious payloads [8]. As current filters act as server-side prevention mechanisms, they may help to mitigate reflected and stored XSS; however, they may not be as effective in preventing DOM-based XSS. Moreover, none of the filters perform checks on JavaScript values. DOM-based XSS exploits the vulnerability in the client-side code that modifies the DOM of the web page. As opposed to reflected and stored XSS, where the vulnerability is caused by accepting a user input from the browser that flows into the server, the flow of input data to the output function occurs in the browser itself during DOM-based XSS. That is, the input does not reach the server.

Improving a filter's ability to handle benign inputs can be achieved through several strategies. First, the distinction between malicious and benign payloads can be improved with context-aware filtering mechanisms in order to determine the correct type of input that should be received at a particular point in the application. For example, HTML tags or functions should not be allowed as attribute values. Secondly, we should also consider which string can be used as a break-out sequence in the context. For instance, the string `')` contained at the start of the payload, i.e., `');%20alert('1` in the example given in Listing 3, can be used to break out of the JavaScript string value context. By providing information about the context, the possible break-out sequence, and the allowed input type, we can improve the precision and recall and use the filters as an XSS prevention mechanism. Lastly, additional checks should be performed on URI values to determine if the URI contains malicious scripts, is hosting a malicious application, or is hosting an application that is vulnerable to XSS. Several studies have attempted this type of identification [42][13][43].

6. Related Work

This section summarizes existing analysis studies in the domain of XSS. McQuade [44] investigated a low-cost open-source black-box web vulnerability scanner alternative to assist agile developers working in small to mid-sized firms. McQuade found that the detection accuracy and test-case coverage of open-source scanners are higher than those of proprietary scanners. Suteva et al. [32] evaluated open-source web vulnerability scanners using a vulnerable web application called WackoPicko. They performed evaluations to assess the scanners' effectiveness in detecting SQL injection and XSS vulnerabilities. Alternatively, our work evaluates XSS filters, rather than web vulnerability scanners, by using a custom web application to assess their weaknesses. Due to the evolution of open-source software development [45], we evaluate open-source XSS filters.

Hill [46] analyzed and evaluated automated XSS tools based on their usability (i.e., ease of use) and functionality. Scholte et al. [47] performed an empirical analysis to investigate how to prevent SQL injection and XSS attacks based on the languages used and their type systems. They determined that these attacks can be prevented by enforcing validation of data types. Fonseca et al. [48] extended the work of Scholte and Balzarotti by studying the characteristics of source-code defects that affect application vulnerability, particularly SQL injection and XSS vulnerabilities. The authors performed analysis on code patches to compare the numbers of vulnerabilities that exist in applications based on weak-type languages (i.e., PHP) and strong-type languages (i.e., Java, C#, and VB). They concluded that applications are less vulnerable to exploitation when written in strong-type languages. Weinberger et al. [8]

analyzed sanitization practices in real-world applications. They compared the features provided by web application frameworks with the features that real-world web applications require. In contrast, we performed a comparative analysis on dynamic open-source XSS filters and attempted to investigate whether the filters' performance against XSS would affect their performance against benign payloads.

7. Conclusions and Future Work

From our study, we have shown that getting a good and effective open-source XSS filter is still an issue. On the default setting, several filters suffer from under-filtering but many more suffer from over-filtering. We managed to successfully sanitize our set of malicious payloads by adjusting the configuration settings of each filter; however, we observed that the filters' performance towards benign payloads are still problematic. It appears that the effectiveness of a filter depends, to a certain extent, on the filter's policy, whether it is whitelist or blacklist, and its input form. Additionally, there is a trade-off in being able to prevent malicious payloads while allowing the usage of benign tags and attributes for HTML elements that can be used to input both types of payloads. It is important to note that while the filters check for patterns or characteristics of payloads that are malicious, they tend to ignore the vulnerability context in which the payloads are received in the output HTML. The condition in which payloads are allowed to break out of the context can be a main cause of injection-related attacks such as XSS. Although it is possible to obtain perfect results for both malicious and benign payloads in the HTML context, finding the filter's appropriate setting to achieve this might not be trivial for web developers, especially those with limited security knowledge. Therefore, one of our aims is to help developers select suitable filters to ensure secure and safe web applications. At the same time, we hope to offer new clues to filter developers who are seeking novel solutions to improve filters. We suggest two possible directions for future work in a similar theme: (1) the development or design of a filter that is more efficient and context-sensitive, and (2) the design of an automatic filter policy update using the latest HTML specification.

