Preventing Persistent Cross-Site Scripting (XSS) Attack By Applying Pattern Filtering Approach

Imran Yusof
Department of Computer Science
International Islamic University Malaysia
Kuala Lumpur, Malaysia
suretarget@gmail.com

Al-Sakib Khan Pathan
Department of Computer Science
International Islamic University Malaysia
Kuala Lumpur, Malaysia
sakib@iium.edu.my

***Abstract*— Cross-Site Scripting (XSS) vulnerability is one of the most widespread security problems for web applications, which has been haunting the web application developers for years. Various approaches to defend against attacks (that use XSS vulnerabilities) are available today but no single approach solves all the loopholes. After investigating this area, we have been motivated to propose an efficient approach to prevent persistent XSS attack by applying pattern filtering method. In this work, along with necessary background, we present case studies to show the effectiveness of our approach.**

Keywords—cross-site; filtering; pattern; persistent; scripting; vulnerability; web

I. Introduction

Cross-Site Scripting (XSS) is a well-known computer security vulnerability typically associated with web applications, which enables malicious attackers to inject client-side script into web pages viewed by other users [1]. Cross-Site Scripting attack was first discussed in Computer Emergency Response Team (CERT) advisory back in 2003 [2].

We see that a huge number of vulnerabilities plague today's web applications. While there are many security flaws that are exploited by the attackers to achieve various kinds of objectives, XSS is often ignored. Proper attention to XSS attacks should be given as it is very prevalent nowadays. XSS is basically a type of injection problem in which malicious scripts are injected into a trusted web site. Attackers useCross-Site Scriptingto execute script in the victim's browser and the malicious script can access any cookie, session token, or other sensitive information storedby the browser. According to White-hat security, 43% of web applications have XSS vulnerabilities [3]. Hence, some kind of efficient mechanism is needed to deal with such security flaw which could be a great threat for web users. This is the main motivation behind this work.

The rest of the paper is organized as follows: Section II talks about the basic operational method of XSS, Section III notes down the types of XSS, Section IV presents our proposed mechanism with examples, explanations, and outcomes of various experiments based on that; and finally, Section V concludes the paper with future directions of research. The goal of this work is to provide the readers with sufficient information about XSS and its mitigation through our proposed mechanism.

II. How Cross-site Scripting Occurs

Whenever some untrusted data are taken by an application and then that are sent to a web browser without proper validation and escaping, Cross-Site Scripting flaws could occur. *Escaping* technically means to take the data someone may already have and help secure that prior to rendering it for the end user. Using XSS, the attackers can execute such types of scripts in the victim's browser that those can hijack user sessions, deface web sites, or redirect the user to malicious sites. Hence, XSS attack allows an attacker to acquire the personal credentials of a legitimate user and possibly, impersonate the same user during the time of interaction with a specific website.

Often a web site may contain a script that returns a user's input (some parameter values) in a HTML (HyperText Markup Language) page. In such cases, if proper sanitizing is not used, XSS attack is possible to launch. To clarify a bit, *sanitization* is the process of removing sensitive information from a document or other medium, so that it may be distributed to a broader audience. Other technical meaning is the removal of malicious data from user input, such as form submissions.

If there is any such flawed mechanism (as noted above) of getting user input; for instance, an input consisting of JavaScript code to be executed by the browser when the script returns this input in the response page (without proper *sanitization*) - in this case, it is possible to make links to the site where one of the parameters consists of malicious JavaScript code. This code then will be executed by a user's browser in the site context, granting it access to cookies that the user has for the site, and other windows in the site through the user's browser.

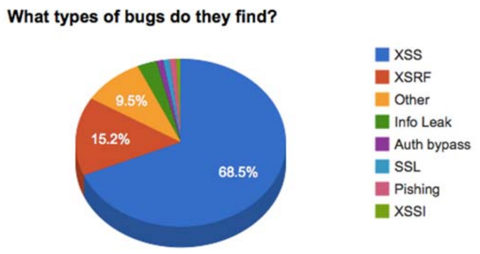


Figure 1. Google vulnerability reward program's statistics (from [5]).

Given the severity of the threat, XSS was ranked second in OWASP (Open Web Application Security Project) Top 10 Most Critical Web Application Security Risks report 2010 and third in OWASP Top 10 Most

Critical Web Application Security Risks report 2013 [4]. According to Google vulnerability reward program's statistics [5], XSS is the most reported issue (see Figure 1). A typical cross-site scripting scenario is depicted in Figure 2 [6].

