一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
IA2017-49 (2017-11)

Xilara: XSS audItor using htmL templAte restoRAtion

Keitaro YAMAZAKI[†], Daisuke KOTANI[‡], and Yasuo OKABE[‡]

[†] Graduate School of Informatics,
Kyoto University
[‡] Academic Center for Computing and Media Studies,
Kyoto University

Abstract Mitigating Cross Site Scripting (XSS) is important to protect user's sensitive data in the web applications. XSS mitigation without modifications of application's code is beneficial to protect many systems by one system. However, such mitigations depend on request or correspondence between request and response. We propose a new XSS filter, Xilara, that audits structure of responses. First, Xilara collects normal responses and restores HTML template automatically. Second, Xilara detects the stored XSS attack by verifying if the structure of response matches with the template. Our preliminary results show that Xilara can mitigate some known stored XSS vulnerabilities in real applications with acceptable performance.

Key words Security, XSS, Web, HTML

1 Introduction

Cross Site Scripting (XSS) is one of the most fearful attack towards web application[**?**] and many vulnerabilities related to this attack is now being reported. Attackers use XSS to gain access to sensitive user information via user's web browser, control the browser or deceive users by presenting fake information. These data include 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 application's bug.

There has been several protection and mitigation techniques against XSS. One of widely used methods is to convert untrusted data to be treated as string in the HTML context. However, many applications have not implemented this method yet, so mitigation techniques are also needed. Some mitigation techniques are implemented in some major browsers as XSS filter (e.g. XSS Auditor[**?**] in Google Chrome and XSS filter[**?**] in IE). Owner of the application can introduce Content Security Policy[**?**], which represents a policy of trusted contents in an application and which is implemented in almost all major browsers. However, the former mitigation technique is effective only for reflected XSS, which is described in next section. The latter is not widely used in general because it requires the owner of application to configure application settings.

A typical XSS vulnerability occurs when an HTML document is constructed with template and data including valid HTML fragments from untrusted sources. In this paper, we propose a new XSS filter: Xilara. First, Xilara observes HTML structure in ordinary HTTP responses and restores the HTML template. To restore HTML templates, we apply existing methods for data extraction from multiple HTML documents. After that, Xilara confirms whether the HTML structure in 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 independent of application code.

We implemented Xilara and conducted preliminary experiments. We used real web applications which have XSS vulnerabilities and applied Xilara to them to detect XSS attack. The results show that Xilara can detect XSS attacks at realistic speed in several applications.

In this paper, Section 2 explains XSS. Section 3 introduces related works for XSS. Section 4 describes our proposed method. Section 5 explains implementation for Xilara, and Section 6 describes preliminary experiments and its results. Section 7 shows a discussion about adaptive attacker, and Section 8 gives concluding remarks and future work.

2 XSS

XSS is an attack that executes JavaScript on the victim browser which accesses the target web page. For example, an attacker embeds an attack script written by JavaScript in the HTML document of the target page by XSS. The attacker can execute the attack script with the authority (called origin) of the target page. As a result, the attacker can steal his/her to impersonate as a victim cookie and can alter the contents of the web page.

XSS is classified into three types in terms of causes of XSS vulnerabilities and conditions required to the attacks.

Reflected XSS

A reflected XSS vulnerability occurs when a web

Figure 1: Example source code written by PHP. It has two XSS vulnerabilities at line 5 and line 6.

```
1 <?php
2 $user = fetchUserInfo($_GET['id']);
3 ?>
4 <html>
5   Id: <?= $_GET['id'] ?>
6   UserName: <?= $user['username'] ?>
7 </html>
```

application embeds the untrusted data included in the HTTP request into the HTML document for the response without sanitizing it. To establish the attack, attacker has the victim browser sends a request including the attack string. Attackers often embed attack string in the URL query or POST data.