References

[1] OWASP Foundation, "OWASP Top Ten." [Online]. Available: https://owasp.org/www-project-top-ten/, Accessed on: Mar. 20, 2019
[2] OWASP Foundation, "OWASP Top 10 2017," 2017. [Online]. Available: https://github.com/OWASP/Top10/issues, Accessed on: Feb. 26, 2019
[3] MITRE Corporation, "CVE Details: The Ultimate Security Vulnerability Datasource," 2013. [Online]. Available: https://www.cvedetails.com/vulnerabilities-by-types.php, Accessed on: Mar. 20, 2019
[4] The Hacker News, "The Hacker News — Cyber Security, Hacking, Technology News." [Online]. Available: http://thehackernews.com/, Accessed on: May 09, 2016
[5] InfoSec Institute, "InfoSec Resources - How to Prevent Cross-Site Scripting Attacks." [Online]. Available: http://resources.infosecinstitute.com/how-to-prevent-cross-site-scripting-attacks/, Accessed on: Mar. 16, 2017
[6] J. Kallin and I. Lobo Valbuena, "Excess XSS: A comprehensive tutorial on cross-site scripting." [Online]. Available: https://excess-xss.com/, Accessed on: Mar. 22, 2017
[7] S. Lekies, B. Stock, and M. Johns, "25 Million Flows Later - Large-scale Detection of DOM-based XSS," in *Proc. of ACM SIGSAC Conf. Comput. Commun. Secur. (CCS 2013)*, Berlin, Germany, pp. 1193–1204, 2013. Article (CrossRef Link)

[8] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of XSS sanitization in web application frameworks," in *Proc. of European Symposium on Research in Computer Security (ESORICS 2011)*, Leuven, Belgium, 2011, *Lecture Notes in Computer Science*, vol 6879, Springer, pp. 150–171, 2011. Article (CrossRef Link)

[9] D. Anderson, "XSSDB Exports." [Online]. Available: http://xssdb.net/, Accessed on: Mar. 07, 2017

[10] Cure53, "HTML5 Security Cheatsheet." [Online]. Available: https://html5sec.org/, Accessed on: Mar. 07, 2017

[11] Kurobeats, "XSS Vectors Cheat Sheet." [Online]. Available: https://gist.github.com/kurobeats/9a613c9ab68914312cbb415134795b45, Accessed on: Mar. 07, 2017

[12] J. Manico and R. Hansen, "XSS Filter Evasion Cheat Sheet," [Online]. Available: https://owasp.org/www-community/xss-filter-evasion-cheatsheet, Accessed on: May 28, 2021

[13] M. Ter Louw and V. N. Venkatakrishnan, "Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers," in *Proc. of 2009 30th IEEE Symp. Secur. Priv.*, Oakland, CA, USA, pp. 331–346, 2009. Article (CrossRef Link)

[14] Microsoft, "Email Address test cases – Testing Testing 1,2,3." [Online]. Available: https://blogs.msdn.microsoft.com/testing123/2009/02/06/email-address-test-cases/, Accessed on: Sep. 24, 2019

[15] D. Miessler, "SecLists." [Online]. Available: https://github.com/danielmiessler/SecLists/, Accessed on: Sep. 24, 2019

