Xilara: An XSS Filter Based on HTML Template Restoration

Keitaro YAMAZAKI, Daisuke KOTANI and Yasuo OKABE

Kyoto University

Abstract. Cross Site Scripting (XSS) is one of the most fearful attacks against web applications because of its potential damage to users, e.g. sensitive data is stolen. XSS filter is one of existing mitigation technologies against XSS by monitoring communications between servers and browsers to find attack codes in HTTP requests. However, some attacks can bypass such XSS filters that checks the requests, for example, attacks that uses complex attack codes like base64-encoded ones, and attacks that may not include attack codes in the request, such as Stored XSS. This paper proposes a new XSS filter: Xilara to detect XSS attacks by monitoring HTTP responses instead of the requests. A key idea is that normal responses have very similar HTML document structures because they are usually generated by the same program (HTML template) and some parameters (untrusted data), but once an XSS attack succeeds, the structure of an HTML document changes because of the attack codes in the untrusted data. As a preparation, Xilara collects normal HTTP responses, and restores HTML templates. To detect that the response is contaminated by XSS attacks, Xilara checks whether an HTML document in the response is an instance of the restored template or not, and regards it being attacked if it fails. Our evaluation using XSS vulnerabilities reported in the real world shows that Xilara can detect XSS attacks whose attack codes are difficult to be detected by existing XSS filters, as well as performance comparison between Xilara and existing XSS filters.

1 Introduction

Cross Site Scripting (XSS) is one of the most fearful attacks towards web applications [1]. Attackers abuse XSS for various purposes such as accessing to sensitive user information, controlling the browser, or deceiving users by presenting fake information. The sensitive user information includes session information which is an identification of the user in the application. It is important to protect users from XSS, but there are still many vulnerable applications because of bugs in applications.

There have been several protections and mitigation techniques against XSS. We focus on an XSS filter, which detects XSS by auditing the network communication between clients and servers, because it can be introduced to the systems independently of the implementation of web applications. Some web browsers have built-in XSS filters [2, 3], and some web application firewalls provide XSS

filter functions [4]. However, since existing XSS filters detect XSS by finding attack codes in HTTP requests, attackers can sometimes bypass the detection mechanisms by carefully crafting and sending attack codes.

A typical XSS vulnerability occurs when an application constructs an HTML document with an HTML template (a structure of HTML documents) and data including valid HTML fragments from untrusted sources. If we can separate the HTML template and data accurately, we can detect XSS attacks through comparison of the structure of many HTML documents in responses.

In this paper, we propose a new XSS filter, Xilara, based on the idea that XSS attacks can be detected by checking the structures of the HTML documents in responses. First, Xilara collects HTML documents in non-harmful HTTP responses and restores HTML templates with the collected documents and existing methods for data extraction from multiple HTML documents. Then, to detect XSS attacks, Xilara confirms whether the structure of an HTML document in an HTTP response matches with the restored template, and regards the response is harmful due to XSS attacks if the response does not match with the template. Xilara can be applied not only to Reflected XSS but also to Stored XSS and can be used independently of an application code.

We implemented Xilara and conducted experiments to evaluate the performance of Xilara. We collected data of XSS attacks reported in the real world and compared the XSS detection capability of Xilara with that of existing XSS filter. The results show that, Xilara detected 94.5% of the XSS attacks but produced false positive detections on 20.6% of the non-attacked HTTP responses. Also, Xilara can detect all of the attacks which have base64-encoded attack codes though an existing XSS filter cannot detect any of these attacks. In addition, we show that Xilara can check whether the response is harmful or not with little overhead in terms of the response time to users.

In the following of this paper, Section 2 describes our research background. Section 3 introduces related works for XSS. Section 4 and 5 describes our proposed method and its implementation. Section 6 describes an evaluation of Xilara and its results. Section 7 shows a discussion, and Section 8 gives conclusion.

2 Background

2.1 XSS

XSS is an attack that injects a malicious script into a target web application. When this attack is successfully executed, an attacker can send a malicious JavaScript code to other clients accessing the target application and execute the code on their web browsers. By using XSS, the attacker can temper the web application's content and grab access tokens owned by other users.

Many web applications accept data submitted from users, but in some cases, data for attacks are submitted. We call these data submitted from users as untrusted data. OWASP classifies XSS [5] by the place where untrusted data is processed and whether untrusted data is permanently stored or not as shown in Table 1.

Table 1: Class of XSS

		Untrusted data is used at	
		Server	Client
Data Persistence	Stored	Stored Server XSS	Stored Client XSS
	Reflected	Reflected Server XSS	Reflected Client XSS

The Server-side XSS occurs when a web application includes untrusted data in an HTML document and sends it to the user. The web application should process untrusted data as text or attribute values in the HTML document, but when it simply concatenates the string representing HTML fragments and untrusted data, XSS occurs. The Client-side XSS occurs when a JavaScript code provided by the web application to web browser mishandles the untrusted data. In this case, an attacker crafts untrusted data to create unintended HTML elements through those APIs and adds HTML elements to execute malicious JavaScript.

Reflected XSS occurs when untrusted data in an HTTP request is not processed correctly in the process of generating HTTP response or in the JavaScript code in the web application. Stored XSS occurs when untrusted data sent from the user is permanently stored in a database, log files in the web server, a database in the web browser, etc. and when the web application does not properly handle these data.