Stored XSS
A stored XSS vulnerability occurs when a web application embeds the attack string, which is stored in the databases or some other places, in the HTML document for the response without sanitizing it. The attacker stores the attack string into the database, and when the victim browses the vulnerable page, attack string is executed. The attacker has no need to have the victim send a crafted request.

DOM based XSS
DOM based XSS is an attack that exploits vulnerability of JavaScript code executed on the browser. This vulnerability is located in client side JavaScript code, not in a code of web application.

We show an example of an application written in PHP which is vulnerable to reflected XSS and stored XSS attacks in Figure 1. Since this application responds the HTML document including the id parameter received from the browser (as written in the line 5), this application has a reflected XSS vulnerability. Also, if an attacker can change username, it has a stored XSS vulnerability because the username is printed at line 6. An attacker could steal the victim's cookie by embedding attack string such as *<script>location.href="//attacker.com/" + document.cookie</script>* using these vulnerabilities.

In this research, we deal with reflected XSS and stored XSS which are related to the vulnerability in server side application.

3 Related works

A basic protection method of XSS is to sanitize HTML special characters in untrusted data. For example, < is 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. For that reason, various mitigation techniques against XSS have been proposed and deployed.

There are mitigation techniques implemented in web browsers. IE 8 using XSS filter [?], and Google Chrome using XSS Auditor [?]. They detect the attack string in the HTTP request and prevent the attack if the HTTP response also includes a similar attack string. These mitigations are effective for reflected XSS because they can detect the attack string in HTTP requests, but are not effective for stored XSS. Also, these are not effective when an attacker hides the attack payload in HTTP requests using a complex conversion process of the application. For example, Kettle [?] has been reported that an attacker can bypass these mitigation techniques when an application uses some WAF.

Also, there are XSS filters using regular expression and blacklists for example in [?]. OWASP ModSecurity Core Rule Set[1] is one of the popular filters including such XSS filter. These filters have the same issues as in the web browser's XSS filter described above.

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) [?], it is possible to specify the location or hash value of valid JavaScript codes by creating a policy. Noncespaces [?] can detect attacks by assigning random numbers to trusted HTML elements and its attribute names. 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 widely used.

RoadRunner [?] and ExAlg [?] are algorithms that restore HTML templates from multiple HTML documents. These researches aim to extract the structured data from web pages. The template generated by RoadRunner is represented by XML consisting of the following XML 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 template *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 contents of its child element is variable.

[1] https://modsecurity.org/crs/

Figure 2: HTML 1

```
<html>
  <div></div>
  <ul>
    <li>First</li>
    <li>Second</li>
  </ul>
</html>
```

Figure 3: HTML 2

```
<html>
  <div><a href="/next">Next</a></div>
  <ul>
    <li>First</li>
    <li>Second</li>
    <li>Third</li>
  </ul>
</html>
```

<subtree> This template represents that it is impossible for RoadRunner to generate the template at this node.

For example, RoadRunner can generate the template using the HTML Figure 2 and Figure 3 which is generated from same template. Figure 4 shows the template generated by RoadRunner.

4 Our approach

When an application has an stored XSS vulnerability, it is not necessary for attackers to craft the victim user's request. Therefore, in order to detect the attack in such case, it is required to check whether the HTML document structure in HTTP response is valid or not (because of the injection of the DOM node caused by the attacker). In typical web applications such as BBS and blogs, user generated data stored in a database is applied to an HTML template where an HTML document is generated. In most cases, HTML templates has syntax which represent variables. For example, *<?= $variable ?>* is a syntax used in PHP as it appears in Figure 1. Therefore, it is considered that there are XSS vulnerabilities in the location where such syntax appears and these variables can be attack string. In addition, since the HTML document structure changes when an attacker injects an attack string composed with HTML elements, it is possible to detect XSS attack by comparing the outputted HTML document structure with the template. Furthermore, if it is possible to restore the HTML template from the HTTP responses of the web application, we can implement the filter independent from the application code or its language.