[16] W3Schools, "W3Schools Online Web Tutorials." [Online]. Available: https://www.w3schools.com/, Accessed on: Mar. 15, 2017

[17] MDN, "<input>: The Input (Form Input) element - HTML: Hypertext Markup Language | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input, Accessed on: May 06, 2020

[18] MDN, "HTML elements reference - HTML: Hypertext Markup Language | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element, Accessed on: Sep. 24, 2019

[19] J. Hedley, "Prevent cross site scripting with jsoup." [Online]. Available: https://jsoup.org/cookbook/cleaning-html/whitelist-sanitizer, Accessed on: Feb. 26, 2019

[20] Naver Corp., "Lucy-XSS." [Online]. Available: https://github.com/naver/lucy-xss-filter, Accessed on: Feb. 26, 2019

[21] J. O'Connell, C. Hendersen, and M. Wever, Semb, "XSS HTML Filter: A Java library for protecting against cross site scripting." [Online]. Available: http://finn-no.github.io/xss-html-filter/, Accessed on: Dec. 22, 2016

[22] G. Toonstra, "xssprotect." [Online]. Available: https://code.google.com/archive/p/xssprotect/, Accessed on: Feb. 26, 2019

[23] P'unk Avenue, "sanitize-html." [Online]. Available: https://www.npmjs.com/package/sanitize-html, Accessed on: Feb. 26, 2019

[24] Yahoo! Inc., "Secure XSS Filters." [Online]. Available: https://www.npmjs.com/package/xss-filters, Accessed on: Feb. 26, 2019

[25] Z. Lei, "xss." [Online]. Available: https://www.npmjs.com/package/xss, Accessed on: Feb. 26, 2019

[26] L. Shi, "xss-filter." [Online]. Available: https://github.com/superRaytin/xss-filter, Accessed on: Feb. 26, 2019

[27] E. Z. Yang, "HTML Purifier - Filter your HTML the standards-compliant way!" [Online]. Available: http://htmlpurifier.org/, Accessed on: Dec. 22, 2016

[28] C. Bolat, "PHP Anti-XSS Library." [Online]. Available: https://code.google.com/archive/p/php-antixss/, Accessed on: Dec. 22, 2016

[29] Mario, "PHP-XSS-Filter." [Online]. Available: https://github.com/JBlond/PHP-XSS-Filter, Accessed on: Feb. 26, 2019

[30] M. Bijon, “xss_clean.” [Online]. Available: https://gist.github.com/mbijon/1098477, Accessed on: Dec. 22, 2016

[31] S. Kojarski and D. H. Lorenz, “Comparing White-Box, Black-Box, and Glass-Box Composition of Aspect Mechanisms,” in *Proc. of the 9 International Conference on Software Reuse (ICSR 2006)*, Turin, Italy, pp. 246–259, 2006. Article (CrossRef Link)

[32] N. Suteva, D. Zlatkovski, and A. Mileva, “Evaluation and Testing of Several Free / Open Source Web,” in *Proc. of 10th Conf. Informatics Inf. Technol.*, Bitola, Macedonia, no. Ciit, pp. 221–224, 2013.

[33] B. Muthukadan, “WebDriver API — Selenium Python Bindings 2 documentation.” [Online]. Available: http://selenium-python.readthedocs.io/api.html, Accessed on: Jul. 12, 2018.

[34] E. Gavryushin and V. Grinenko, “html-differ - npm.” [Online]. Available: https://www.npmjs.com/package/html-differ, Accessed on: Sep. 24, 2019

[35] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, “Precise client-side protection against DOM-based Cross-Site Scripting,” in *Proc. of 23rd USENIX Secur. Symp. (SEC 2014)*, San Diego, CA, USA, pp. 655–670, 2014.