In this research, we deal with Server-side XSS, and the XSS in the following examples and subsequent sections represents Server-side XSS except explicitly mentioned.

2.2 HTML Template

We will explain an HTML template, which is a concept used in this research. Many web applications use HTML templates to create HTML. For example, Ruby on Rails, which is a popular web application framework uses an HTML template called ERB [6], and Flask uses an HTML template called Jinja [7].

In the same web page, data is encoded in the same way [8]. The representation of HTML document generation method is called HTML template in this research. Many web applications generate an HTML document by replacing the variables in HTML templates with data.

There are algorithms such as RoadRunner [9], ExAlg [8] and DEPTA [10] which restore HTML templates from the multiple outputs of web pages. Though these algorithms are designed to extract data from web pages constructed from databases, they generate HTML templates during the process.

RoadRunner receives multiple HTML documents generated from the same HTML template and outputs a program called wrapper which extracts data from an HTML document without any knowledge of the web page structure. Since this program represents the encoding method of data in the web page, it is an HTML template. RoadRunner defines the wrapper with prefix markup-

languages which abstract the structure that appears in general web pages, and it is represented by the XML mainly consisting of the following elements.

<**tag**> HTML element. <*p class="a"*> will be represented as <*tag element="p" attrs="class:a"*>.
<**and**> [T1, ..., Tn]. A template which is a set of n templates (T1, ..., Tn).
<**plus**> [T1, ..., T1]. A template which is a set of consecutive templates *T1*.
<**hook**> (T1)?. A template which has optional template *T1*. *T1* sometimes appears in this template and sometimes doesn't appear.
<**variant**> Template indicating that the content of its child element is variable.
<**subtree**> This template represents that it is impossible for RoadRunner to generate the template at this node.

3 Related Works

There are roughly three types of countermeasures to prevent a Server-side XSS attack. One is to install XSS filter between the web application server and client. Second is to modify application code to detect XSS. The third is to modify the web browser to detect XSS. We introduce these types of existing XSS countermeasures while comparing with our research.

3.1 XSS Filter in Web Application Firewalls

Some web application firewalls (WAF) have XSS filters using regular expression and blacklists for example in Javed and Schwenk [11]. People can relatively easily install this XSS filter because it can be used independently from the web application. In their study, they consider that the HTTP request is an attack when it matches the following regular expression. OWASP ModSecurity Core Rule Set[1] is one of the well-known filters including such regular expression.

```
1  /(?:=|U\s*R\s*L\s*\()\s*[^>]*\s*S\s*C\s*R\s*I\s*P\s*T\s*:/i
```

These mitigations are effective when they can detect the attack string in HTTP requests. However, these are not effective when an attacker hides the attack payloads in HTTP requests using a complicated converting process of the application. For example, Kettle [12] reported that an attacker can bypass these mitigation techniques when an application uses some WAF and a web browser has built-in XSS filter. Another example can be found in an application which converts untrusted data given from outside as hexadecimal numbers into a UTF8 encoded string and displays it[2], and XSS filters introduced above cannot detect

[1] https://modsecurity.org/crs/
[2] It comes from a real application that has converted the external input value by calling the function(*utf8HexDecode*) as the following URL. `https://sourceforge.net/p/subsonic/code/4715/tree/trunk/subsonic-main/src/main/java/net/sourceforge/subsonic/util/StringUtil.java#l410`

attacks against this web application. Our method checks an HTTP response so that it can detect the attacks. Also, in the dataset used for the experiment of our research, we found a case where an attacker encodes attack string in base64 format and a web application decodes[3]. In this case, attack payload[4]

`j48c3ZnL29ubG9hZD1wcm9tcHQoL3hzc3Bvc2VkLyk`

is included in the HTTP Request, so regular expression implemented in above WAF cannot detect the attack.

3.2 XSS Protection Installed in an Application

Another method is to modify an application code and this method is relatively hard to be introduced because it is necessary to update the application code by hand or it is only applicable to applications developed in specific programming languages. However, it is possible to detect and process the untrusted data accurately because this method is implemented inside the application code.

A basic protection method of Server XSS is to escape HTML special characters in untrusted data when these data are going to be combined with strings representing HTML structure. For example, `<` in untrusted data should be converted to `<` so that it is treated as a character in HTML documents. However, there are still many vulnerable applications because sanitizing all untrusted data comprehensively is difficult in some cases.

In addition, there are methods using a policy configured in application servers to validate the HTTP response. The policy is used to prevent web browsers from loading the code not intended by the administrator of the application. Using Content Security Policy (CSP) [13], it is possible to specify the location or hash value of valid JavaScript codes by creating a policy. Noncespaces [14] and Document Structure Integrity [15] can detect attacks by assigning random numbers to trusted HTML elements and its attribute names. xJS [16] isolates legitimate client-side JavaScript code from the code comes from untrusted data. However, since these methods require specific configuration for each application, it is necessary to rewrite the code of the application in some cases, which is a great burden to the server administrator. Therefore, they are not necessarily said to be used widely and properly[17].

Since our method is an XSS filter-based defense mechanism, there is no restriction to the programming language of the web application and modification to its source code to install the filter. Therefore, compared with these XSS protections, an operator of the web application who is not the developer of it can install our XSS filter easily.