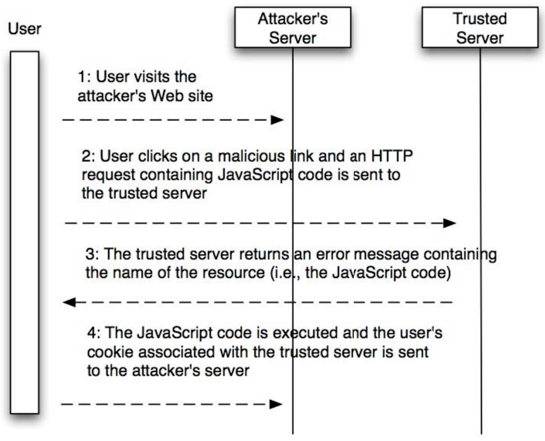

User
Attacker's Server
Trusted Server
1: User visits the attacker's Web site
2: User clicks on a malicious link and an HTTP request containing JavaScript code is sent to the trusted server
3: The trusted server returns an error message containing the name of the resource (i.e., the JavaScript code)
4: The JavaScript code is executed and the user's cookie associated with the trusted server is sent to the attacker's server

Figure 2. An example scenario of cross-site scripting (from [6]).

III. Types of Cross-Site Scripting

There are three major types of XSSs: (i) *persistent*, (ii) *non-persistent*, and (iii) *DOM-based*.

A. Persistent XSS

Persistent XSS, which is also known as Stored XSS or "*Type-I XSS*" occurs when a web application gathers input from a malicious user, and after that, stores that input in a data-store for possible later use [7]. If the stored input is not properly filtered, that malicious data will appear to be a part of the website and will be run within the user's browser under the privileges of the web application. This kind of XSS does not need a malicious link to be exploited. Whenever a user visits a web page with a stored XSS, a successful exploitation occurs.

Persistent XSS is often difficult to detect and could cause serious harm to the system. This type of XSS is considered more harmful than other types as an attacker's malicious script is used automatically, without any need to individually target victims or lure them to a third-party website. For instance, an attacker could be seemingly an innocuous blog user who leaves a malicious script on a blog's comment field of a vulnerable blogging web application. Anybody who visits the blog with the malicious script, his browser would be affected.

B. Non-persistent XSS

Non-persistent attacks are also called reflected XSS attacks. When a web application is vulnerable to this type of attack, it will pass invalidated input sent through requests to the client. A typical example is that the attacker engineers social behavior of the user or attracts a user to load an offending URI (Uniform Resource Identifier) on his browser, which then executes the offending code using the credentials of the user.

C. DOM (Document Object Model)-based XSS

DOM-based XSS is also sometimes called "*Type-0 XSS*". Some similarities are there in this type with *non-persistent* type, but in this case, the JavaScript payload does not have to be echoed back from the web server. Often it is like, simply the value from an URL (Uniform Resource Locator) parameter is echoed back onto the page on the fly when loading an already resident JavaScript. It occurs when the XSS vector executes as a result of a DOM (Document Object Model) modification on a website in a user's browser. Here, the HTTP response does not change on the client side however, the script executes in malicious manner. This exploit only works if the browser does not modify the URL characters [8]. This is recognized as the most sophisticated and least-known type of XSS. As many application developers simply do not understand how it works, it can occur in many occasions.

IV. The Proposed Approach: Experiments and Outcome

To defend against persistent Cross-Site Scripting attacks, a simple task has to be performed for input filtering: Any data from the input must be transformed or filtered in a way that it is not executed by a browser if sent to it. To avoid XSS, developers must sanitize the user's input before storing it in the database. The conceptual model of filtering persistent XSS vulnerabilities is designed based on the analysis of the results of various experimentations.

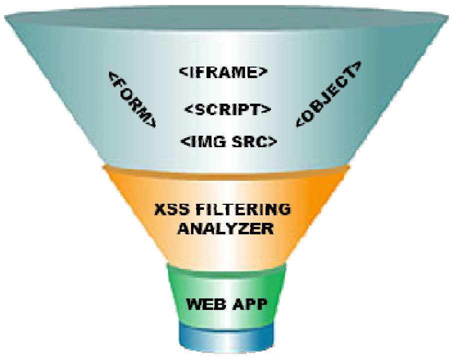


Figure 3.Conceptual model of the persistent XSS filter.

