Pertanika J. Sci. & Technol. 31 (3): 1353 - 1377 (2023)

PERTANIKA JOURNALS

SCIENCE & TECHNOLOGY

Journal homepage: http://www.pertanika.upm.edu.my/

Server-Side Cross-Site Scripting Detection Powered by HTML Semantic Parsing Inspired by XSS Auditor

Chrisando Ryan Pardomuan[1,2]*, Aditya Kurniawan[1], Mohamad Yusof Darus[3], Muhammad Azizi Mohd Ariffin[3] and Yohan Muliono[1]

[1]School of Computer Science, Bina Nusantara University, Kota Jakarta Barat, Daerah Khusus Ibukota, Jakarta, 11530, Indonesia
[2]Computer Science Department, Bina Nusantara University, Jakarta, 11530, Indonesia
[3]Faculty of Computer and Mathematical Sciences, UiTM, Shah Alam, 40450, Malaysia

ABSTRACT

Cross-site Scripting attacks have been a perennial threat to web applications for many years. Conventional practices to prevent cross-site scripting attacks revolve around secure programming and client-side prevention techniques. However, client-side preventions are still prone to bypasses as the inspection is done on the user's browser, so an adversary can alter the inspection algorithm to come up with the bypasses or even manipulate the victim to turn off the security measures. This decreases the effectiveness of the protection and leads to many web applications are still vulnerable to cross-site scripting attacks. We believe that XSS Auditor, which was pre-installed in Google Chrome browser for more than 9 years, is a great approach in combating and preventing XSS attacks. Hence, in this paper, we proposed a novel approach to thoroughly identify two types of cross-site scripting attacks through server-side filter implementation. Our proposed approach follows the original XSS Auditor mechanism implemented in Google Chrome. However, instead of placing the detection system on the client side, we design a detection mechanism that checks HTTP requests and responses as well as database responses for possible XSS attacks from the server side. From 500 payloads used to evaluate the proposed method, 442 payloads were classified correctly, thus showing that the proposed method was able to reach 88.4% accuracy. This work showed that the proposed approach is very promising in protecting users from devastating Cross-site Scripting attacks.

Keywords: Cross-site scripting, injection attack, server-side detection, web application security

ARTICLE INFO

Article history:
Received: 08 February 2022
Accepted: 24 May 2022
Published: 31 March 2023

DOI: https://doi.org/10.47836/pjst.31.3.14

E-mail addresses:
chrisando.pardomuan@binus.edu (Chrisando Ryan Pardomuan)
adkurniawan@binus.edu (Aditya Kurniawan)
yusof@tmsk.uitm.edu.my (Mohamad Yusof Darus)
mazizi@fskm.uitm.edu.my (Muhammad Azizi Mohd Ariffin)
ymuliono@binus.edu (Yohan Muliono)
*Corresponding author

ISSN: 0128-7680
e-ISSN: 2231-8526

INTRODUCTION

Cross-site Scripting (XSS) is one of the only vulnerabilities in OWASP Top 10 (Wichers & William, 2017) four times in a row—2007, 2010, 2013, and 2017. Moreover, Cross-site Scripting could lead to a more devastating impact, such as obtaining personal information or stealing user cookies to hijack their identity in a fraudulent session, allowing attackers to steal sensitive data or even take control of certain devices (Takahashi et al., 2013). As the adoption of JavaScript spreads through the Internet and more UI libraries (e.g., React, Vue, Angular, among others) are developed and experienced significant growth, the stake and devastating impact that an adversary can execute malicious JavaScript on a user's browser environment is exponentially increasing to the degree that has not been seen before. Although some preventive action has been taken, fixing all vulnerabilities and eradicating Cross-site Scripting attacks by implementing defensive coding techniques on every website is undeniably difficult.

Some research has been proposed to detect Cross-site Scripting attacks occurring in a browser by placing the filtering engine on the browser to search through the HTML string for any occurrence of the input, such as the IE8 Filter implemented in Internet Explorer (Swiat, 2008) that unfortunately achieves low reliability due to its inability to detect partial scripting detection and has no context-aware sanitation (Sarmah et al., 2018). There is also a method called NoScript (https://noscript.net/) filter that analyzes outgoing requests from the browser for potentially malicious XSS payload occurrences; it often leads to high *false positive* rates because there is no way to verify whether the suspicious scripts appear in the HTML document response (Sarmah et al., 2018). With the intention to solve the issues, a more sophisticated method has also been proposed by Bates et al. (2010). Instead, they placed the filtering engine just after the browser's HTML parser to reduce false positivity in the detection result. In their research, Bates et al. (2010) argue that placing the filter after the browser's HTML parser is considered to have a complete interposition in the browser with great performance and high fidelity. Their research shows that by examining the response just after parsing, the filter can examine the semantics of the response, as interpreted by the browser, with more efficiency and error-prone processing.

Furthermore, the proposed method has been implemented first in WebKit and later in Google Chrome, publicly known as the XSS Auditor (The Chromium Projects, 2019) for almost 10 years. Although Google eventually decided to remove XSS Auditor from their Google Chrome browser by late 2020, we believe the originally proposed method is still quite effective in combating Cross-site Scripting attacks. The only caveat we think can be improved is that XSS Auditor, among many other common XSS detection approaches, is designed to be a broad-spectrum detection system that focuses on detecting probable attacks across almost every website opened by the browser. It eliminates the capability of the detection system to protect the user from Stored XSS attacks, only the Reflected and

DOM-based XSS. In Stored XSS attacks, the malicious payload comes inside the database instead of directly from the user's HTTP request. Nevertheless, as the client-side detection system is always placed on the users' browsers, XSS payloads stored inside the database will already be inserted into the HTML document when the browser receives and renders it. Hence the attack cannot be detected by client-side approaches. Moreover, Stored XSS attacks can lead to more devastating damage because while Reflected XSS attacks often limit the scope of the attack to the attacker (i.e., the outcome of the attack is *reflected* by whoever injected the payload), Stored XSS attacks allow the payload injected by an attacker to be executed multiple times and even affecting on the unsuspicious user's browser context because of its nature the payload is *persistently* stored inside the application's database (Cui et al., 2020).

Hence, we propose a detection mechanism by adopting the original XSS Auditor mechanism to detect Cross-site Scripting attempts on the server side, not just Reflected Cross-site Scripting attacks but Stored Cross-site Scripting. Instead of examining an HTTP request and its related HTTP response, we propose that the third element should include the database result when a certain request is processed on the server. Examining the database's response in the same way the XSS Auditor examines the HTTP request makes the proposed method detect most of the Stored Cross-site Scripting attacks. The remainder of this paper is organized as follows. First, we will briefly review three types of Cross-site Scripting (XSS) attacks and the original concept of XSS Auditor proposed by Bates, Barth & Jackson (2010). Then, we will proceed by elaborating an in-depth explanation of our proposed method: the alternative to detect two of the three types of XSS attacks through server-side filtering based on the XSS Auditor mechanism. Finally, the evaluation process and the result will be presented to measure the performance of the proposed method in detecting Cross-site Scripting (XSS) attacks.