3.3 Web Browser built-in XSS filter

There are mitigation techniques implemented in web browsers. IE 8 using XSS filter [2], and Google Chrome using XSS Auditor [3]. They detect the attack

[3] `https://www.openbugbounty.org/reports/113400/`

[4] This is base64 encoded attack string of *"><svg/onload=prompt(/xssposed/)*

string in the HTTP request and prevent the attack if the HTTP response also includes a similar attack string.

These mitigation methods have the same issue with the XSS filter in WAF. They cannot detect XSS attacks when the attack codes are not included in HTTP request and when an attacker hides the attack payloads in HTTP requests using a complicated converting process of the application. However, they can use the same HTML parser installed in the web browser, and it improves the accuracy of the detection.

4 Our approach

We focused on the following features that appear when a Server-side XSS attack successes against a web application.

- Many web applications use HTML templates that describe how to encode data in HTML when constructing HTML dynamically.
- To execute JavaScript code on the victim's browser, attackers often inject new HTML elements and HTML element attributes.
- After attacker injects a new HTML element or attribute, the structure of HTML document becomes different from the structure of HTML document encoded by HTML template with expected data.

In particular, we focused on the difference between the structure of HTML document which the application usually generates with HTML template and that of HTML which the application constructs after attacker conducts XSS attack. For example, normal structure of HTML document generated by web application shown in Figure 1 will be that of Figure 2. The application receives an ID from a query parameter in the URI and has vulnerability against Reflected XSS. When one of the user accesses to the attack URI (e.g. `http://example.com/?id=<script>ATTACKSTRING</script>`), the structure of the HTML document will be that of Figure 3. In this research, our filter detects XSS depending on whether the observed structure of HTML document can be output from an HTML template.

```
1 <html>
2   <h1>Sample Vuln app</h1>
3   <p>
4     Hello, <?php echo $_GET['id'] ?>
5   </p>
6 </html>
```

Fig. 1: Example source code written by PHP. It has vulnerabilities against Reflected XSS and Stored XSS.

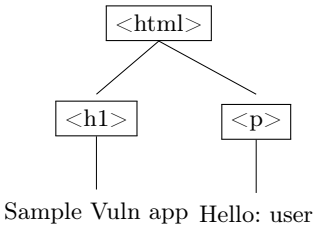

Fig. 2: HTML tree constructed with ordinary HTTP request

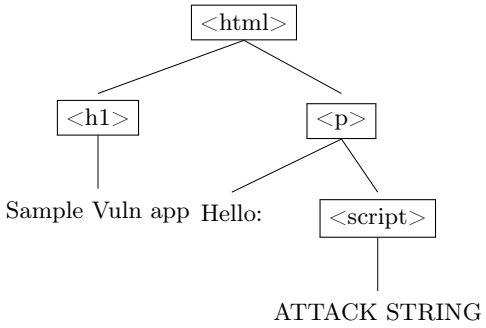

Fig. 3: HTML tree constructed when XSS attack is conducted

We propose a new XSS filter: Xilara (XSS filter based on HTML template restoration) which restores the HTML template from the HTTP response and detects the XSS. Figure 4 represents the overview of Xilara, and it consists of three stages.

HTML Collection Stage Xilara collects HTTP responses from the web server for some periods.

HTML Template Restoration Stage Xilara tries to restore the HTML template used by the web application from the HTTP response.

XSS Detection Stage Xilara uses the restored HTML template to detect if observed HTTP response is XSS attacked or not.

4.1 HTML Template

In this research, we define an HTML template as a tuple consisting of the following nodes.

Tag This node represents an HTML element that has a list of HTML element names and pairs of attribute name and attribute value. There are two types of attribute: variable and fixed. The Tag template t whose $t.name$ is p and $t.attributes$ is $[class="a"]$ (value is fixed) is encoded in HTML $<p\ class="a">$. Furthermore, the Tag node has an HTML template as a child element, and the parent-child relationship between the Tag nodes represents the parent-child relationship in the HTML element in the HTML document.

Loop This node represents that at least one HTML template ($T1$) appears consecutively. HTML template $T1$ is a child element of the Loop node.

Optional This node represents that one template ($T1$) sometimes appears and sometimes does not appear. HTML template $T1$ is a child element of the Optional node.

Ignore This node represents an element that could not restore an HTML template. This node has no child elements.