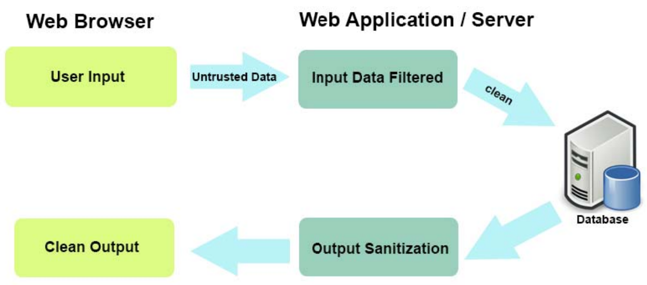


Figure 4. Process of filtering persistent XSS.

Figure 3 shows the conceptual model to prevent persistent XSS attacks and the process of filtering persistent XSS attack is illustrated in Figure 4. Figure 4 depicts that inputs from user are taken from the web browser as untrusted data, which go through a filtering process to get a "*clean*" status. These clean data are stored in the database to generate clean output from that after output sanitization.

A. Filtering Event Handlers

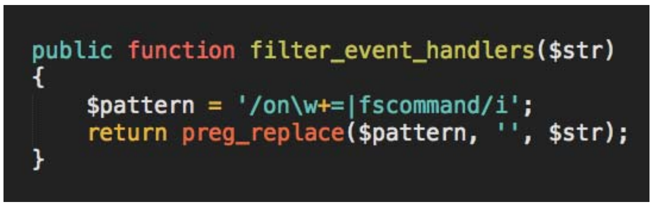
Event handlers are JavaScript codes that are not added inside the `<script>` tags, but rather inside the HTML tags, that execute JavaScript when something happens, such as pressing a button, moving mouse over a web link, or submitting a form[9]. When any event occurs, the function that is assigned to an event handler runs. Some examples of well-known event handlers are: `onError` (when loading of a document or image causes an error), `onClick` (if someone clicks on a form), `onLoad` (attacker executes the attack string after the window loads), `onMouseOver` (cursor moves over an object or area). Examples of XSS vectors using this method are [10]:

- `<img/src=`%00` onerror=this.onerror=confirm('XSS') ?>`
- `<///style///><span %2F onmousemove=alert&lpar;1&rpar;>SPAN`
- `<input/onmouseover="javaSCRIPT&colon;confirm&lpar;1&rpar;"`
- `<img/&#09;&#10;&#11; src=`~` onerror=prompt(1)>`
- `<iframesrcdoc='&lt;bodyonload=prompt&lpar;1&rpar;&gt;'>`

These kinds of XSS vectors do not require using any variants of "`javascript:`" or "`<SCRIPT...`"for the attacker to accomplish the XSS attack. By applying pattern of regular expression as noted below, we can filter these kinds of XSS vectors.

`/on\w+=|fscommand/i`

When an attacker submits or injects XSS vector into a web application that implements or uses this pattern filtering, that web application will filter out the event handler in the input submitted by attacker by replacing it with *null*. Thus, the XSS payload would become invalid. Figure 5 shows how this filter can be used in a PHP (PHP: Hypertext Preprocessor) class.



```
public function filter_event_handlers($str)
{
    $pattern = '/on\w+=|fscommand/i';
    return preg_replace($pattern, '', $str);
}
```

Figure 5. PHP method for filtering event handlers.

Explanation:

1. **on** matches the characters on literally; case insensitive.
2. **\w+** match any word character [a-zA-Z0-9]. *Quantifier*: Between one and unlimited times, as many times as possible, giving back as needed.
3. = matches the character = literally
4. **fscommand** matches the characters *fscommand* literally; case insensitive. (Attacker can use this when executed from within an embedded Flash object)
5. **i** modifier: insensitive. Case insensitive match ignores case of [a-zA-Z].