MATERIAL

Related Works

Cross-Site Scripting. Cross-site Scripting (XSS) exploits are like most code-based injection attacks, similar to SQL Injection, which injects an arbitrary code into an application so that the application executes the arbitrary code on behalf of the attacker (Rodriguez et al., 2020). In this context, the injected code on XSS attacks usually is JavaScript code or HTML tags that contain JavaScript code.This attack exploits output functions (e.g., browser render process) in an application that references or processes incorrectly sanitized user input results. An XSS attack's basic idea is to use special characters that cause the browser to change the browser's interpretation of a document or input from context as data into context as program code (Liu et al., 2019).For example, when an HTML page references user input as data, an attacker can include an HTML <script> tag that can activate the

JavaScript interpreter in the browser and execute the arbitrary JavaScript code put by the attacker inside the tag. When an attacker manages to do that, and the injected code is executed in the client browser, it can result in more nefarious activities such as stealing cookies, defacing the interface of the website, or even unauthorized request being sent from the victim's browser by unbeknown to the victim (Liu et al., 2019). XSS attacks can be divided into Reflected XSS, Stored XSS, and DOM-based XSS. The characteristics of each attack are the following:

1. Reflected cross-site scripting
   This type of XSS occurs when a dangerous string that causes XSS comes from inputting user data on HTTP requests. This type of XSS is usually found in a web application's error messages and search results (Shar & Tan, 2011). In addition, this type of attack can generally occur due to input sanitation errors on users on the server side (server-side fault). In some cases, interaction by the victim is required for the arbitrary JavaScript code to be executed, for example, through a spoofing attack. In a spoofing attack, the attacker must trick the victim into clicking somewhere on the website to trigger the JavaScript code execution (Rodriguez et al., 2020).
2. Stored cross-site scripting
   This type of XSS occurs when a web application stores a dangerous string from user input on the database, and then the target application's HTML web document references the harmful input. Attacks on social networking sites are generally this type of XSS attack (Shar & Tan, 2011). As Reflected in XSS, this type of attack also occurs due to input sanitation errors on the server side (server-side flaws).
3. DOM-based cross-site scripting
   This type of XSS attack focuses on manipulating the environment and JavaScript program code (Gan et al., 2020), the HTTP response from the original website to the victim does not change, but the client-side script runs unlike it should because the DOM environment of the page has been manipulated by the attacker (Shar & Tan, 2011). This type does not exploit web application server vulnerabilities.

XSS Auditor. XSS Auditor is a rather novel XSS detection method that Bates et al. (2010) proposed. It has been implemented in Google Chrome (and another Chromium-based browser) since 2010 with the release of Google Chrome v4 (The Chromium Projects, 2019). Unfortunately, Google decided to disable and remove XSS Auditor from their browser in 2019. Before the method is proposed, most XSS detection engine relies on examining payload occurrences inside an HTTP response *before* the browser processes the response. According to Bates et al. (2010), those method leads to lower filter precision as it processes the syntax of the response, not its semantics. Also, searching for malicious

payload occurrences inside the HTML response with regular expressions tends to produce un- necessary false positives. In Chrome's XSS Auditor, however, the filter is placed strategically between the HTML parser and the JavaScript engine. Placing the filter in such a position has at least three advantages (Bates et al., 2010):

1. High performance

   As the filter simply receives the HTML response already parsed (or interpreted) by the browser intoa DOM-tree, it does not need the extra time to performa time-consuming, error-prone simulation that usually occurs when a filter performs the parsing itself. Also,the filter can block the execution of the XSS payloads safely instead of forcefully modifying the pre-parsed stream to mangle the injected payload, which reduces the resources and processes taken to mitigate the attack.

2. High precision

   Moreover, since the DOM-tree to be examined by the filter is coming from the browser itself, false positives that usually occur because of a different DOM-tree interpretation between the client's browser and the XSS filter of the same HTML response can be greatly reduced or eliminated.

3. Complete interposition

   Additionally, placing the filter in front of the JavaScript engine lets the filter completely interpose all elements and tags that will be treated as JavaScript instructions by the browser. If the filter detects a malicious payload and wants to block it, it can simply withhold the payload from the JavaScript engine, and the malicious payload will not be executed.

In the actual implementation in WebKit, the filter mediates between the HTML parser engine inside the WebCore component and the JavaScript engine inside the JavaScriptCore component. To perform the inspection, the filter intercepts and examines every attempt to run inline scripts, inline event handlers, and JavaScript URLs. In addition, the filter also intercepts and examines the loading of external scripts and plugins, such as using document. write to write arbitrary script tags to the HTML document itself (which we believe also works as a bypass trick for other XSS filters).

Before searching for malicious payloads in the HTTP request sent by the browser on behalf of the user, it must transform the URL request and any POST data to mimic how the browser sees them (Satish & Chavan, 2017). The transformation includes URL decode, Character set decode, and HTML entity decode. Although the transformation process is involved, the filter does not have to simulate the complex, resource-expensive process of the parser, such as tokenization and element re-parenting. Instead, after the transformation has been done, the filter merely performs a matching algorithm between what can be found

in the transformed HTTP request (including the POST data) and the HTML elements that are to be passed to the JavaScript engine that is intercepted from the parsing engine. If the filter found any match, it means that some (or all) part of the user input is reflected on the element that will be executed by the JavaScript engine, thus indicating a Cross-site Scripting attack. When that happens, the filter refuses to deliver the element to the JavaScript engine.

Server-Side XSS Detection System. Many other proposed methods have put the XSS detection system on the server side instead of the client-side (i.e., browser). For example, Kazal and Hussain (2021) designed a server-side detection and prevention system specifically for Stored (persistent) XSS attacks. The proposed method works by checking and sanitizing the user's input before storing it in the database, preventing the XSS payload from the very beginning of the attack. The input sanitation is conducted using *harmlist*, a list of dangerous keywords and substrings that can be used for XSS attacks and configured by the system's administrators. Their evaluation process was measured on Damn Vulnerable Web Application (DVWA) application, and the proposed method could detect the attempted attacks with staggering accuracy.

Meanwhile, Abaimov and Bianchi (2019) proposed a deep-learning (CNN) model to detect malicious code-injection queries such as SQL Injection and XSS attacks. The dataset used to train and evaluate the model is taken from an online publicly available common code-injection payload list. One of the proposed approach's main novelties lies in the pre-processing module, which the authors argue helps improve the detection rate and accelerate the training process by enriching the dataset with semantic labels and filtering out *noisy* and ambiguous information from the payloads. The detection system is installed on the server-side, but instead of capturing and processing raw data directly from HTTP request, the system converts the payload into an encoded pattern to remove the randomness of the payload and help the model identify the significance of each symbol found on the payload. The model's performance was measured using a *confusion matrix* (Accuracy, Precision, and Recall) and reported achieving 94% accuracy, 99% precision, and 93% recall value in detecting SQL Injection and XSS attack payloads.

Both methods propose a rather sophisticated detection system that prevents the attack, including Stored XSS attacks before the attack even begins (i.e., when the moment the user's payload is to be received and processed) by sanitizing or analyzing the inputs received. However, most of the existing web applications might already have suffered the attack prior to installing the detection system, and the XSS payloads might already be inside their database. Hence, even if the detection system is placed on the server-side to enable Stored XSS detection capability, they will have difficulty detecting and removing threats from XSS payloads already stored inside the database because the content of the database is not inspected or analyzed by the proposed systems.

METHODOLOGY

This section describes the design and implementation of a server-side XSS detection system based on Google Chrome's original XSS Auditor algorithm (Figure 1). The system is designed to perform pattern-matching analysis of HTML responses in the form of an HTML document tree (DOM-tree) instead of analyzing raw string representation of HTML responses that the server received; it aims so that the proposed method checks on the HTML elements that are similar to what the browser sees and render, right before the JavaScript code run by the JavaScript engine. By examining the response after HTML parsing, the detection system can easily identify which part of the response is being treated as a JavaScript instruction. Instead of running regular expressions over the bytes that comprise the response, the detection system examines the DOM-tree created by the parser, making the semantics of those bytes clear similar to how Google Chrome's XSS Auditor works.