We propose a new XSS filter: Xilara, which has two stages. Figure 5 represents an overview of its approach.

In the first stage, Xilara collects the HTML documents in the HTTP responses, and restores the HTML template which is used as the valid HTML document structure. In this stage, it requires that application runs in an environment where attackers do not exist. These requirement enables the filter to collect the HTML documents that are not attacked. We think it is not difficult to prepare such environment. For example, test environments where only limited user can access satisfy the requirement.

Also, we introduce some special templates. Browsers execute JavaScript code only if the values of some HTML element attributes follows a special format (e.g. *<a href="javascript:alert(1)">*). Those attributes have multiple states. For example, with regard to the *href* attribute of the *A* HTML element, sometimes its value represents an URL such as *<a href="http://example.com">* and sometimes represents an JavaScript code such as *<a href="javascript:back()">*. In the former case, the attribute is used to represent a link to some web page. In the latter case, it is used to trigger some actions. Since an attacker can cause XSS using these attributes with unintentional purpose, we add some special templates and distinguish the attribute values starting with *javascript:* and other values. If *href* attribute values always represent URL, we treat this attribute as URL. Restored template ensures that the attribute value never starts with *javascript:*. Now, it possible to detect XSS attacks exploiting *href* attribute values intended to be used as a URL link to another web page.

In the second stage, the filter runs in a production environment, and attackers can access the application. In this stage, the filter audits the HTML document and judges whether it matches the template to detect the XSS attack.

Xilara considers the template as a finite automaton and starts the matching from the first HTML element node with breadth first search. In this automaton, both opening nodes and closing nodes of the *<tag>* template are the states and a transition function is basically matching of next HTML element node and next template node. Also, *<loop>* and *<hook>* affect this transition function to accept looped nodes and optional nodes. If all HTML element nodes can match with template nodes and if last state is an accepting state, the matching will success, otherwise the matching will fail. If the structure of HTML document matches the template, user will receive it, if it does not match, the incident is reported to the administrator. When the XSS is detected, administrator can choose whether to block the HTTP response or not. If you want to raise the availability as much as possible, HTTP responses will not be blocked and user will receive HTML documents. If you want to protect the user as much as possible, HTTP responses will be blocked and users will receive an error message.

Since Xilara behaves as an HTTP reverse proxy (or may be an HTTP forward proxy) in both stages, server

Figure 4: Template generated by RoadRunner

```
<and>
  <tag element="html" depth="0" attrs=""/>
  <tag element="head" depth="1" attrs=""/>
  <tag element="/head" depth="1" attrs=""/>
  <tag element="body" depth="1" attrs=""/>
  <hook>
    <and>
      <tag element="a" depth="2" attrs="href:/next"/>
      <pcdata depth="3"><![CDATA[Next]]></pcdata>
      <tag element="/a" depth="2" attrs="href:/next"/>
    </and>
  </hook>
  <tag element="ul" depth="2" attrs=""/>
  <plus>
    <and>
      <tag element="li" depth="3" attrs=""/>
      <variant label="_A_"><pcdata depth="4"><![CDATA[Third]]></pcdata></variant>
      <tag element="/li" depth="3" attrs=""/>
    </and>
  </plus>
  <tag element="/ul" depth="2" attrs=""/>
  <tag element="/body" depth="1" attrs=""/>
  <tag element="/html" depth="0" attrs=""/>
</and>
```

administrators can easily introduce the filter without the knowledge of the application's code.

5 Implementation

Next, we describe more implementation details of Xilara.

In the first stage, we implement the HTTP proxy running between the web application and its user to collect the HTTP response. Then, Xilara restores the template from HTTP responses. In this research, we adopted RoadRunner [**?**] [2] as an engine for restoring templates. We combined the restored template with our special template which distinguish some attributes due to its values.