B. Filtering Data URI

The data URI (Uniform Resource Identifier) format is pretty simple and was presented in RFC 2397 [11]. The basic format of data URI is as follows [12]:

`data:[<mimetype>][;charset=<charset>][;base64],<encoded data>`

Data URI is a self-contained link that contains document data and metadata entirely encapsulated in the URI. 'data:' URI, being entirely self-contained, does not include a filename. When presented with 'data:' URI with MIME (Multipurpose Internet Mail Extensions) types that trigger the save dialog like '*application/octet-stream*', browser attempts to save the URI content as a file on the local file system.

The use of some keywords in the user input has been blacklisted in our web application - keywords like for instance, *JavaScript*, *alert*, *script, round brackets*, *double quotes*, and *colon*. Generally, `<script>` tag must be used to execute a JavaScript. Here, attacker cannot use the `<script>` tag because, the application validates user input against specific keywords. Hence, to execute a JavaScript, attacker tries using a data URI. In this attempt, attacker can now inject the following payload:

-
  ```
  <object
  data="data:text/html;base64,PHNjcmlw
  dD5hbGVydCgiWFNTIik7PC9zY3JpcHQ+"></
  object>
  ```

-
  ```
  <object
  data="data:image/gif;base64,R0lGODlh
  EAAJANUrAB4aGyAcHfHx8fb19fb29vX19fLy
  8lZTVCklJkNAQOXk5fX09C0qKkVCQ/T09J2b
  m////5mXmOnp6SUhIqSioz05OuDf4IeFhVRR
  Uj87PFVSUyomJ/39/a6sre/u7lBNTWhlZjk1
  NrCur9nY2cPCw/f393p4eB4bG7++vl9dXSMf
  IP///wAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAACH/C1hNUCBEYXRhWE1QPD94
  cGFja2V0IGJlZ2luPSLvu78iIGlkPSJXNU0w
  TXBDZWhpSHpyZVN6TlRjemtjOWQiPz4gPHg6
  eG1wbWV0YSB4bWxuczp4PSJhZG9iZTpuczpt
  ZXRhLyIgeDp4bXB0az0iQWRvYmUgWE1QIENv
  cmUgNS4zLWMwMTEgNjYuMTQ1NjYxLCAyMDEy
  LzAyLzA2LTE0OjU2OjI3ICAgICAgICAiPiA8
  cmRmOlJERiB4bWxuczpyZGY9Imh0dHA6Ly93
  d3cudzMub3JnLzE5OTkvMDIvMjItcmRmLXN5
  bnRheC1ucyMiPiA8cmRmOkRlc2NyaXB0aW9u
  IHJkZjphYm91dD0iIiB4bWxuczp4bXA9Imh0
  dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC8i
  IHhtbG5zOnhtcE1NPSJodHRwOi8vbnMuYWRv
  YmUuY29tL3hhcC8xLjAvbW0vIiB4bWxuczpz
  dFJlZj0iaHR0cDovL25zLmFkb2JlLmNvbS94
  YXAvMS4wL3NUeXBlL1Jlc291cmNlUmVmIyIg
  eG1wOkNyZWF0b3JUb29sPSJBZG9iZSBQaG90
  b3Nob3AgQ1M2IChNYWNpbnRvc2gpIiB4bXBN
  TTpJbnN0YW5jZUlEPSJ4bXAuaWlkOkUxQjk5
  RTZERkQ2RjExRTNBRkM5OUE5QUFGMTY0NDlG
  IiB4bXBNTTpEb2N1bWVudElEPSJ4bXAuZGlk
  OkUxQjk5RTZFRkQ2RjExRTNBRkM5OUE5QUFG
  MTY0NDlGIj4gPHhtcE1NOkRlcml2ZWRGcm9t
  IHN0UmVmOmluc3RhbmNlSUQ9InhtcC5paWQ6
  RTFCOTlFNkJGRDZGMTFFM0FGQzk5QTlBQUYx
  NjQ0OUYiIHN0UmVmOmRvY3VtZW50SUQ9Inht
  cC5kaWQ6RTFCOTlFNkNGRDZGMTFFM0FGQzk5
  QTlBQUYxNjQ0OUYiLz4gPC9yZGY6RGVzY3Jp
  cHRpb24+IDwvcmRmOlJERj4gPC94OnhtcG1l
  dGE+IDw/eHBhY2tldCBlbmQ9InIiPz4B//79
  ```