Pre-Processing

Before the system looks for signs of malicious JavaScript in both the HTTP request and the Database response and compares them with the content of the HTTP response, the input data (HTTP request or Database response) needs to be transformed as follows:

1. URL Decode
   The input from HTTP requests is often encoded; hence we need to decode them to avoid misinterpretation of the data and detect bypass attempts.
   For example, %41 will be changed to the character 'A'.
2. Character-set Decode
   Should the input is encoded using the uncommon encoding (e.g., UTF-7, among others.), we convert the input to Unicode encoding to ensure that the user's browser and our detection system see the input in the same format.
   For example, UTF-7 encoding is transformed into Uni-code characters.
3. HTML Entity Decode
   The input from HTTP requests is also often encoded into HTML entities (e.g., & < &gt, among others.). Therefore, we also need to decode them to avoid misinterpretation of the data and detect bypass attempts.
   For example: & is transformed into '&'.

The transformation is necessary because the server-side detection system must mimic what the browser does (Satish & Chavan, 2017) to reduce and even prevent the different interpretations between them (the client's browser and the server). We did not design the pre-processing step to be run during the development of the system, but instead is embedded in the system and runs imminently every time an HTTP request or data from

the database is about to be put into the HTML document inside the HTTP response. Table 1 describes every attribute and data source we considered a probable place for the XSS payload that is the subject of this pre-processing step.

Table 1

Attributes for pre-processing

No	Attribute	Source	Example
1	Query Param	HTTP Request (URL)	?search=<script>alert(1)</script>
2	HTTP Body	HTTP Request (Body)	username=<script>alert(1)</script>
3	Cookie	HTTP Request (Header)	SESSION=<script>alert(1)</script>
4	User Agent	HTTP Request (Header)	User-Agent: <script>alert(1)</script>
5	Values from Database	Database	note_name: "<script>alert(1)</script>"

After the inputs are pre-processing, the HTTP response is passed into the system's HTML parser to be converted into an HTMLDOM-tree for XSS checking before it is sent back to the client.

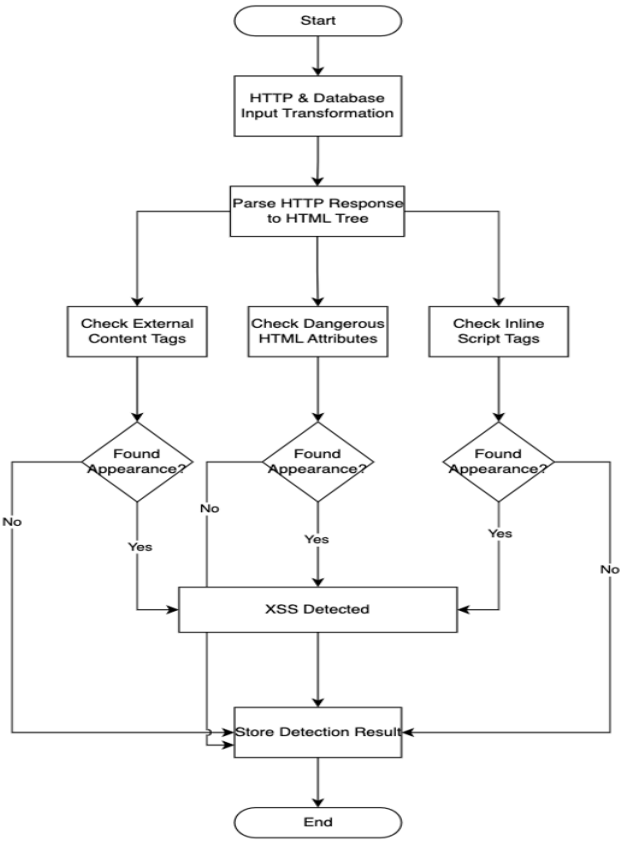

Figure 1. Cross-site scripting detection methodology

Detection Algorithm

According to Stock et al. (2014), the detection process in the original XSS Auditor (from Google Chrome) is designed to analyze every single tag from the HTML DOM-tree and check whether the tag or some of its attributes are also contained in the HTTP request. Our detection system does the same, only it checks and compares the tag with the database response. If the detection system finds a tag (or its attributes) from the HTML DOM-tree contained inthe HTTP request or the Database response, it will consider the tag malicious, and the HTTP request will be marked as a malicious request. In this proposed approach, the detection system assumes every data from the HTTP request (*direct* from user input) and the Database response (*indirect* from user input) as the same input source. Thus, no specific differentiation is applied between the two sources. The examined tag criteria are as follows:

1. Inline script tags
   As the major injection point for most cross-site scriptingattacks, the detection system examines every inline script tag from the document tree.
   For example, <**script**>**alert(1)**</**script**>
2. Dangerous HTML attributes
   The detection system checks for attributes in every HTML tag that are known can be used for JavaScript execution, as can be seen in Table 1.
   For example, <**img src=”profile.jpg” onload=’javascript:alert(1)’**> that allows the attacker to execute JavaScript by injectingthe payload inside an *onload* attribute.
3. External content tags
   The detection system also checks the possibility of the attacker trying to smuggle malicious JavaScript code from external content (e.g., malicious payload placed on arbitrary websites). Thus, the auditor also checks for every script tag that contains *the src* attribute.
   For example, <**script foo=bar src=”http://hacker.com/evil.js”**\>

Figure 2 illustrates how our detection system examines HTTP packets to find any occurrences of arbitrary XSS payload.

When the server receives an HTTP request packet, our detection system inspects the packet and extracts user-inputted values in the request. As we can see in the above picture (right-side), inside the query parameters alone, there are two pieces of information supplied by the user: the username and the search parameters. Furthermore, the arbitrary XSS payload is located inside the search parameter. Comparing the data with the HTTP response (left-side), we can see that the input (<script>alert(1)<script>) is reflected on the HTML response from the server. It is a Reflected Cross-site Scripting attack. In general, for

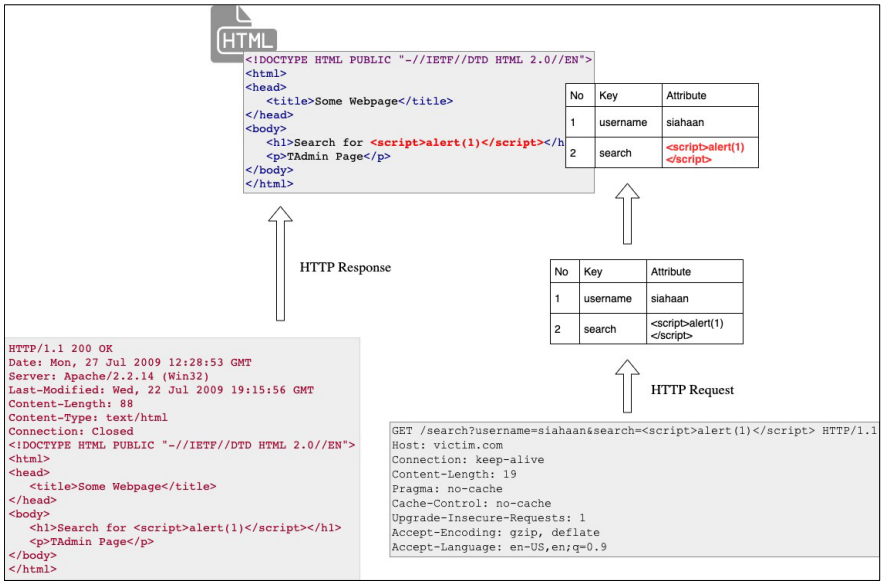

Figure 2. Payload inspection illustration

our detection system to detect the attack, first, it examines the HTTP request and extracts every possible user input from the request (request body, query parameters, among others). Then, when the web server is about to respond to the request with an HTTP response containing the HTML string, our detection system will intercept the response and parse the HTML string into the HTML DOM tree. After that, for all elements and tags that satisfy the examination criteria above, our detection system will check whether the element and/or its content can be found inside the collected user inputs. If so, the detection system will mark the request as a request with a potential Cross-site Scripting attack and either halt the server's response or simply log the event for further examination.