Furthermore, we complement the differences between the HTML document tree generated by the HTML parser of RoadRunner and that generated by the HTML parser of the actual web browser (such as Google Chrome). These differences occur when the HTML structure of the document is not valid (e.g. closing HTML element without corresponding open HTML element). Before RoadRunner processes the HTML documents, we parse the HTML documents with Google Chrome using *DOMParser* API, then reconstruct the HTML string. It prevents the attacker crafting attack string with HTML elements or attribute values which are detected by web browsers but are not detected by RoadRunner's HTML parser.

[2]RoadRunner's implementation is publicly available at http://www.dia.uniroma3.it/db/roadRunner/

In the second stage, Xilara judges whether the HTML document structure served by the application matches the template. Before the judging, Xilara parses the HTML document with Google Chrome in the same way as in the first stage, because the template is derived from HTML documents parsed by the browser.

6 Preliminary Experiments

In order to evaluate the process speed and XSS detection rate of Xilara, we conducted preliminary experiments with one web applications and two WordPress plugins. The targeted applications are shown in Table 1.

Table 1: Applications used for experiments

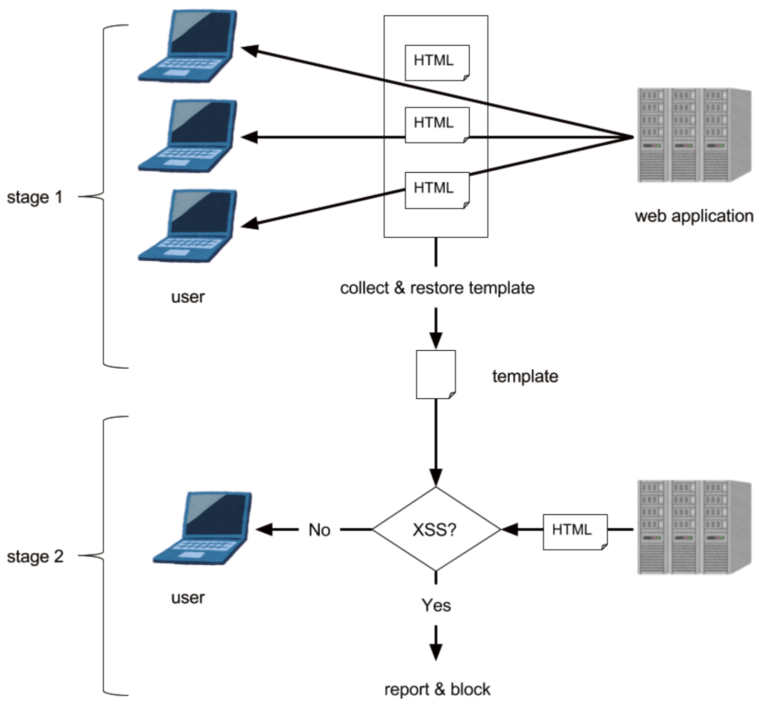
Application	Version	CVE
Webmin	1.678	CVE-2014-0039
Count Per Day	3.5.4	N/A[1]
AffiliateWP	2.0.9	N/A[2]

For experiments we used MacBook Pro 2016 with 2.9 GHz Intel Core i5 CPU and 8GB memory.

For each application, we changed the URL parameter and data in databases and obtained 4 to 6 ordinary HTTP responses of the page where the XSS vulnerability exists. We then restored the template and tested

[1]https://wpvulndb.com/vulnerabilities/8587
[2]https://wpvulndb.com/vulnerabilities/8835

Figure 5: Approach overview of Xilara



whether Xilara can detect the XSS with the HTTP response created by the attack string provided as PoC. Also, we tested whether ordinary HTTP responses were detected by Xilara.

As a result of experiment, we found that normal responses were not detected as XSS in all applications. Also, 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 a template of AffiliateWP and the *subtree* appears in the template where the attack string is inserted. Since we confirmed that $hook$ template was not restored correctly, eliminating this error is a future work.