[36] C. Arthur, “Why the default settings on your device should be right first time | Technology | The Guardian,” 2013. [Online]. Available: https://www.theguardian.com/technology/2013/dec/01/default-settings-change-phones-computers, Accessed on: Oct. 21, 2019

[37] Panda Security, “Default Settings, and Why the Initial Configuration is not the Most Secure,” 2017. [Online]. Available: https://www.pandasecurity.com/mediacenter/security/default-settings-initial-configuration-not-secure/, Accessed on: Oct. 21, 2019

[38] D. Bates, A. Barth, and C. Jackson, “Regular expressions considered harmful in client-side XSS filters,” in *Proc. of 19th Int. Conf. World wide web (WWW 2010)*, Raleigh, North Carolina, USA, pp. 91-100, 2010. Article (CrossRef Link)

[39] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and a D. Keromytis, “Detecting Targeted Attacks Using Shadow Honeypots,” in *Proc. of USENIX Security Symposium (SSYM' 2005)*, Berkeley, CA, USA, pp. 129–144, 2005.

[40] B. V. Chess and G. E. McGraw, “Static analysis for security,” *IEEE Secur. Priv.*, vol. 2, no. 6, pp. 76–79, Nov.-Dec. 2004. Article (CrossRef Link)

[41] S. J. Murdoch and R. Anderson, “Tools and Technology of Internet Filtering,” *Access Denied: The Practice and Policy of Global Internet Filtering*, The MIT Press, ch. 3, pp. 57-72, 2008. Article (CrossRef Link)

[42] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, “Noxes : A Client-Side Solution for Mitigating Cross-Site Scripting Attacks,” in *Proc. of 2006 ACM Symp. Appl. Comput. (SAC 2006)*, pp. 330–337, Dijon, France, 2006. Article (CrossRef Link)

[43] C. Yue and H. Wang, “Characterizing Insecure JavaScript Practices on the Web,” in *Proc of the 18th International Conference on World Wide Web, (WWW 2019)*, Madrid, Spain, 2pp. 961–970, 2009. Article (CrossRef Link)

[44] K. Mcquade, “Open Source Web Vulnerability Scanners : The Cost Effective Choice ?,” in *Proc. of Conf. Inf. Secur. Appl. Res.*, Baltimore, Maryland, USA, vol. 2014, pp. 1–13, 2014. Article (CrossRef Link)

[45] M. Volpi, “How open-source software took over the world | TechCrunch,” 2019. [Online]. Available: https://techcrunch.com/2019/01/12/how-open-source-software-took-over-the-world/, Accessed on: May 05, 2020

[46] G. Hill, “Comparison of Automated XSS Fuzzing & Injection Tools,” Abertay University.

[47] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda, “An empirical analysis of input validation mechanisms in web applications and languages,” in *Proc. of 27th Annu. ACM Symp. Appl. Comput. (SAC 2012)*, Trento, Italy, pp. 1419–1426, 2012. Article (CrossRef Link)

[48] J. Fonseca, N. Seixas, M. Vieira, and H. Madeira, “Analysis of Field Data on Web Security Vulnerabilities,” *IEEE Trans. Dependable Secur. Comput.*, vol. 11, no. 2, pp. 89–100, 2014. Article (CrossRef Link)

Nurul Atiqah Abu Talib received the BIT (Hons) in Computer System Security from Universiti Kuala Lumpur, Malaysian Institute of Information Technology (MIIT), Malaysia, in 2013. She is currently a Ph.D candidate in the Department of Computer Science and Engineering at Hanyang University ERICA, Gyeonggi-do, South Korea. Her research interests include web security, machine learning and program analysis.

Kyung-Goo Doh (corresponding author) received the B.S. degree in industrial engineering from Hanyang University in 1980, the M.S. degree in computer science from Iowa State University in 1987, and the Ph.D. degree in computer science from Kansas State University in 1992. From 1993 to 1995, he was with the University of Aizu as an assistant professor. He then joined the Department of Computer Science at Hanyang University ERICA, where now is a professor. His primary research interests are programming languages, program analysis, software engineering and software security.