XSS Attacks and Possible Defenses

Travis Starkson
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
stark451@morris.umn.edu

ABSTRACT

Cross-site scripting (XSS) is one of the most serious threats to web security. It has been one of the top web security vulnerabilities on Open Source Web Application Consortium (OWASP) for over a decade. Much research has gone into this field to defend browser users against XSS, but with each new advancement attackers quickly find ways to circumvent them, staying abreast of each new countermeasure. In this paper, we look at some of the previous defenses against XSS and their flaws. More specifically we discuss how previous solutions are either too slow or can be circumvented by some types of XSS attacks. Then we introduce more recent solutions and how they improve upon from older solutions. These more recent solutions are not the answers to all XSS attacks, or even some older types of XSS attacks. They are, however, one step in a cycle that will help make the next solutions stronger, so understanding these solutions is a crucial step in understanding how to combat XSS attacks.

1. INTRODUCTION

As the Internet continues to grow and develop, so do the methods web site attackers use. Much research has gone into web-based attacks and ways to combat them. In particular cross-site scripting (XSS) is one of the most serious threats in web security. It has been said to be one of the most serious vulnerabilities to web security for over a decade [1]. XSS attacks can vary from harmless pranks to the attacker acquiring personal information such as bank account numbers.

One example of an XSS attack is in 2008 when an attacker redirected the link to Obama's web page so that instead it went to Hillary Clinton's. This example, of course, is on the lighter side of XSS attacks, and the person who hijacked the site admitted to the prank [3]. Recently, on eBay it was discovered that a script could be injected into the product listing pages. By doing this the attacker can steal the user's credentials, resulting in a much more serious attack than the previous. This is not the first report of XSS attacks on eBay [4]. There are numerous accounts of XSS attacks on major web sites all the time. Some of the most prominent sites include Twitter, Facebook, Youtube, and eBay [8][4].

Exploiting an XSS vulnerability involves three steps. The first is that the attacker uses some means to deliver a payload to a vulnerable site, or in other words a malicious script. Then the payload is used by the site when generating a web page sent to the user's browser in response to a request. If the site is vulnerable to the type of XSS attack that was sent, then in the third step the user's browser would execute the attacker's injected code in the page returned by the site [7]. Thus the attacker can gain elevated access-privileges to sensitive page content, session cookies, and a variety of other information maintained by the browser on behalf of the user [8]. The amount of steps required to execute an XSS attack may seem too numerous and difficult to implement, but XSS attacks account for over 84% of all security vulnerabilities documented by Symantec, an American technology company, in 2007[8]. In this paper we define what an XSS attack is, what were previous solutions to defend against it, and talk about some more recent solutions. We discuss how previous solutions either are too slow or can be circumvented by some types of XSS attacks. We then describe how some more recent solutions, as well as how they improve upon previous solutions.

2. DEFINITIONS

Here are a few definitions of terms used in the paper.

- Cross-site scripting (XSS): A type of injection attack on the server-side in which malicious script is inserted into a returned web page. This is usually done using a scripting language like JavaScript [5][7].
- Sanitize: Sanitizing a request means that the XSS defense tries to remove or replace characters that can be considered part of a malicious script [7].
- Attack vector: This is the form or path by which an XSS attack inserts malicious script.
- Attack surface: The set of attack vectors that pass for a browser.
- HTTP requests/responses: An HTTP request is a request from a browser to a web site for information on the site in order to display its contents. An HTTP response is the sites response to the browsers HTTP request, of which it sends the appropriate information, specified on the request, back to the browser.

UMM CSci Senior Seminar Conference, December 2014 Morris, MN.

- Scriptless attack: This form of XSS attack is performed by not using any form of script-based code. Instead it is carried out by using other web applications such as Cascading Style Sheets (CSS). An example of this is the use of a CSS property called `content` which in combination with the value property `attr` can extract sensitive attribute values like password-field values.

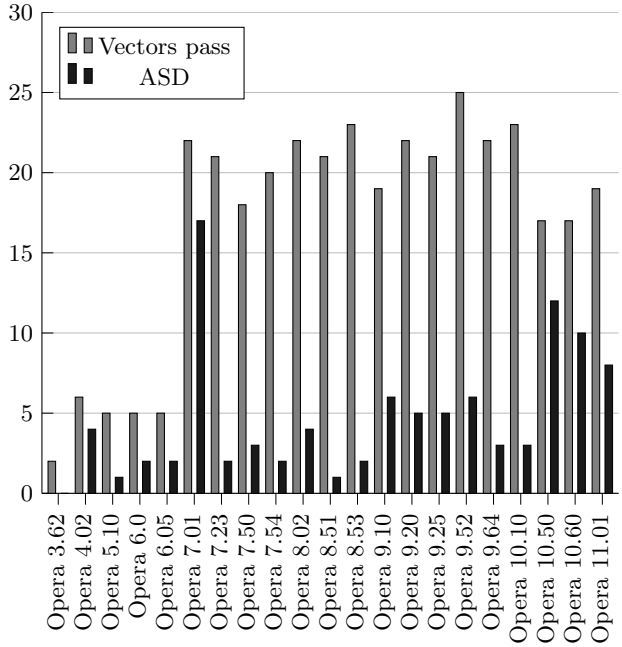

Figure 1: Opera regression [1].