Table 2

HTML attributes that are considered dangerous

No	HTML Attribute	Description	Example
1	href	Specifies the URL of a page the link goes to	<... href="http://web.com/" />
2	src	Specifies the URL of an external (or internal) resource	<... src="run.js" />
3	content	Store the value associated with the http-equiv or name attribute	<... content="some value" />

Table 2 (Continue)

No	HTML Attribute	Description	Example
4	Data Attributes (data-*)	Store an arbitrary data private to the page on all HTML elements	<... data-price=”2000” />
5	Event Attributes (on*)	Let an event triggers action in a browser	<... onerror=someFunctionName />

The detection system utilizes the *source-destination* information of the network packet. For example, when an HTTP request arrives at the server, it contains information about where it came from (called *source*) and where the packet should be delivered to (called *destination*). Later, when the server had finished processing the request, an HTTP response packet was sent back to the client. The packet also contains information about where it came from (*source*) and where it will be delivered to (*destination*), but the *source* and *destination* are inverted compared to the *source* and *destination* in the request package. Thus, if the *source-destination* combination in the HTTP requests is 1234-5678, the HTTP response will have 5678-1234 as the *source-destination* combination.

The detection system also considers the database response so that the detection system can detect more than just a Reflected Cross-site Scripting attack, as designed in the original XSS Auditor paper (Bates et al., 2010), but a Stored Cross-site Scripting attack as well. A Stored Cross-site Scripting attack indicates that the payload will not just reflect on the response of the HTTP request but also be stored persistently on the server (e.g., in the database). Hence, to examine the HTTP response where the user input is not from the HTTP request but from the Database response, our detection system needs to identify which Database response is an HTTP response. It is impossible to use the source-destination combination like in the previous because although almost every database could communicate via TPC/IP protocol, we would need to put two separate listeners: one between the network and the web server and the other between the web server and the Database. That way, we discover that it is not possible to correlate the source-destination combination of the HTTP request to the web server and the source-destination combination of the web server to the database. Therefore, using a timestamp the detection system is designed to correlate the HTTP request and the database response.

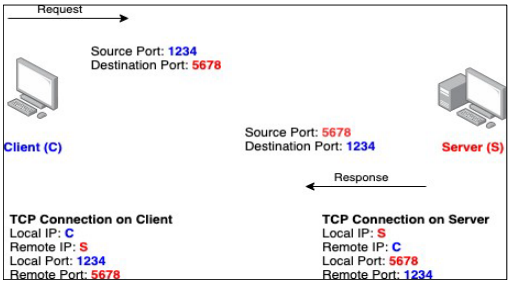

Figure 3. HTTP request and response matching

Nevertheless, we encounter that the implementation of such a method could lead to at least two problems, mainly because the time taken to fetch the data from the database between each HTTP request varies; it can be 0 ms to a few seconds, depending on how big the data and the load of the database server itself.

First problem: we designed the system to record the timestamp of both events, the arrival of either the HTTP request or the Database response, and the completion of the request after being processed by the web server that ends with an HTTP response being sent back to the client. The timestamps of the two events can differ by a few milliseconds, so our detection system needs to ensure that both are synchronized. Timestamp synchronization is important for two reasons: the atomicity of the report and the effectiveness of the correlation process.

Second problem: there may be several (if not many) HTTP request or Database response that arrives at the very same moment, which leads to many request-response elements having the same timestamp record. It can confuse the examination process. The detection system might be unable to differentiate one HTTP response that belongs to an HTTP request from the other responses if the timestamp is similar; hence, the detection system yields low performance and low detection accuracy.

The detection system calculates the average time between the arrival of the HTTP request in the server and the arrival of the Database response to solve both problems, as seen in Equation 1.

$$\bar{T} = \frac{1}{n}\sum_{T=0}^{n} T \tag{1}$$

The detection system calculates the delay between the server receiving an HTTP request until the HTTP response is ready to be sent back to the client after being processed by the server and stored along with the HTTP request data into a structured dataset. Then, when the detection system detects a Database response being sent back to the web server, it calculates the average delay time (Equation 1) and takes every HTTP request that arrived T time before and after the timestamp of the Database response from the structured dataset, where T is the calculated average time. Finally, the detection system retrieves every HTTP request within the scope of the average timestamp constraint and looks for any partial or full occurrences from both input sources (HTTP request or Database response) in the HTTP response. It will take more computing power, but the detection accuracy will be significantly higher.

Implementation

The system needs to intercept traffic from HTTP and the database to ensure the detection system can retrieve data from the user's input and the database. Hence, the implementation

of server-side XSS Auditor in this research consists of two modules: the HTTP Sniffer module and the server-side XSS Auditor itself. HTTP Sniffer module intercepts every HTTP network packet, both Request and Response after it arrives at the server and before it is sent back to the client. The intercepted packet is stored in Redis in-memory storage to support seamless data retrieval. When an HTTP response is ready to be sent back to the client, XSS Auditor takes the HTTP request data from Redis and performs the XSS detection process. If the attack indication is found, the HTTP response packet willbe tainted, or the detection system will generate log information about the attack. Figure 4 illustrates the design of the system proposedin this paper. Both modules are developed using Python, and the HTTP interception capability is possible using the Scapy library. Meanwhile, the database interception capability is made possible with the help of Packetbeat by Elasticsearch[1]. When the server's database sends a query result back to the web application, Packetbeat will intercept them and forward the resulting rows to the HTTP Sniffer module for malicious payload checking. Communication between HTTP Sniffer and Packetbeat is done through a socket-based connection so that they do not occupy any network port while at the same time protecting the confidentiality and integrity of the database by making it harder for perpetrators to intercept and peek at the transferred information.