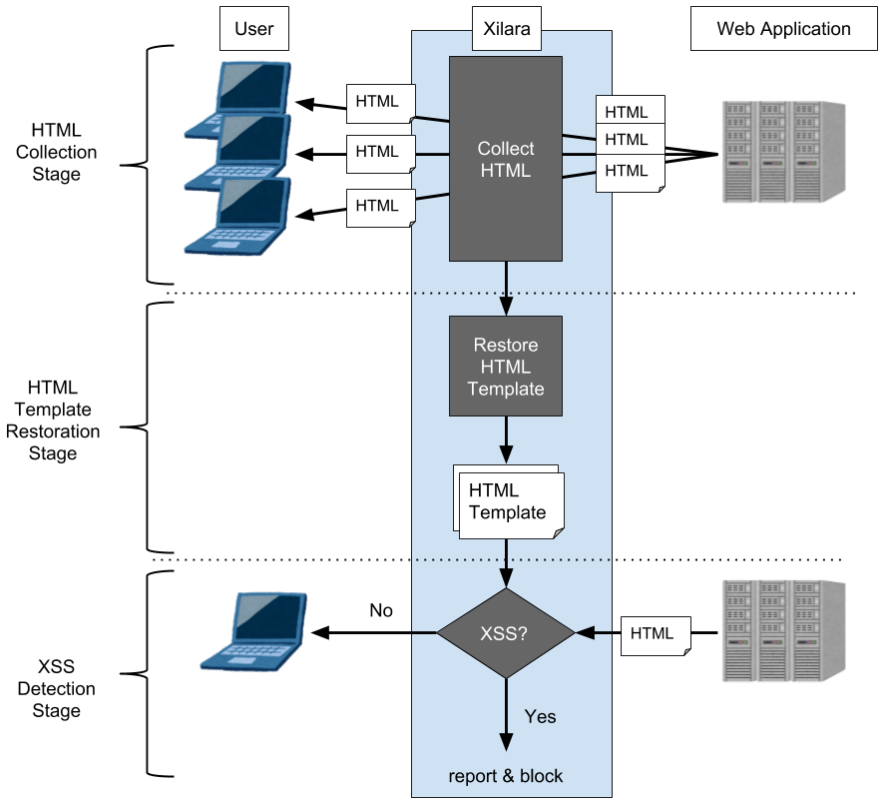

Fig. 4: Approach overview of Xilara

Null This node represents an empty node. This node has no child elements.

This definition is similar to RoadRunner's HTML template as described in Section 2.2, but we add the type of attribute value to Tag node to find whether the attribute value is variable or not for each Tag element. Also, to deal with XSS using attribute values of HTML element, a specific attribute of the Tag node of the HTML template is classified by attribute value. If an attacker can set an arbitrary value of *onerror*, *onload* attribute, *href* attribute of *A* HTML element, and so on, the attacker can conduct XSS attack. For example, attacker establishes the attack by setting attribute values like *onerror="ATTACK STRING"* or *href="javascript:ATTACK STRING"*. It is difficult to detect whether the value of *onerror*, *onload* attribute is malicious code or not because there is no clue for detection. We focus on two patterns of *href* attribute of the *A* HTML element. One is to use as a link to another web page using the HTTP protocol like *<a href="http://example.com">*. Another is to execute JavaScript code when the user clicks the HTML element. For example,

<a href="javascript:history.back()"> represents an HTML element that web browser backs to the previous page after the user clicks the element. Even if the developer of the Web application uses this *href* attribute for a link to another web page, if an attacker can set the attribute value from the outside, the attacker can embed the JavaScript code like the latter to execute attack strings written by JavaScript. Therefore, if there is no sample that the value of this attribute starts with *javascript:* and is always used for the former, we judge that the attribute is used as a link to another web page and guarantee not to be used in the latter. In addition to the *href* attribute of the *A* element, we treat *src* attribute of the *iframe* element, and so on, in the same way. [5].

4.2 HTML Document Collection Stage

In the first stage, Xilara behaves as an HTTP proxy and collects HTML documents in HTTP responses by monitoring communication between users and web applications. To ensure that the collected HTML document is a model of an HTML document without any XSS attack, it is desirable that the application works in an environment without an attacker at this stage. We think this constraint is not problematic because this constraint is easily fulfilled, for example if an administrator runs the web application on test environment which is commonly used to check the application's behavior in local or on the environment which is accessible only to the invited user.

4.3 HTML Template Restoration Stage

At this stage, Xilara restores the HTML template from the HTML documents collected at the previous stage. To restore HTML templates, Xilara applies existing algorithms such as ExAlg and RoadRunner. Template nodes in the HTML template outputted by these algorithms are corresponding to Tag node, Loop node, Optional node, and so on. Therefore, we can convert from the HTML template outputted by these algorithms into the HTML template handled in our research. In our research, the HTML template considers whether each value of some attributes starts with *javascript:* as described in Section 4.1. We describe detailed implementation of the conversion in 5.

4.4 XSS Detection Stage

In this stage, the application works in the real environment, and an external attacker may access it. At this stage, Xilara behaves as a reverse HTTP proxy

[5] We found these attributes in `https://html5sec.org/` have the same characteristics. *formaction* attribute in *button* element / *poster* attribute in *video* element / *href* attribute in *math*, *a*, *base*, *go*, *line* element / *xlink:href* attribute in any element / *background* attribute in *table* element / *value* attribute in *param* element / *src* attribute in *embed*, *img*, *image*, *script* element / *action* attribute in *form* element / *to*, *from* attribute in *set*, *animate* element / *folder* attribute in *a* element

and audits communication between clients and servers. It detects the XSS by checking whether the HTML document that the application server sends to the user is an instance of the HTML template generated at the previous stage. Algorithm 1 shows how to judge whether or not the observed HTML document is an instance of the HTML template.