3. PREVIOUS SOLUTIONS

In this section, we will look at how XSS attacks have been dealt with in the past. Some well-known defenses against XSS attacks include NoScript, IE8 (Internet Explorer 8), noXSS, and XSSAuditor. All of these defenses have been found with faults that allow a number XSS attacks. Some of these vulnerabilities have been addressed in newer solutions which provide a better defense against XSS attacks, and shall be discussed later in section 4.

NoScript and IE8 are filters that utilize regular expressions to identify the pressence of JavaScript in HTTP request parameters and attempts to sanitize them before submission. The problem with this is that it can lead to overly strict filtering. Which can prevent some web pages from being loaded or interfere with the functionality of the web page [7]. NoScript also incurs many of false positives [7]. While IE8 induces fewer false positives, it does so by disabling itself for same-origin requests, which are requests to a web page previously visited safely. This, however, causes more false negatives to occur due to the potential that malicious script may have been added since the last visit [2]. NoScript's high false positive rate is due to its use of regular expressions, which have to be stringent enough to handle the worst case scenarios. Since both filters sanitize the outgoing request, they can set the parameters of the outgoing request to a very different set of values from what they were originally. This can make the request fail on the server-side, corrupt data stored on the server-side, or return an incorrect page [7]. Also, just the sheer number of regular expressions casues issues for maintenance. NoScript has about 40 non-trivial regular expressions invovled in detection and sanitation [7]. Even with this complex set of regular expressions, parsing tricks can still bypass the filter. This is partially due to the fact that NoScript uses a same-origin request system similar to IE8's. A parsing trick is adding certain characters into an attack script that allows it to bypass a filter. One such trick is the character / which can be used to do some complex modifications such as <a<img/src/onerror=alert(1)//< [2]. In this example the attack can only be carried out due to the same-origin response system NoScript has in place. With this fault in place, this allows an attack to use the <img/src> tag to post a hyperlink on the site. The / character can be interpreted as a closing JavaScript marker, which is done so in this example. Also, neither filter can detect scriptless attacks [5].

XSSAuditor and noXSS are filters that use exact substring matching to match reflected content. Exact sub-string matching checks the content in an HTTP response with the content in the HTTP request that generated the response, which tries to look for entire malicious scripts [7]. noXSS achieves a high fidelity rate, the rate at which it follows the protocols set within itself in real-time, but at the cost of slower performance (14% overhead load time on average) [2]. Also, noXSS does not handle HTML entity encoded JavaScript URLs. An HTML entity is a character reference using an ampersand followed by the reference name and a semicolon. These are used instead of the actual character since a browser can mistake them for specialy operations, like < and > could be mistaken as tags by the browser. This allows a hacker to bypass the filter by inserting a full-page hyperlink which, if the user clicks anywhere on the page, allows the attacker to run arbitrary script as the target site [2]. XSSAuditor is implemeted on the client-side and has improved upon most of the faults that noXSS and IE8 present. However, XSSAuditor cannot handle partial-script injections, which can alter the structure of an existing script on the web page. This in turn can allow attackers to insert malicious script onto the page. Partial-script injection is a type of XSS attack that changes part of a web pages script in order for the web page to run malicious script, which is quite different from its counterpart whole-script injection. With this inability to handle partial-script injections, XSSAuditor can not determine the beginning and end of an injected string when sanitizing [7]. XSSAuditor also can not handle attacks that are not script-based, such as an attack carried out by CSS. Using non script-based web applications, attackers can bypass the filter and inject malicious code [5].