/Pv6+fj39vX08/Lx8O/u7ezr6uno5+bl5OPi
4eDf3t3c29rZ2NfW1dTT0tHQz87NzMvKycjH
xsXEw8LBwL++vby7urm4t7a1tLOysbCvrq2s
q6qpqKempaSjoqGgn56dnJuamZiXlpWUk5KR
kI+OjYyLiomIh4aFhIOCgYB/fn18e3p5eHd2
dXRzcnFwb25tbGtqaWhnZmVkY2JhYF9eXVxb
WllYV1ZVVFNSUVBPTk1MS0pJSEdGRURDQkFA
Pz49PDs6OTg3NjU0MzIxMC8uLSwrKikoJyYl
JCMiISAfHh0cGxoZGBcWFRQTEhEQDw4NDAsK
CQgHBgUEAwIBAAAh+QQBAAArACwAAAAAEAAJ
AAAGQ8CVcEgsGo/FhYFQIBAhhcFAUFqhGoDT
hKSgCCKAQCA1EnoyKtWmokqkVaDi5fBhvEMY
TQdpMSEeEkIcSCsOIoSIREEAOw=="
/></object>

The executions of the injected payloads are shown in Figure 6 and Figure 7.

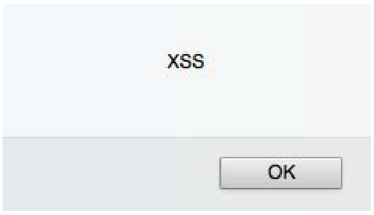


Figure 6. Data URI HTML JavaScript.

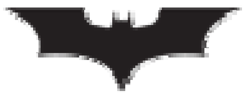
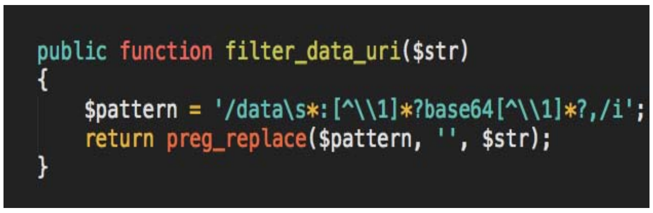


Figure 7. Data URI HTML image.

Data URI attack can be prevented by applying the following regular expression pattern filtering in our web application (see Figure 8):

```
/data\s*:[^\\1]*?base64[^\\1]*?,/i
```



```
public function filter_data_uri($str)
{
    $pattern = '/data\s*:[^\\1]*?base64[^\\1]*?,/i';
    return preg_replace($pattern, '', $str);
}
```

Figure 8. PHP method for filtering data URI.

Explanation:

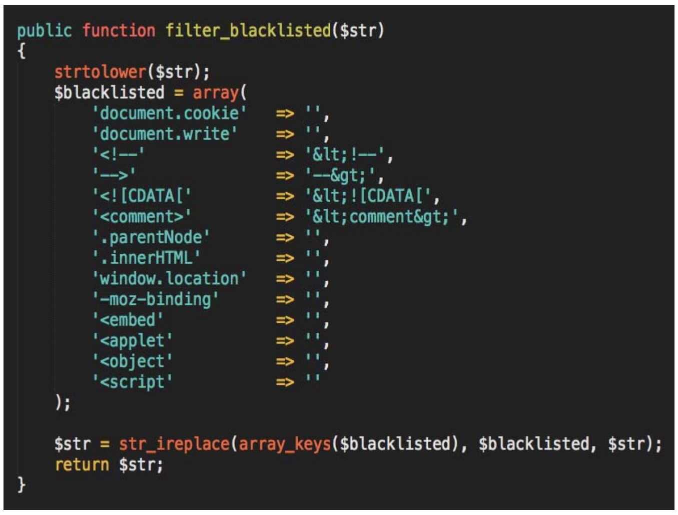
1. **data** matches the characters data: literally case insensitive
2. **\s*** match any white space character [\r\n\t\f] *Quantifier*: Between zero and unlimited times, as many times as possible, giving back as needed.
3. **:** matches the character: literally
4. **[^\\1]*?**match a single character not present in the list below:
   Quantifier: Between zero and unlimited times, as few times as possible, expanding as needed.
   \\ matches the character \ literally
   1 the literal character 1
   base64 matches the characters base64 literally (case insensitive)
5. **[^\\1]*?** match a single character not present in the list below:
   Quantifier: Between zero and unlimited times, as few times as possible, expanding as needed.
   \\ matches the character \ literally
   1 the literal character 1, matches the character, literally.
6. **i** modifier: insensitive. Case insensitive match ignores case of [a-zA-Z].