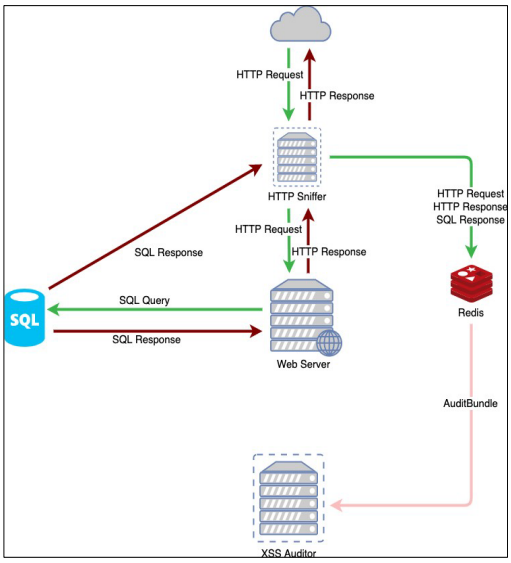

Figure 4. Server-side cross-site scripting detection architecture

HTTP Sniffer

This module is responsible for capturing HTTP packet traffic in the server. The module parses and extracts various attributes from the HTTP packet relevant to Cross-site Scripting detection (Table 2). There are two types of HTTP packets, the HTTP request packet and the HTTP response packet. When the captured packet is an HTTP request packet, the module takes note of the timestamp and stores the data in Redis along with the timestamp information. In addition to the request packet, when the module receives a Database response packet from the database server through Packetbeat, the module will try to correlate the packet with a single HTTP request that is already stored in Redis according to their timestamp difference; by retrieving the HTTP request from Redis, combined it with the Database

1 https://www.elastic.co/beats/packetbeat

response, and then stored it back to the Redis (Figure 6). List of data and information captured from the HTTP can be seen on Table 3. Then, both data are combined and stored in Redis. Finally, when the module captures an HTTP response packet, it retrieves the HTTP request packets and (if any) the Database response from Redis and sends the combined data to the XSS Auditor module for inspection. Also, both storing and retrieving operation inside Redis is done using RedisTimeSeries (https://oss.redislabs.com/redistimeseries/) data structure available on Redis as a third-party module. RedisTimeSeries are used for two reasons: to preserve efficiency and speed when computing and retrieving time data and to make Redis able to store time data effectively.

Table 3

Obtained information from HTTP request packet

No	Attribute	Location	Purpose
1	Cookie	HTTP Header	Potential XSS payload location
2	Host	HTTP Header	Information about the attack
3	Request Body	HTTP Body	Potential XSS payload location
4	Query Param	HTTP Header	Potential XSS payload location
5	Content-type	HTTP Header	Limit checking to text/html content only
6	Request Method	HTTP Header	Information about the attack

Every data collected by the HTTP Sniffer module: the HTTP request, Database response (if any), and the HTTP response, as we called it AuditBundle data to be short, has a unique identifier when stored in Redis. This unique identifier, generated based on the timestamp of arrival as a UUID string, helps this module and every other module find the connection between an HTTP request, a Database response, and an HTTP response. While the raw data is stored inside Redis, the detection system also stores a combination of the uniquely generated packet identifier (named BundleID) and its timestamp of arrival (T) in the form of time series data inside RedisTimeSeries (Figure 5).

Algorithm 1: *AuditBundle* Data Storing in Redis

$httpPacket \longleftarrow$ The Captured Packet;
$T \longleftarrow CurrentTimestamp$;
$BundleID \longleftarrow uuid()$;
$storeIndexToTimeSeries(BundleID, T)$;
$storeAuditBundle(httpPacket, BundleID)$;

Figure 5. Storing AuditBundle to Redis and RedisTimeSeries

A similar concept also applies to data retrieval. When this module needs to retrieve the previously stored data from Redis, it performs a TimeSeries query and retrieves every BundleID that has a timestamp between $\bar{T}$ time before and after the current timestamp (the time when the detection system captures the HTTP response) (Figure 6), where $\bar{T}$ is the calculated average response time of the web server (1). It is

important to know that the TimeSeries query might return more than one BundleID at that time range, depending on how much the traffic load is experienced by the server. After that, for every retrieved BundleID from the query, the module will retrieve the AuditBundle accordingly. Every AuditBundle retrieved will be passed into the XSS Auditor module for a thorough inspection. In the optimal condition, an AuditBundle will contains an HTTP request, a database response, and an HTTP response (Table 3).

Algorithm 2: *AuditBundle* Data Retrieval from Redis

$T \longleftarrow CurrentTimestamp$;
$\Delta T \longleftarrow getAverageResponseTime()$;
for $t \in \{T - \bar{T}, \ldots, T + \bar{T}\}$ **do**
  $BundleID \leftarrow getIndicesFromTimeSeries(t)$;
  $httpPacket \leftarrow getAuditBundle(BundleID)$;
  $performXSSDetection(httpPacket)$;
end for

Figure 6. Retrieving AuditBundle from Redis

Table 3

Obtained information from HTTP request packet

No	Attribute	Location	Purpose
1	Cookie	HTTP Header	Potential XSS payload location
2	Host	HTTP Header	Information about the attack
3	Request Body	HTTP Body	Potential XSS payload location
4	Query Param	HTTP Header	Potential XSS payload location
5	Content-type	HTTP Header	Limit checking to text/html content only
6	Request Method	HTTP Header	Information about the attack

XSS Auditor

This module has two main responsibilities: to manage the data (AuditBundle data) required for the XSS inspection from the HTTP Sniffer module and to perform the inspection itself. Therefore, we designed this module to have two inspection types: Light and Deep Inspections.

Light Inspection is designed so that the XSS Auditor module does not have to wait for a Database response to perform an XSS detection since some requests do not require database interaction (e.g., a simple visit GET request). Light Inspection occurs the moment the HTTP Sniffer module captures an HTTP response. This module then receives an AuditBundle that contains the related HTTP request packet and HTTP response packet from the HTTP Sniffer module and performs Reflected Cross-site Scripting detection. Since Light Inspection does not wait for the Database response, only user input in the HTTP request (e.g., Query Parameter on POST Body) will be checked. On the other side, the Deep Inspection is designed to run the XSS inspection only after the HTTP Sniffer module intercepts a database response. When performing Deep Inspection, this module receives the HTTP request, the Database response information, and the HTTP response, all compiled into an AuditBundle data from the HTTP Sniffer module. With these data, the XSS Auditor module performs Stored and

Reflected Cross-site Scripting detection using the algorithm comprehensively described in the previous section (see Detection Algorithm section).

Finally, suppose the detection system found any occurrences of the evaluated HTML component (see Detection Algorithm section) on the input (HTTP request, Database response, or both). In that case, the detection system taints the response by injecting a custom HTTP header X-XSS-Detected to the HTTP response so that the client-side code or the browser can invalidate the response and prevent the payload from being rendered. In this version of the research, we designed the detection system only to detect attacks, not to prevent attacks. Therefore, when the detection system detects an attack, it logs all information about the request and stores it in a centralized log database for human verification.

RESULT AND DISCUSSION

In this section, we evaluate the accuracy of our server-side Cross-site Scripting detection engine. The accuracy will be measured using a confusion matrix because the reliability of our detection system depends on how many payloads the detection system can detect, shown by the rate of false positives and false negatives (Bates et al., 2010). The evaluation was carried out on two Cross-site Scripting attacks: Reflected Cross-site Scripting and Stored Cross-site Scripting. An attack condition will be made by simulating an HTTP request to the server where our detection system is installed and active. The application that becomes the target of exploitation is Damn Vulnerable Web Application (DVWA), a vulnerable-by-design web application developed for security testing and experiments. The attack simulation consists of 499 HTTP requests, divided into 239 malicious payloads taken from XSS Payload List in GitHub (https://github.com/pgaijin66/XSS-Payloads) and 261 benign payloads taken from HTTP CSIC Dataset 2010 (Giménez et al., 2010). An example of the evaluation data can be seen in Table 4. Note that for the malicious payloads, since some of the payloads listed on the XSS Payload List might not be working on Google Chrome due to some circumstances (e.g., the payload is specific for another browser), we filtered and curated the payload into 239 payloads that can be verified are working in Google Chrome browser.