Algorithm 1 Check if an HTML document is an instance of HTML template

Require: $HTMLRoot$: HTML Tree, $TemplateRoot$: HTML Template
1: $nodePairQueue := [[HTMLRoot, TemplateRoot]]$
2: **while** $nodePairQueue$ has element **do**
3: $nodePair :=$ first element of $nodePairQueue$
4: $html := nodePair[0]$
5: $template := nodePair[1]$
6: **if** $!checkNode(html, template)$ **then**
7: **continue**
8: **end if**
9: **if** The children of *html* and that of *template* should be checked **then**
10: Append node pairs which should be checked into $nodePairQueue$.
11: **continue**
12: **end if**
13: **if** The next siblings of *html* and that of *template* should be checked **then**
14: Append node pairs which should be checked into $nodePairQueue$.
15: **continue**
16: **end if**
17: **if** The next node of *html* and that of *template* should be checked **then**
18: Append node pairs which should be checked into $nodePairQueue$.
19: **continue**
20: **end if**
21: **return** True
22: **end while**
23: **return** False

Algorithm 1 receives HTML document and HTML template and judges if the root node of HTML document can be an instance of the root node of HTML template by using a deep-first search. First, this algorithm checks the attribute and name of HTML document node and HTML template node with function $checkNode$ described in Algorithm 2. Next, it checks the children of HTML document node and HTML template node. Similarly, it checks the next sibling nodes of HTML document node and HTML template node and next sibling nodes of parent nodes of them. Finally, if the HTML document node can be an instance, the algorithm returns *True*.

If the HTML document is an instance of the HTML template, Xilara sends the HTTP response to the user. If it is not an instance, Xilara reports to the administrator that an XSS attack is detected and sends an error message to the user. The administrator can configure the process executed after Xilara detects

Algorithm 2 Check if properties of *html* are same with that of *template*

```
1: procedure checkNode(html, template)
2:     if name of HTMLNode ≠ name of TemplateNode or HTMLNode has attributes not included in TemplateNode then
3:         return False
4:     end if
5:     while not an end of attributes of TemplateNode do
6:         tAttr := next attribute of TemplateNode
7:         hAttr := attribute of HTMLNode which has same name of tAttr
8:         if name of tAttr = id or class then
9:             if tAttr has fixed value and value of hAttr ≠ value of tAttr then
10:                return False
11:            end if
12:        else if tAttr receives both an URI and a JavaScript code then
13:            if the values of tAttr do not start with javascript: and value of hAttr start with javascript: then
14:                return False
15:            end if
16:        end if
17:    end while
18:    return True
19: end procedure
```

XSS, for example, he or she may send an HTTP response to the user even if Xilara detects XSS to keep the application highly available.

5 Implementation

In this chapter, we describe the detail of Xilara's design for each stage.

5.1 HTML Collection Stage

In the HTML Collection Stage, Xilara acts as an HTTP reverse proxy. As the input, Xilara receives the hostname and port number of the destination web application and port number of the reverse proxy. After startup, Xilara observes an HTTP request and an HTTP response between client and web application and saves the pair of an HTTP request path and HTTP response contents. Also, to prevent the saving of non-HTML content such as images, Xilara confirms *Content-Type* in an HTTP response header is *text/html* and stores only HTML documents. To collect the HTTP responses, the administrator can introduce some existing automatic web crawling techniques. For example, Heydon and Najork proposed a scalable web crawler [18] and Galan et al. [19] proposed a multi-agent XSS scanner which discovers the input locations and sends HTTP request[6]. Administrators can also manually generate an HTTP request by using the web application as its user.

[6] In this case, injected data should not be malicious attack code

5.2 HTML Template Restoration Stage

If the web application uses multiple HTML templates, Xilara should classify each collected HTML documents by its source HTML template. Usually, web applications use a different HTML template if a requested URI is different. Some web applications use URI routing patterns which indicate that all URIs which match the same pattern are related to the same HTML template. Xilara receives the collection of regular expressions as URL routing patterns from the owner of the web application. If Xilara receives no URI routing patterns, Xilara considers that URIs that has the same pathname are related to the same HTML template. Then, Xilara groups HTML documents constructed from the same HTML template.

When RoadRunner restores the HTML template, RoadRunner may parse the HTML document in a different way that real web browser does. These differences occur when the HTML structure of the document is not valid (e.g., closing HTML element without corresponding open HTML element). If Xilara uses a different parse result than that of the web browser, the attacker can exploit the difference and can successfully add attributes of HTML elements and HTML elements only recognized by the web browser. So, at this stage, Xilara parse the HTML document using *DOMParserAPI* on *Google Chrome* and encode it to a string representing an HTML document. Strictly speaking, since different web browsers parse an HTML document differently, Xilara should conduct this process for all web browsers, but in our first implementation, Xilara only considers about *Google Chrome*.

Then, Xilara uses RoadRunner to restore HTML template from HTML documents in the same group. RoadRunner receives HTML documents and preferences file. In the initial preferences of RoadRunner, it is required to match the HTML elements whose attribute values of *id* and *class* are same, and it is defined in the *attributeValues* setting. However, this setting prevents HTML template from being restored because some web applications set these attribute values dynamically. Therefore, in this research, we clear this setting. Instead, after restoring the HTML template, Xilara investigates the values of *id*, *class* attributes of each Tag node of the HTML template and if the attribute value is always same, Xilara considers the attribute value as constant. Xilara investigates the possible values of these attributes by collecting the attribute values for each Tag node through checking the correspondence between Tag nodes in the HTML template and HTML elements in the input HTML documents. Xilara uses *nodePair* matched finally in Algorithm 1 and to collect the correspondence.