3.1 Progression of Web Browser Defenses

In this section, we state that browser providers do not have a systematic regression strategy (reducing the amount of XSS vulnerabilities as they are found) in developing new versions of their browers. In other words a consistant reduction in XSS vulnerabilities is not observed for each new version of a browser. To validate this, [1] tests two hypotheses:

H1. *Browsers belonging to two different families have different attack surfaces. In other words, they are not sensitive to the same attack vectors. This first hypothesis is crucial to*

vector/browser	Chrome 11.0.696.68	IE 8.0.6001.19048	Opera mobile 11	Opera11.11 rev2109	IE mobile	Safari Mac OSX	iPhone 3GS	Android 2.2	Firefox 5 Android	Firefox 8.0a1	IE 6.0.2900.2180	Firefox 2.0.0.2	Netscape 4.8	IE 4.0.1	Opera 4.00
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1
5	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1
6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
8	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
9	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
10	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
11	1	1	0	0	1	1	1	1	1	1	1	1	0	1	0
12	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
14	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
18	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
21	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
22	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
31	1	0	1	1	0	1	1	1	0	0	0	1	0	0	0
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
33	1	1	0	0	1	1	1	1	1	1	1	1	0	1	0
34	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
35	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
36	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
37	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
38	0	1	0	0	1	0	0	0	0	0	1	0	1	0	0
39	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
41	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
42	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
43	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
44	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
45	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 1: Results on vectors 1 through 42 [1].

vector/browser	Chrome 11.0.696.68	IE 8.0.6001.19048	Opera mobile 11	Opera11.11 rev2109	IE mobile	Safari Mac OSX	iPhone 3GS	Android 2.2	Firefox 5 Android	Firefox 8.0a1	IE 6.0.2900.2180	Firefox 2.0.0.2	Netscape 4.8	IE 4.0.1	Opera 4.00
45	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
46	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
47	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
48	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
49	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
50	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
51	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
52	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
53	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0
54	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0
55	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
56	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
57	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
59	1	0	1	1	0	1	1	1	1	1	0	0	0	0	0
60	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
61	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
62	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
64	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
65	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
66	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
67	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
68	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
69	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
70	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
71	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0
72	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
73	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
74	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
75	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
77	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0
78	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
79	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
81	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
82	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
83	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
84	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
85	0	0	0	0	0	0	0	1	1	1	0	0	1	1	0
86	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
87	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2: Results on vectors 42 through 84 [1].

understand whether there is a shared security policy between web browser vendors headed against XSS attacks to protect clients against web attacks. [1]

H2. *Web browsers are not systematically tested w.r.t their sensitivity to XSS vectors. This second hypothesis explores whether there is a clear continuity or a convergence in the attack surface of a given web browser over time. The validation of this hypothesis would mean that web browser providers do not have a systematic regression strategy for improving the robustness of their web browser from one version to the next one. [1]*

To prove H1, the researchers executed 84 different XSS attack vectors against three types of browsers: modern/recent, mobile, and legacy versions. All of the attack vectors were obtained from either a variety of XSS cheat sheets or were created from an n-cube test generator. The test generator used is a combination using HTML4 tags and property sets with Javascript calls in order to produce 6 usable vectors in testing. The vectors choosen represent a large variety of dissimilar XSS attack vectors [1]. Then, each attack vector was tested against a variety of browsers from each of the three types of browsers. The results are represented in Tables 1 and 2. Each block with a 1 represents a threat exposure to that particular XSS attack vector while a 0 represents no vulnerability [1]. To demonstrate the variety of the XSS attack vectors we shall look into a few test cases. Vectors #3 and #6 are basic <script> tag based XSS with various payloads. Vectors #12 and #13 are <body> tags based with an `Onload` event set to execute the payload. #17 is a <script> tag with double brackets to evade basic filters. Vectors #53, #54, and #59 are based on HTML5 tags and properties.

Using these tables to explore vulnerabilities we see that some web browsers have very similar 'signatures', but after a more careful consideration we see that they are unique. This information supports H1 by showing that each browser has its own unique threat exposure [1]. Also, if we take a look at Table 3, which does not include all test cases, we see that between the desktop and the mobile versions of the exact same browsers there is still a difference in threat exposure [1].

Figures 1, 2, and 3 represent the evolution over time for 3 different browsers. Figure 1 shows the regression of their set of attack vectors with the evolution of Opera. Figures 2 and 3 show the exact same for Firefox and Internet Explorer respectively. The attack surface distance (ASD) is the number of differences between two browsers threat exposures, the current and previous versions, which is shown in black. The XSS attack vectors that pass are shown in gray. We see a choatic and unstabilized regression in these browser's

browser	0	1	2	3	8	9	10	13	14	16	26	28	30	34	35	36	40	42	43	47	48	50	51	53	56	69	70	72	74
Opera 11 Desktop	1	1	1	1	0	1	1	1	1	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	0	0	1
Opera 11 Mobile	1	1	1	1	0	1	1	1	1	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	1
Firefox 4 Desktop	1	1	1	1	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0	0	1	1	0	0	0	0
Firefox 4 Mobile	1	1	1	1	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0	1	0	1	1	1	1	0	0	0	0

Table 3: Mobile versus Desktop versions [1].