Table 4

Example of evaluation dataset

Nature	Payload	Source
Benign	maria-terzon@lingotes.com.org	HTTP CSIC Dataset
Benign	<x src=”alert(1)”>IMPORTANT<x>	Custom List
Malicious	<meta http-equiv="refresh" content="0;url=javascript:confirm(1)">	XSS Payload List
Malicious	<script>alert(“XSS”);</script>	XSS Payload List
Malicious	<form><isindex formaction="javascript:confirm(1)"	XSS Payload List

The reason behind the use of those data sources is that HTTP CSIC Dataset 2010 is widely used for training models that can identify dangerous payloads (Yavanoglu & Aydos, 2017; Vartouni et al., 2018); while XSS Payload list provides various types of payloads with encoding, obfuscation, and detection evasion properties that are similar to actual conditions. Furthermore, there are two types of attack scenarios that we set up for the evaluation:

1. Direct-input attack
   In this scenario, the source of the attack comes directly from the user's input (i.e., HTTP request), both on Stored XSS and Reflected XSS attack attempts. Thus, no information from the database is involved in analyzing and detecting the attack. This scenario evaluates our detection system's capability of Light Inspection (see Detection Algorithm section).

2. Persisted attack
   In this scenario, the attack's source originates from data already stored inside the database. That means we first make sure that the XSS payloads are inputted and stored with no alert or prevention from every detection system intentionally, but when those payloads are retrieved from the database into the DVWA page, the detection systems will try to detect the attack. This scenario evaluates our detection system's Deep Inspection (see Detection Algorithm section) capability, where information from the database is analyzed to detect potential attacks.

As a benchmark, we compare the performance of our detection system with a server-side wide-range attack detection system named PHPIDS and the XSS Auditor itself, a client-side XSS attack detection system. Table 5 shows the comparison between our proposed method and the benchmark methods. Both attack scenario above is also performed on the PHPIDS and the XSS Auditor, and the confusion matrix will be measured accordingly. Since XSS Auditor is unavailable on the more recent version of Google Chrome, the evaluation process will use Google Chrome version 70.0.3538.77. Because XSS Auditor is a client-side system, to determine whether it detects the attempted XSS attack, we set up a reporting URL endpoint and tell the XSS Auditor to report the detected attack into the reporting URL using "X-XSS-Protection header=1; report=http://ourservice.com/report". Any attempted attack detected by the XSS Auditor can be collected and verified. As for PHPIDS and our detection method, we rely on the detection result to consider if the attacks have been detected. Should the detection methods classify the payload as malicious, since we have verified that every payload inside the malicious list is working on the Google Chrome browser, the detection result can be considered reliable, and the payload is truly malicious.

Table 5

Comparison between ours and benchmark methods

Comparisons	Our Proposed Method	PHPIDS	XSS Auditor
Location	Server-side	Server-side	Client-side
Scope of Scanning	Database Data, HTTP Request, HTTP Response	HTTP Request, HTTP Response	HTTP Request, HTTP Response
Supported Web	Any web application	PHP-based Web Application	Any web application that a browser can render
Type of XSS attack detected	Stored XSS, Reflected XSS	Reflected XSS	Reflected XSS

Attack Result Comparison

The results obtained by each detection system after the attack simulation can be found in Tables 6 and 7.

Table 6

Attack simulation result

Metrics	Our Detection System		PHPIDS		XSS Auditor	
Scenario	Direct-input	**Persisted Attack**	Direct-input	Persisted Attack	**Direct-input**	Persisted Attack
TP	180	**192**	225	0	**239**	0
FP	0	**0**	0	0	**0**	0
TN	261	**261**	261	261	**261**	261
FN	59	**47**	14	239	**0**	239

For the Direct-input scenario, our analysis of the result is as follows. From 239 malicious payloads, our detection system detected 180 as potential Cross-site Scripting payloads (*true positive*), while 59 failed to be identified (*false negative*). Meanwhile, from 261 benign payloads, our detection system successfully identified 261 requests, or all requests, as harmless (*true negative*), leaving 0 payloads misclassified (*false positive*). On the other hand, the same attack payload is replayed to the DVWA application on different servers with two security countermeasures: (1) PHPIDS and (2) XSS Auditor, as previously mentioned. PHPIDS correctly identified 225 malicious payloads (*true positive*), leaving 14 payloads classified falsely (*false negative*). For the benign samples, PHPIDS could correctly identify all requests as not malicious (*true negative*), also leaving zero payloads

misclassified. Furthermore, XSS Auditor on Google Chrome could correctly detect 239 malicious XSS payloads, leaving 0 payloads that failed to be identified. Furthermore, for benign samples, XSS

Auditor was performing similarly in identifying all requests as not malicious correctly. With that, it can be concluded that XSS Auditor outperforms the other detection mechanisms when detecting XSS attacks through mere HTTP requests. The confusion matrix of the results can be seen in Figure 7.

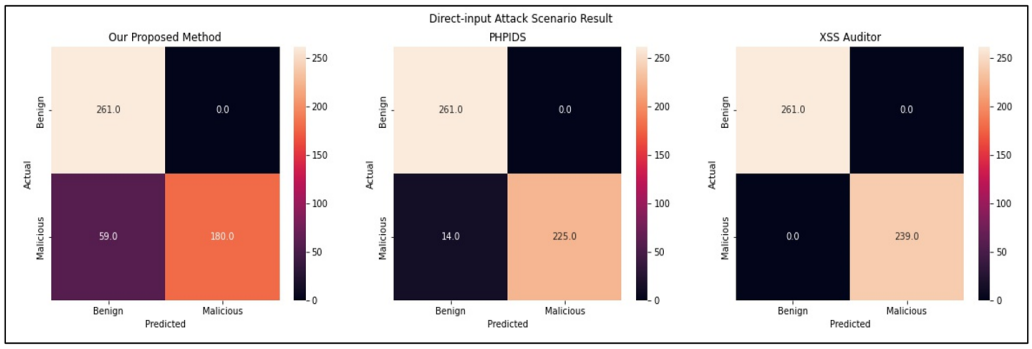

Figure 7. Cross-site scripting detection confusion matrix on direct-input scenario

Meanwhile, our analysis of the result for the Persisted Attack scenario is as follows. From 239 malicious payloads, our detection system detected 192 as potential Cross-site Scripting payloads (*true positive*), while 47 failed to be identified (*false negative*). From 261 benign payloads, our detection system successfully identified all the requests as harmless (*true negative*), leaving zero payload misclassified (*false positive*). On the other hand, neither PHPIDS nor XSS Auditor could correctly identify any malicious XSS payloads that originated inside the database, resulting in 239 *false negatives*. Though, for the benign samples, PHPIDS and XSS Auditor could still correctly identify all requests as not malicious (*true negative*), leaving 0 payloads misclassified. With that, it can be concluded that our proposed method outperforms the other detection mechanisms when detecting XSS attacks that originate from inside the database. The confusion matrix of the results can be seen in Figure 8.

We also calculated 4 basic evaluation metrics from the confusion matrix: *accuracy*, *precision* and *recall*, and the F1-score. The *precision* and *recall* metrics specifically are calculated because in designing such an attack detection system, the risk of having a malicious payload detected as a benign payload (*false negative*) could result in a devastating effect, more than the risk of having a benign payload falsely identified as malicious payload. Therefore, the *precision* and *recall* metrics allow us to capture that characteristic. However, achieving low *false positives* and *falseFeatures negatives* is the grand objective. Finally,

the calculation result of our proposed method, PHPIDS, and XSS Auditor can be seen in Table 7.