By implementing pattern filtering as mentioned above, our web application would be free from this kind of attack.

C. Filtering Insecure Keywords

An insecure keyword is in a list of known bad data to block illegal content from being executed. We have to filter pattern that should not appear in user input. If a string matches this pattern, it is then marked as *invalid*; then, will be removed or replaced with equivalent words. Some of the insecure keywords are (examples are shown in Figure 9):

- document.cookie
- document.write
- window.location
- innerHTML
- parentNode
- <applet
- <embed
- <script



```
public function filter_blacklisted($str)
{
    strtolower($str);
    $blacklisted = array(
        'document.cookie'   => '',
        'document.write'    => '',
        '<!--'              => '&lt;!--',
        '-->'               => '--&gt;',
        '<![CDATA['         => '&lt;![CDATA[',
        '<comment>'         => '&lt;comment&gt;',
        '.parentNode'       => '',
        '.innerHTML'        => '',
        'window.location'   => '',
        '-moz-binding'      => '',
        '<embed'            => '',
        '<applet'           => '',
        '<object'           => '',
        '<script'           => ''
    );

    $str = str_ireplace(array_keys($blacklisted), $blacklisted, $str);
    return $str;
}
```

Figure 9. PHP method for filtering insecure keywords.

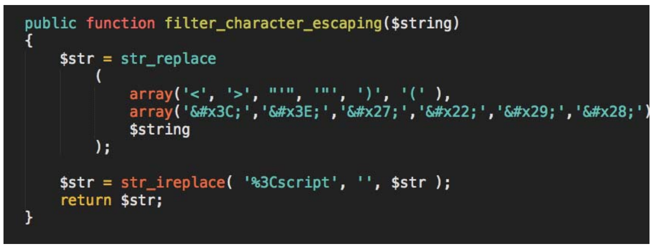
Basically we would compile a listing of all the blacklisted words, and then verify that the input received from the user is not one of the blacklisted words.

D. Filtering Character Escaping

To help prevent XSS attacks, web application needs to ensure that all variable outputs in a page are encoded before being returned to the end user. Preventing XSS attacks means to substitute every special character used in these attacks. We can escape dangerous characters by using the &#sequence followed by its character code. Table I shows is a list of common escape codes.

TABLE I. List of common escape codes [13].

Display	Numerical Code	Hex Code
"	&#34;	&#x22;
#	&#35;	&#x23;
&	&#38;	&#x26;
'	&#39;	&#x27;
(	&#40;	&#x28;
)	&#41;	&#x29;
/	&#47;	&#x2F;
;	&#59;	&#x3B;
<	&#60;	&#x3C;
>	&#62;	&#x3E;



```
public function filter_character_escaping($string)
{
    $str = str_replace
        (
            array('<', '>', "'", '"', ')', '(' ),
            array('&#x3C;','&#x3E;','&#x27;','&#x22;','&#x29;','&#x28;')
            $string
        );

    $str = str_ireplace( '%3Cscript', '', $str );
    return $str;
}
```

Figure 10. PHP method for filtering character escaping.

For instance, if an attacker injects `<script>alert ("XSS")</script>` into a variable field of our web application, the character such as '<', '>' will be substituted with &#x3C and &#x3E; then, would display the script as part of the web page but the browser will not execute the script. Thus, we could prevent our web application from XSS.

PHP method for filtering character escaping is shown in Figure 10.

E. Filtering Common Words in XSS Payload

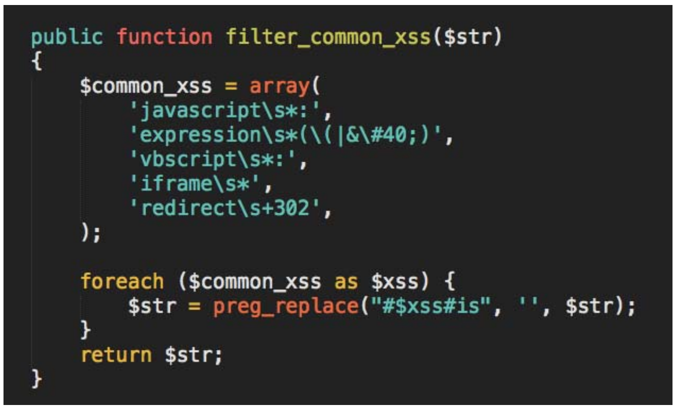
XSS attacks are constantly evolving. Everyday new vectors are created. By implementing common words in XSS payload, we can defend our web application from malicious code injection. As an example, the payload below will be filtered by our web application.