Since the output of RoadRunner is an XML document composed of the elements described in Section 2.2, Xilara converts the output of RoadRunner to the HTML template used by Xilara with the following rule.

$< tag > \rightarrow Tag$ A <tag> node is converted to a Tag node having the same element name, attributes and child elements.

$< and > \rightarrow Tuple$ An <and> node which is a set of HTML templates is converted into a tuple containing its child elements.

$< plus >\rightarrow Loop$ A <plus> node is converted to a Loop node having the same child elements.

$< hook >\rightarrow Optional$ A <hook> node is converted to an Optional node having the same child elements.

$< subtree >\rightarrow Ignore$ A <subtree> node is converted to an Ignore node having the same child elements.

5.3 XSS Detection Stage

At this stage, Xilara behaves as an HTTP proxy like the HTML collection stage and does not detect XSS if the *Content-Type* header in HTTP response is not *text/html*. Xilara parses each HTML document through *Google Chrome* as in the HTML template restoration stage. After that, Xilara searches the corresponding HTML template from the URI in the HTTP request and checks whether the HTML document is an instance of the HTML template or not.

6 Evaluation

6.1 Depth Evaluation with Specific Vulnerabilities

To evaluate the process speed and XSS detection capability of Xilara, we conducted manual evaluation experiments with one web applications and two WordPress plugins. The targeted applications are shown in Table 2. For experiments, we used MacBook Pro 2016 with 2.9 GHz Intel Core i5 CPU and 8GB memory.

Table 2: Applications used for experiments

Application	Version	CVE or vuln info
Webmin	1.678	CVE-2014-0339
Count Per Day	3.5.4	`https://wpvulndb.com/vulnerabilities/8587`
AffiliateWP	2.0.9	`https://wpvulndb.com/vulnerabilities/8835`

We obtained 4 to 6 HTTP responses of the page where the XSS vulnerability exists through simulation of the typical use for each application, which causes a change of URI parameters and data in databases. We then restored the HTML template and tested whether Xilara can detect the XSS with the HTTP response created by the proof of concept (PoC) of the vulnerability. In addition, we tested whether Xilara detected XSS in normal HTTP responses by mistakes.

As a result of the experiment, we were able to detect attacks on Webmin and Count Per Day. However, Xilara could not detect the attack on AffiliateWP. This is because RoadRunner fails to restore an HTML template of AffiliateWP and the *subtree* appears in the template where the attack string is inserted. Also,

normal responses were not detected as XSS in all applications. We confirmed that RoadRunner had some problems with restoring *Optional* HTML templates.

Table 3 shows the average times of vulnerable pages (calculated ten times) and the average times which Xilara takes to parse HTML and judge XSS attacks. The result shows that the processing time of Xilara is moderate or low.

Table 3: Xilara performance result

Application	Response time	Xilara overhead
Webmin	423.46ms	14.16ms
Count Per Day	109.72ms	27.5ms
AffiliateWP	186.84ms	21.4ms

6.2 Large-Scale Evaluation with Vulnerable Website Dataset

Next, we conducted experiments using more web applications to investigate the differences of the behavior between Xilara and another XSS filter. We used the OpenBugBounty[7] as a dataset. OpenBugBounty lists the web pages containing the XSS vulnerability and the attack URI including attack strings against the page as a part of responsible disclosure. Experiments were carried out by the following procedure.

1. Collect reports that are still valid from OpenBugBounty.
2. Create normal requests and collect HTTP responses.
3. Investigate whether Xilara can detect XSS attacks.
4. Investigate whether another XSS filter can detect attacks for each report.

We will describe the detail of each step and results in the following sections.

Data Collection from OpenBugBounty First, we collected XSS datasets registered in OpenBugBounty. On November 26, 2017, the number of reports was 179702, and the number of published XSS reports was 74888. Since it takes time to investigate all the reports in this experiment, we only handle reports whose ID ends with 0.

Furthermore, we investigated whether each vulnerability exists even now. We collected HTTP responses outputted after we accessed the attack URI and confirmed the vulnerability by monitoring the execution of JavaScript functions such as *alert*, *prompt*, *confirm* after rendering the HTTP responses on *Google Chrome*. As a result, 4601 reports have vulnerabilities not fixed up to now. We proceeded the experiment using these 4601 reports.

[7] `https://www.openbugbounty.org/`

HTML Collection Stage Next, to create an HTML template and to verify XSS filters do not erroneously detect the XSS from a normal HTTP response, we created normal HTTP requests by removing the malicious code from the attack URI of each report. In most cases, if a web application has XSS vulnerability and XSS attack string is included in the query parameter, the web application considers the value of the query parameter is variable and applies to the HTML template. Therefore, if the value of query parameter in the URI matches one of the two regular expressions in Figure 5, we consider that the query parameter is used for the attack and replace the value of the query parameter with numerals so that we can fetch an HTML document constructed without attack strings. The first regular expression in Figure 5 matches with the HTML element such as `"><script>alert(1)</script>` whose text includes attack JavaScript code. The second regular expression matches with the HTML element such as `"><img src=x onerror=alert(1)>` whose attribute value includes attack JavaScript code. After we discovered the attack strings in URIs, we change the value of the parameter to five numbers from 1 to 5. We sent HTTP requests with replaced URIs and obtained the corresponding HTTP responses. As a result, we could collect all HTTP responses for 3408 reports.

```
1 /(['"]?[^>]*>)*<[^>]+>[^<]*(alert|confirm|prompt)
      [^<]*(<\/[^>]+>)?/ig
2 /(['"]?[^>]*>)*<[^>]+(alert|confirm|prompt)([^>]+>)?/ig
```

Fig. 5: Attack patterns we consider

Furthermore, we confirmed that some attack strings are encoded in special formats. For example, in eight reports, attack strings are encoded with base64. In one report, attack strings are displayed with hexadecimal digits. We also changed the value of the parameter used for each of these attacks and collected HTTP responses.