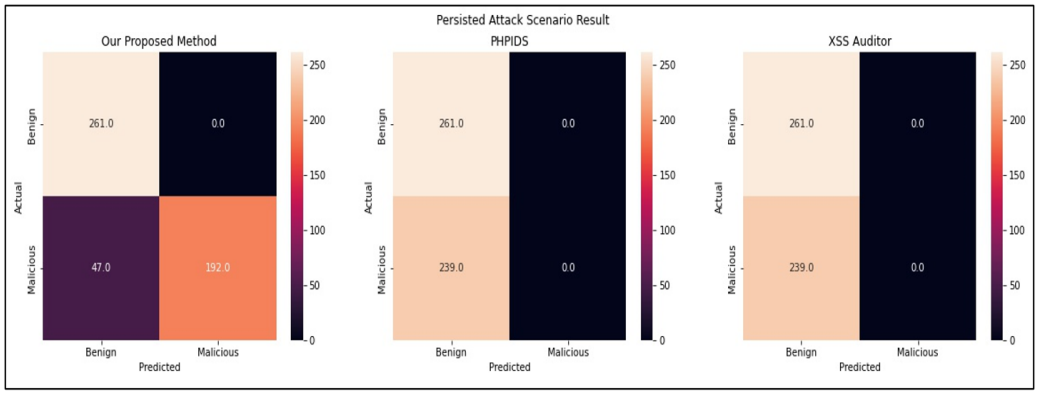

Figure 8. Cross-site scripting detection confusion matrix on persisted scenario

Table 7

Accuracy, precision, recall, and F1 of the evaluation process

Metrics	Our Detection System		PHPIDS		XSS Auditor	
Scenario	Direct-input	**Persisted Attack**	Direct-input	Persisted Attack	**Direct-input**	Persisted Attack
Accuracy	0.88	**0.91**	0.97	0.52	1.00	0.52
Precision	1.00	**1.00**	1.00	0.26	**1.00**	0.26
Recall	0.76	**0.80**	0.94	0.50	**1.00**	0.50
F1-score	0.86	**0.89**	0.97	0.34	**1.00**	0.34

Moreover, our analysis of the result obtained by our proposed method is described as Equation 2:

$$Accuracy = \frac{180 + 261}{180 + 261 + 0 + 58} = 0.88 \quad (2)$$

As an overall performance, the detection system achieves an accuracy of 0.88 means it was able to identify correctly 88% of the payloads (both benign and malicious) that were sent to the server. We believe this number is promising, although there is still much space for further improvement, especially in reducing the false negative rate (Equation 3).

$$Precision = \frac{180}{180+0} = 1 \quad (3)$$

The precision calculation reaching 1.0 point indicates that the detection mechanism has identified all the received attack payloads accordingly or with no false positives. The achieved result must be maintained because any *false positive* will directly reduce the system's effectiveness and inhibit the response to an attack due to additional checks that must be carried out on incoming warnings.

$$Recall = \frac{180}{180+58} = 0.76 \quad (4)$$

While precision measurement reaches a very high score, recall measurement indicates the opposite. As the score only reaches 0.76 points, this reflects that the detection mechanism still results in a high number of false negatives (Equation 4). It will have a side effect on the overall system's reliability since there are possibilities that an attack will go through undetected.

$$F1 = \frac{1\times 0.76}{1+0.76} = 0.86 \quad (5)$$

Finally, the F1-score measurement to calculate the harmonic mean between precision and recall that reaches 0.86 shows that the values of precision and recall are quite balanced but not perfect (Equation 5). As explained before, the high number of *false negative*s or low *recall* values is the main cause.

Result Analysis

From the above results, on the Persisted Attack scenario, our proposed method outperforms the other detection mechanisms in detecting Stored XSS attacks, in which the payload is already stored inside the database. This capability can be impactful, especially when implemented on an existing web application that might (or might not) have suffered an XSS attack because of recent attacks that can be detected and attacks that are prior to the installation of the detection mechanism. PHPIDS, although it is placed on the server-side, does not achieve the same result, mainly due to its inability to examine payloads from the database. Because PHPIDS's checking mechanism relies on the input from HTTP request alone, should an adversary find a way to trigger an XSS attack without having to put the payload on the HTTP request (e.g., through SQL Injection, HTTP request smuggling (Jabiyev et al., 2021), or Stored XSS attack performed before the PHPIDS is installed), the XSS payload cannot be detected.

Moreover, Chrome's XSS Auditor was also unable to detect Stored XSS attack attempts because PHPIDS cannot. Being a client-side detection mechanism, the source of information used by XSS Auditor to determine whether an XSS injection has occurred is only the information that the browser sends (i.e., HTTP request) and receives (i.e., HTTP response). Hence, every bit of data (including the malicious XSS payload) that might

originate from other locations, especially from the database, would already have been inserted into the HTTP response document when it arrived on the user's browser. As no part of the HTTP request is dangerously reflected in the HTTP response received by the auditor, no alert will be triggered. Figure 9 illustrates the gaps in our proposed detection system covered by both approaches.

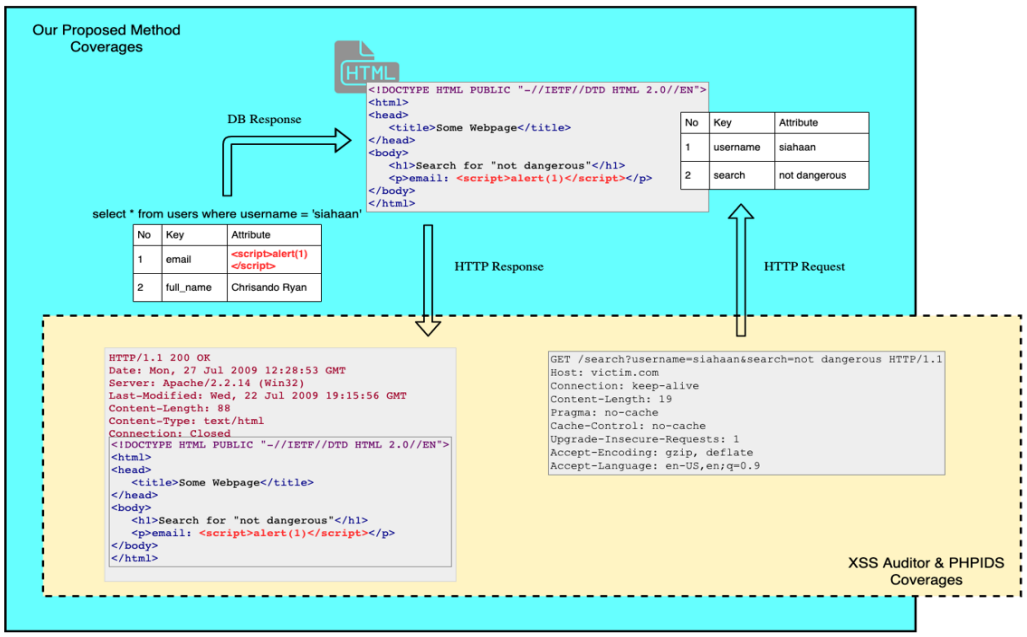

Figure 9. Different detection approaches between ours and other methods

However, another conclusion is that despite the good performance, our proposed method still had a noticeable number of *false negatives*. Our post-simulation analysis discovered that a high rate of a false negatives is majorly caused by the following:

1. Encoding

   As we used the XSS Payload List as the primary malicious data source, we found that many payloads use a non-standard encoding such as UCS-4BE and Windows-1251 to avoid detection of Web Application Firewall (WAF). Although the detection system is designed to mimic the browser's interpretation as closely as possible, these non-standard encoding differences were still causing an unhandled interpretation difference between the detection system with the web server and the browser. Table 8 illustrates the different interpretation that occurs in the simulation.