- `<iframe %00 src="&Tab;javascript:prompt(1)&Tab;" %00>`

Explanation:

1. **javascript** matches the characters JavaScript literally case insensitive.
2. **\s*** match any white space character [\r\n\t\f] *Quantifier*: Between zero and unlimited times, as many times as possible, giving back as needed.
3. **:** matches the character : literally
4. **i** modifier: insensitive. Case insensitive match ignores case of [a-zA-Z].

PHP method for filtering common XSS words is shown in Figure 11.



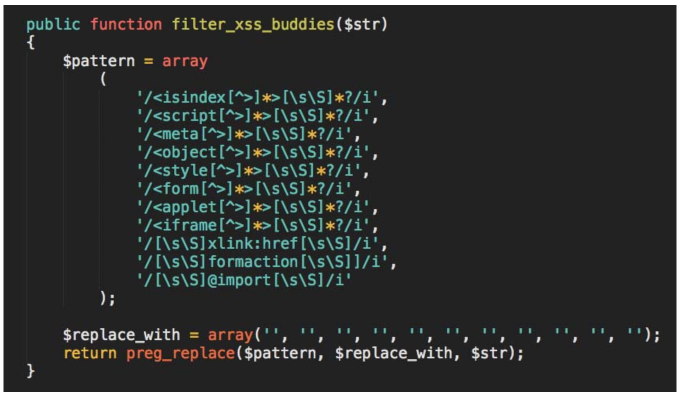
```
public function filter_common_xss($str)
{
    $common_xss = array(
        'javascript\s*:',
        'expression\s*(\(|&\#40;)',
        'vbscript\s*:',
        'iframe\s*',
        'redirect\s+302',
    );

    foreach ($common_xss as $xss) {
        $str = preg_replace("#$xss#is", '', $str);
    }
    return $str;
}
```

Figure 11. PHP method for filtering common XSS words.

F. Filtering XSS Buddies

The friend of my enemy is my enemy. HTML elements such as `<isindex>`, `<meta>`, `<form>`, `<object>`, `<style>`, `<script>`, and etc., should be filtered out from user input toprevent our web application form XSS code injection. Here is an example of XSS payload filtered by this method:

- `<iframe src=j&Tab;a&Tab;v&Tab;a&Tab;s&Tab;c&Tab;r&Tab;i&Tab;p&Tab;t&Tab;:a&Tab;l&Tab;e&Tab;r&Tab;t&Tab;%28&Tab;1&Tab;%29></iframe>`
- `<form><button formaction=javascript&colon;alert(1)>CLICKME`



```
public function filter_xss_buddies($str)
{
    $pattern = array
        (
            '/<isindex[^>]*>[\s\S]*?/i',
            '/<script[^>]*>[\s\S]*?/i',
            '/<meta[^>]*>[\s\S]*?/i',
            '/<object[^>]*>[\s\S]*?/i',
            '/<style[^>]*>[\s\S]*?/i',
            '/<form[^>]*>[\s\S]*?/i',
            '/<applet[^>]*>[\s\S]*?/i',
            '/<iframe[^>]*>[\s\S]*?/i',
            '/[\s\S]xlink:href[\s\S]/i',
            '/[\s\S]formaction[\s\S]]/i',
            '/[\s\S]@import[\s\S]/i'
        );

    $replace_with = array('', '', '', '', '', '', '', '', '', '', '');
    return preg_replace($pattern, $replace_with, $str);
}
```

Figure 12. PHP method for filtering XSS buddies.

As presented in Figure 12,

1. **<isindex** match the characters <isindex literally; case insensitive
2. **[^>]*** match a single character not present in the list below
   Quantifier: Between zero and unlimited times, as many times as possible, giving back as needed.
   >a single character in the list > literally case insensitive
   > matches the characters > literally
3. **[\s\S]*?** match a single character present in the list below
   Quantifier: Between zero and unlimited times, as few times as possible, expanding as needed
   \s match any white space character [\r\n\t\f]

\S match any non-white space character [^\r\n\t\f].