We continued our experiments using these 3417 reports.

XSS Detection with Xilara We restored the HTML template from four HTML documents with parameters 1 to 4 collected in the HTML collection stage. We succeeded to restore the HTML template in 3295 reports.

Then, we investigated whether Xilara can detect XSS in HTTP response generated from attack URI and whether Xilara detects no XSS in HTTP response collected in the previous stage.

XSS Detection with other XSS filters To compare Xilara with existing XSS filters, we investigated whether ModSecurity [4] and OWASP ModSecurity CRS can detect reported attacks. We use *libapache2-modsecurity* (version 2.7.7)

as a ModSecurity implementation with Apache (version 2.4.18-2ubuntu 3.5). We enabled *SecRuleEngine* option for ModSecurity and used default settings for other options. Also, we use OWASP ModSecurity CRS version 3 and enable the rules[8] to block the HTTP request when ModSecurity detects XSS.

We replaced the hostname and port number in the attack URI to those of the Apache server and investigated whether ModSecurity blocks the HTTP request or not after we send HTTP request of the attack URI.

Evaluation Result We compared the XSS detection rates and XSS misdetection rates among Xilara and ModSecurity with OWASP ModSecurity CRS.

Row 1 and 2 in Table 4 shows XSS detection rates against 3417 attack URIs. In total, Xilara detected XSS in 3230 attack URIs. Xilara could not detect XSS in 121 HTML documents because it could not restore the HTML template, and Xilara could not detect XSS in 66 HTML documents thought it could restore the HTML template. ModSecurity could detect 99.6% of the attacks in attack URIs because we used the attack URIs which match the patterns in Figure 5. However, ModSecurity could not detect all of the nine attacks in which attack string is encoded with base64 or hexadecimal. Xilara can detect eight of these attacks and it fails to restore the HTML template correctly in one of these attacks.

Row 3 and 4 in Table 4 shows the rates of XSS detection against $3417 * 4 = 13668$ normal HTTP responses (and HTTP requests) which were used to create HTML template and 3417 normal HTTP responses (and HTTP requests) which were used for verification. Xilara detected XSS by mistakes in 1640 HTML documents though they are used to construct HTML templates. Xilara detected XSS by mistakes in 703 HTML documents which were used for verification.

Table 4: XSS detection rates against attack URIs and normal HTTP responses

	Xilara	ModSecurity with CRS
All attacks	94.5%	99.6%
Attacks using some encodings	88.9%	0%
Template source responses	12.0%	0.18%
Verification responses	20.6%	0.18%

7 Discussion

7.1 Adaptive Attacker against Xilara

We discuss adaptive attacker against Xilara and attacks that cannot be detected by Xilara. Xilara detects XSS attacks using the result of matching of restored

[8] REQUEST-941-APPLICATION-ATTACK-XSS.conf and REQUEST-949-BLOCKING-EVALUATION.conf

templates and the HTML documents. It means if an attacker can craft HTML documents for XSS attack that matches the template, the attacker can bypass Xilara.

In the above example, an attacker can increase the number of li elements. However, an attacker cannot execute arbitrary scripts to steal user's data. For an attacker to avoid Xilara and run scripts, an HTML element or attribute that can include a context for executing JavaScript (in this paper, we call a JavaScript execution context) should appear in the template. In addition, this context should not be a fixed value and should exist after the part that the attacker can control in the document. As a result, there are following patterns of HTML document structures that an attacker can avoid detection.

JavaScript execution context in Loop If Loop node includes a JavaScript execution context that is not a fixed value as shown in Figure 6 and if the attacker can inject HTML element, the attacker can avoid the filter by sending *text1<script>attack string</script>
*.

JavaScript execution context in Optional If Optional node includes a JavaScript execution context that is not a fixed value as shown in Figure 7 and if the attacker can inject HTML element, the attacker can avoid the filter by sending *text1<script>attack string</script>*. This attack is available only if *<script>* HTML element in Optional node does not appear.

Attacker controlled JavaScript execution context If the attacker can directly control the text in the *<script>* element or attribute values which are JavaScript execution context, the attacker can insert attack strings without changing the HTML document structure. In some cases, the attacker can also bypass general protection methods that use HTML escaping.