Table 8

Mismatch encoding interpretation on our detection algorithm

No	Payload	Browser Interpretation	Detection system Interpretation
1	<IMG SRC=”jav◆ascript:alert(’XSS’);”>		\t
2	<body background=javascript:alert(◆XSS◆);>	“	“
3	<iframe/◆/ src=javaSCRIPT:alert(1)	null	%00

◆ indicates the position of encoding mismatch

2. The flaw in the detection algorithm

In our post-simulation analysis, we recorded every payload sent to the server, its kind (benign or malicious), and the detection system’s response to the payload. From the failed detected payload list, we found that some HTML tags apparently can also be used to perform a Cross-site Scripting attack but are not included in the list of elements that must be checked by the detection system, as can be seen in Table 9.

Table 9

Mischecked element on the detection algorithm

No	Payload	Element
1	<svg><style> font-style:’<iframe/onload=alert(1)>’	<style>
2	<div/style=”width:expression(confirm(1))”>X	<div/style=…>
3	<iframe %00 src=” javascript:prompt(1) ”%00>	<iframe>
4	<iframe srcdoc=’<body onload=prompt(1)>’>	Srcdoc=…

CONCLUSION

Our proposed approach, a modified design for a server-side XSS detection system based on Google Chrome’s XSS Auditor, was able to maintain high fidelity characteristics by keeping interposition on the interface for the HTML parser and achieve a good result in detecting XSS payloads from both HTTP request input and inputs that are already stored inside the database. We believe that the original design of XSS Auditor can be more impactful in protecting users from XSS attacks by examining the Database response instead of only the user’s direct input, as can be observed on many client-side XSS detection systems. Compared to other detection approaches, our proposed methods have a strategic advantage in detecting Stored and Reflected XSS attacks with their ability to intercept and analyze user input already stored inside the database. The evaluation result shows that our approach achieves 88% accuracy, a considerably high number in identifying and

detecting malicious XSS payloads, although there is still room for further improvements. In the future, our detection algorithm can be extensively improved by considering more HTML elements to be checked and tweaking and matching the encoding process between the proposed detection system and the browser in general to minimize the false negative rate. A more comprehensive evaluation can also be conducted to measure the performance of the proposed method in terms of speed and memory consumption. Finally, while most browsers have their XSS protection, having more layers of security on the server side to guard the users might not be bad.

ACKNOWLEDGEMENT

We thank everybody involved in this research for their breakthroughs, guidances, and supports, including Universitas Bina Nusantara, Universiti Teknologi MARA (UiTM) Shah Alam, and the entire world's cybersecurity industry.

REFERENCES

Abaimov, S., & Bianchi, G. (2019). CODDLE: Code-injection detection with deep learning. *IEEE Access*, 7, 128617-128627. https://doi.org/10.1109/ACCESS.2019.2939870

Bates, D., Barth, A., & Jackson, C. (2010). Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International Conference on World Wide Web* (pp. 91-100). ACM Publishing. https://doi.org/10.1145/1772690.1772701

Cui, Y., Cui, J., & Hu, J. (2020). A survey on XSS attack detection and prevention in web applications. In *Proceedings of the 2020 12th International Conference on Machine Learning and Computing* (pp. 443-449). ACM Publishing. https://doi.org/10.1145/3383972.3384027

Gan, J. M., Ling, H. Y., & Leau, Y. B. (2020). A Review on detection of cross-site scripting attacks (XSS) in web security. In M. Anbar, N. Abdullah, & S. Manickam (Eds.), *International Conference on Advances in Cyber Security* (Vol. 1347, pp. 685-709). Springer. https://doi.org/10.1007/978-981-33-6835-4_45

Giménez, C. T., Villegas, A. P., & Marañón, G. Á. (2010). *HTTP data set CSIC 2010*. Information Security Institute of CSIC (Spanish Research National Council). https://www.tic.itefi.csic.es/dataset/

Jabiyev, B., Sprecher, S., Onarlioglu, K., & Kirda, E. (2021). T-Reqs: HTTP request smuggling with differential fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1805-1820). ACM Publishing. https://doi.org/10.1145/3460120.3485384

Khazal, I. F., & Hussain, M. A. (2021). Server side method to detect and prevent stored XSS attack. *Iraqi Journal for Electrical & Electronic Engineering*, *17*(2), 58-65. https://doi.org/10.37917/ijeee.17.2.8

Liu, M., Zhang, B., Chen, W., & Zhang, X. (2019). A survey of exploitation and detection methods of XSS vulnerabilities. *IEEE Access*, *7*, 182004-182016. https://doi.org/10.1109/ACCESS.2019.2960449

Rodríguez, G. E., Torres, J. G., Flores, P., & Benavides, D. E. (2020). Cross-site scripting (XSS) attacks and mitigation: A survey. *Computer Networks*, *166*, Article 106960. https://doi.org/10.1016/j.comnet.2019.106960

Swiat. (2008). *IE 8 XSS filter architecture/implementation*. Microsoft. https://msrc.microsoft.com/blog/2008/08/ie-8-xss-filter-architecture-implementation/

Sarmah, U., Bhattacharyya, D. K., & Kalita, J. K. (2018). A survey of detection methods for XSS attacks. *Journal of Network and Computer Applications*, *118*, 113-143. https://doi.org/10.1016/j.jnca.2018.06.004

Satish, P. S., & Chavan, R. K. (2017). Web browser security: Different attacks detection and prevention techniques. *International Journal of Computer Applications*, *170*(9), 35-41.

Shar, L. K., & Tan, H. B. K. (2011). Defending against cross-site scripting attacks. *Computer*, *45*(3), 55-62. https://doi.org/10.1109/MC.2011.261

Stock, B., Lekies, S., Mueller, T., Spiegel, P., & Johns, M. (2014). Precise client-side protection against DOM-based cross-site scripting. In *23rd USENIX Security Symposium* (pp. 655-670). USENIX Association.

Takahashi, H., Yasunaga, K., Mambo, M., Kim, K., & Youm, H. Y. (2013). Preventing abuse of cookies stolen by XSS. In *2013 Eighth Asia Joint Conference on Information Security* (pp. 85-89). IEEE Publishing. https://doi.ieeecomputersociety.org/10.1109/ASIAJCIS.2013.20

Vartouni, A. M., Kashi, S. S., & Teshnehlab, M. (2018). An anomaly detection method to detect web attacks using stacked auto-encoder. In *2018 6th Iranian Joint Congress on Fuzzy and Intelligent Systems (CFIS)* (pp. 131-134). IEEE Publishing. https://doi.org/10.1109/CFIS.2018.8336654

Wichers, D., & Williams, J. (2017). *OWASP top 10 - 2017.* OWASP Foundation. https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf

The Chromium Projects. (2019). *XXX Auditor*. https://www.chromium.org/developers/design-documents/xss-auditor

Yavanoglu, O., & Aydos, M. (2017). A review on cyber security datasets for machine learning algorithms. In *2017 IEEE International Conference on Big Data (Big Data)* (pp. 2186-2193). IEEE Publishing. https://doi.org/10.1109/BigData.2017.8258167