Also, we present the average times of vulnerable pages (calculated 10 times) and the average times which Xilara takes to parse HTML and judge XSS attack in Table 2. It is shown that the processing time of the filter is low, so the overhead of Xilara is moderate or low.

Table 2: 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

7 Discussion

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 templates and the HTML documents. This means if attacker can craft HTML documents for XSS attack that matches the template, Xilara will be bypassed. For example, if an attacker can control the content of the li element of the template Figure 4 and if he sets it as *text1</li><li>text2*, the number of *li* elements will be changed and it may cause unintended result.

In this example, an attacker can increase the number of li elements however, an attacker cannot execute arbitrary scripts to steal user's data. In order for an attacker to avoid Xilara and execute scripts, an element or attribute that can include a context for executing JavaScript (in this paper, we call a JavaScript execution context) should be appeared in the template. Also, 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 document structures that an attacker can avoid detection.

JavaScript execution context in <plus> If <plus> includes a JavaScript execution context that is not a fixed value as shown in Figure 6 and the attacker can control the data at line 3, the attacker can avoid the filter by sending *text1<script>attack string</script></li><li>*.

JavaScript execution context in <hook> If <hook> includes a JavaScript execution context that is

Figure 6: Template which contains dynamic JavaScript code in <plus>

```
<plus><and>
  <tag element="li" />
    <variant><pcdata /></variant>
    <tag element="script" />
      <variant><pcdata /></variant>
    <tag element="/script"/>
  <tag element="/li"/>
</and></plus>
```

Figure 7: Template which contains dynamic JavaScript code in <hook>

```
1 <and>
2   <variant><pcdata /></variant>
3   <hook>
4     <tag element="script" />
5       <variant><pcdata /></variant>
6     <tag element="/script"/>
7   </hook>
8 </and>
```

not a fixed value as shown in Figure 7 and the attacker can control the data at line 2, the attacker can avoid the filter by sending *text1<script>attack string</script>*. This attack is available only if <*script*> element in <hook> 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, attacker can also bypass general protection methods that uses HTML escaping.

8 Concluding Remarks

In this paper, we propose a XSS filter, Xilara, which can be used independent of the server side application's code and without handwriting the policy. We confirmed that attacks against two applications out of the three were detected in our experiment. Even for one case that Xilara could detect the attack, it is considered that the attack can be detected if the template is restored correctly. Also, our experiment shows that overhead of Xilara in each request is moderate or low. In addition, Xilara is implemented as a proxy between client and server, and Xilara can coexist with the existing XSS filters. Therefore, Xilara can be used as a complementary filter to detect attacks that cannot be detected by other XSS filter.

However, there are some situations that attackers can avoid Xilara as we discussed in previous section. It is our future work to experiment and investigate how many such patterns are included in a larger number of templates in various web applications. Also, since each web application often has several templates, techniques to choose the template for each response is needed and it is our future work.

References

[1] Arvind Arasu and Hector Garcia-Molina. Extracting structured data from web pages. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 337–348. ACM, 2003.

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

[3] Valter Crescenzi, Giansalvatore Mecca, Paolo Merialdo, et al. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, volume 1, pages 109–118, 2001.

[4] Ashar Javed and Jörg Schwenk. Towards elimination of cross-site scripting on mobile versions of web applications. In *International Workshop on Information Security Applications*, pages 103–123. Springer, 2013.

[5] James Kettle. When security features collide. `http://blog.portswigger.net/2017/10/when-security-features-collide.html`, 2017.

[6] David Ross. Ie 8 xss filter architecture / implementation. `https://blogs.technet.microsoft.com/srd/2008/08/19/ie-8-xss-filter-architecture-implementation/`, 2008.

[7] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, pages 921–930. ACM, 2010.

[8] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *NDSS*, 2009.

[9] Dave Wichers. Owasp top-10 2013. *OWASP Foundation, February*, 2013.