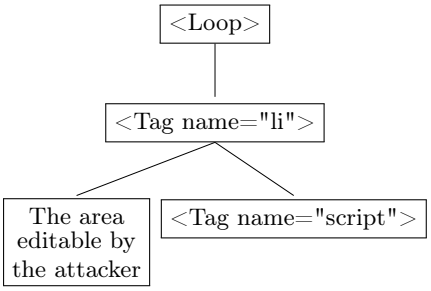

Fig. 6: Template which contains dynamic JavaScript code in Loop

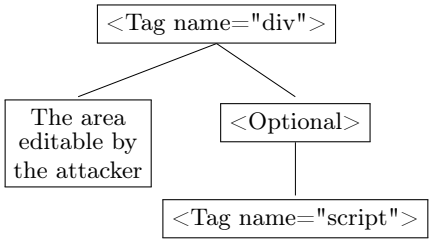

Fig. 7: Template which contains dynamic JavaScript code in Optional

7.2 Constraint of Xilara and Its Use Case

There are some constraints to use Xilara. One of the constraints is that Xilara should works in an environment without an attacker at HTML document collection stage. Another is that the user of Xilara should prepare the URL routing patterns if the web application uses multiple HTML templates. However, if the URIs of the application that has the same pathname are always related to the same HTML template, the user does not need to prepare. The third is that the user of Xilara should run Xilara as an HTTP reverse proxy.

We suppose that the administrator of the web application can use Xilara because he or she can prepare the environment without an attacker. Since Xilara does not require the source code of the application, the user of Xilara should not always be a developer of the application, and Xilara runs independently of the programming language of the application.

8 Conclusion

In this paper, we propose a new XSS filter, Xilara, to address the issue that attackers can sometimes bypass existing XSS filters that checks attack codes in requests by carefully crafting and sending the attack codes. Our key idea is that the harmful HTML documents in responses have different structures of the document, and it can be detected because many web applications generates HTML documents with very similar structures in responses from the same programs. Xilara uses an HTML template, and we define our HTML template model. In our HTML template model, we distinguish some HTML attributes with its value to detect XSS attacks which exploit those attribute values. Xilara observes an HTML structure in ordinary HTTP responses and restores HTML templates to detect XSS. To restore the HTML templates, we apply RoadRunner which has been developed for data extraction from multiple HTML documents. We implement Xilara as a proxy between clients and servers, and Xilara can coexist with the existing XSS filters.

We also conducted experiments to evaluate the performance of Xilara. We collected the XSS attack dataset from OpenBugBounty and evaluated Xilara and confirmed that Xilara detected 94.5% of the XSS attacks but judged XSS attacks mistakenly on 20.6% of the non-attacked HTTP responses. Xilara can also identify the attacks which use some encodings though an existing XSS filter cannot detect any of these attacks. In addition, our manual experiment shows that overhead of Xilara in each request is moderate or low.

Future works include more extensive evaluation of Xilara using various kinds of XSS vulnerabilities that current XSS filters are hard to detect, and improvement of accuracy of HTML template restoration to decrease the false positive rate.

References

1. Wichers, D.: Owasp top-10 2013. OWASP Foundation, February (2013)

2. Ross, D.: Ie 8 xss filter architecture / implementation. https://blogs.technet.microsoft.com/srd/2008/08/19/ie-8-xss-filter-architecture-implementation/ (2008)
3. Bates, D., Barth, A., Jackson, C.: Regular expressions considered harmful in client-side xss filters. In: Proceedings of the 19th international conference on World wide web, ACM (2010) 91–100
4. Trustwave: Modsecurity: Open source web application firewall. https://www.modsecurity.org/ (2004)
5. Wichers, D.: Types of cross-site scripting. https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting
6. Dave, T., David Heinemeier, H.: Agile web development with rails. Citeseer (2005)
7. Lokhande, P., Aslam, F., Hawa, N., Munir, J., Gulamgaus, M.: Efficient way of web development using python and flask. (2015)
8. Arasu, A., Garcia-Molina, H.: Extracting structured data from web pages. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, ACM (2003) 337–348
9. Crescenzi, V., Mecca, G., Merialdo, P., et al.: Roadrunner: Towards automatic data extraction from large web sites. In: VLDB. Volume 1. (2001) 109–118
10. Zhai, Y., Liu, B.: Structured data extraction from the web based on partial tree alignment. IEEE Transactions on Knowledge and Data Engineering **18**(12) (2006) 1614–1628
11. Javed, A., Schwenk, J.: Towards elimination of cross-site scripting on mobile versions of web applications. In: International Workshop on Information Security Applications, Springer (2013) 103–123
12. Kettle, J.: When security features collide. http://blog.portswigger.net/2017/10/when-security-features-collide.html (2017)
13. Stamm, S., Sterne, B., Markham, G.: Reining in the web with content security policy. In: Proceedings of the 19th international conference on World wide web, ACM (2010) 921–930
14. Van Gundy, M., Chen, H.: Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In: NDSS. (2009)
15. Nadji, Y., Saxena, P., Song, D.: Document structure integrity: A robust basis for cross-site scripting defense. In: NDSS. Volume 2009. (2009) 20
16. Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E.P., Karagiannis, T.: xjs: practical xss prevention for web application development. In: Proceedings of the 2010 USENIX conference on Web application development, USENIX Association (2010) 13–13
17. Weichselbaum, L., Spagnuolo, M., Lekies, S., Janc, A.: Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM (2016) 1376–1387
18. Heydon, A., Najork, M.: Mercator: A scalable, extensible web crawler. World Wide Web **2**(4) (1999) 219–229
19. Galán, E., Alcaide, A., Orfila, A., Blasco, J.: A multi-agent scanner to detect stored-xss vulnerabilities. In: Internet Technology and Secured Transactions (ICITST), 2010 International Conference for, IEEE (2010) 1–6