G. *Outcome of Our Experiments*

We have tested and evaluated a series of XSS attack scenarios. A collection of XSS Cheat Sheets [14], [15], [16] was used. All of those were filtered and sanitized effectively. It has been proved that the proposed mechanism is effective enough in filtering out various malicious XSS vectors.

V. Conclusion and Future Work

Using XSS attack, it is possible to steal or manipulate victim's sessions and cookies, which may be used to impersonate a legitimate user of a system. Hence, it is very important to filter user input to secure any web application. Our main concern in this paper is that the applications should perform input and output filtering to achieve an appropriate level of protection for its users. With our various experiments and investigation, the proposed XSS attack prevention model has been found to be very effective. However, each time, various kinds of tricks and techniques are being devised [17] – hackers are not sitting idle, in fact their attacks are getting more and more sophisticated as the time moves ahead. Therefore, there is no 100% guarantee that our solution presented in this paper would work in the coming years in the same way. Given this fact, the best way to prevent or defend against XSS attack is to code the web applications with security in mind and to use proper escaping mechanisms in the right places. In this case, it is better never to trust the data coming from the user. Every bit of data must be validated, filtered, and escaped on output.

As our future work, we plan to investigate thoroughly how our prevention model can be extended and applied to non-persistent and other types of Cross-Site Scripting.

Acknowledgment

This work was supported by NDC Lab (Networking and Distributed Computing Laboratory), KICT, IIUM.

References

[1] M. Dabbour, I. Alsmadi, and E. Alsukhni, "Efficient Assessment and Evaluation for Websites Vulnerabilities Using SNORT," International Journal of Security and Its Applications, Vol. 7, No. 1, pp. 7-16, January 2013.

[2] "Cert advisory ca-2000-02.Malicious HTML Tags Embedded in Client Web Requests," February 2000.

[3] (2013, May). *Website Security Statistics Report. WhiteHat Security* [Online]. Available: https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf

[4] (2013). *OWASP Top 10 - 2013: The Ten Most Critical Web Application Security Risks. OWASP* [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[5] (2012, December 16) *Google's Vulnerability Reward Program* [Online]. Available: http://www.nilsjuenemann.de/2012/12/news-about-googles-vulnerability-reward.html

[6] E. Kirda, C. Kruegel, G., Vigna, and N. Jovanovic, "Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks," Proceedings of the 2006 ACM symposium on Applied computing (SAC'06), pp. 330-337.

[7] (2014, April 04) *Cross Site Scripting* [Online]. Available at: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[8] A. Klein. (2005) *DOM Based Cross Site Scripting or XSS of the Third Kind. Web Application Security Consortium* [Online]. Available: http://www.webappsec.org/projects/articles/071105.shtml [last accessed on 10 July 2014]

[9] *Understanding "event handlers" in JavaScript* [Online]. Available: http://www.javascriptkit.com/javatutors/event1.shtml

[10] A. Javed. (2014) *100 XSS Vectors* [Online]. Available: http://www.bugsheet.com/cheat-sheets/100-xss-vectors-by-ashar-javed

[11] L. Masinter. *The "data" URL scheme*, RFC 2397, IETF, August 1998.

[12] (2012, September 16) *HTML5 media and data URIs* [Online]. Available: http://www.iandevlin.com/blog/2012/09/html5/html5-media-and-data-uri

[13] *ASCII to Hex* [Online]. Available: http://www.asciitohex.com

[14] *XSS Cheat Sheet* [Online]. Available: http://seguretat.wiki.uoc.edu/index.php/XSS_Cheat_Sheet

[15] *Complete Cross site Scripting (XSS) cheat sheets* [Online]. Available: http://www.breakthesecurity.com/2012/02/complete-cross-site-scriptingxss-cheat.html

[16] (2014, September 09) *XSS Filter Evasion Cheat Sheet* [Online]. Available: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

[17] D.A. Kindy, and A.-S.K., Pathan, "A Detailed Survey on various aspects of SQL Injection in Web Applications: Vulnerabilities, Innovative Attacks and Remedies," International Journal of Communication Networks and Information Security, Vol. 5, No. 2, pp. 80-92, August 2013.