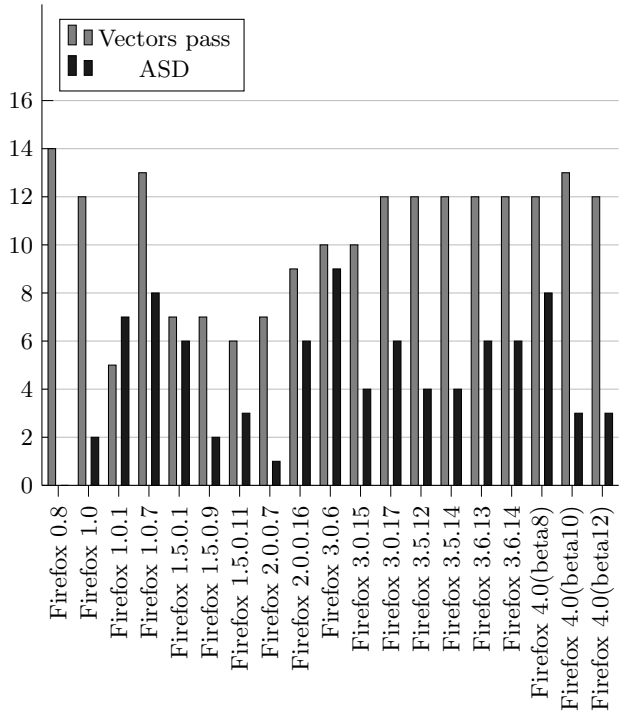

Figure 2: Firefox regression [1].

testing [1]. If we take a look at Figure 1 we see that Opera 10.50 and 10.10 have about the same number of passing attack vectors (23 and 17 respectively), but the attack surface distance is quite high (12)[1]. This shows a strong instability between these two minor versions. It also reveals a lack of systematic regression testing between versions. If we take a look at Figures 2 and 3 we see the same occurrence happening where we have similar number of attack vectors passing, but a large difference in attack surface distance. In Figure 2, when we look at Firefox 2.0.0.16 and Firefox 3.0.6 we see that the number of passing vectors is very similar, but the attack surface distance is quite high (9). With this we can say that H2 is validated [1].

4. POSSIBLE SOLUTIONS

In this section, we discuss some recent solutions to defend against XSS attacks. First we will look at XSSFilt and how it improves upon Google Chrome's XSSAuditor and other methods. Then we look at a browser patch that can help in defense against some scriptless attacks. Section 4.3 discusses other possible solutions.

4.1 XSSFilt

XSSFilt is a client-side defense that utilizes an approximate string matching algorithm. This allows it to detect whole and partial-script injections. Before that though, XSSFilt parses the URL into parameters. It then excludes any parameters that cannot possibly have JavaScript code or HTML. This includes parameters containing less than 8 characters and parameters containing only alphanumeric characters, underscores, spaces, dots and dashes [7]. Then it utilizes the approximate string matching algorithm. The algorithm checks the length of the parameters and the script of the web page. If the parameters are longer than the script, it searches within the parameters for the script for a whole-script injection. If the script is longer than the parameter, it searches the script for the parameters for partial-script injection [7].

The drawbacks that XSSFilt has in comparison to XSSAuditor is that it runs significantly slower. XSSAuditor uses exact substring matching, which has a linear time. Exact substring matching essentially has a set of strings that it searches for within the script of a web page that are considered harmful. This in turn is quite different from XSSFilt's approximate string matching which searches for strings that match a pattern approximately. XSSFilt also has a higher false positive rate in comparison to XSSAuditor. This is due to XSSAuditor using exact string matching which is stricter than XSSFilt's approximate string matching. Because of this, XSSAuditor's possibility for coincidental matches for an entire string is smaller than that for its substrings [7]. XSSFilt, however, can deal with application-specific sanitations better than exact string matching. Also, XSSFilt can deal with partial-script injections while exact string matching is unable to determine them [7]. Table 4 shows how many XSS attack vectors out of two different cheat sheets that XSSFilt, XSSAuditor and NoScript prevent. Noscript did well against both cheat sheets, but this is due to the fact that the attack vectors tested used basic JavaScript commands. NoScript excels at stopping these types of attacks, but can be bypassed by using other JavaScript commands that require more skill and effort [7].

4.2 Scriptless Attack Defenses

Now we introduce some defense mechanisms that can be used against some types of scriptless attacks. According to [5], Content Security Policy (CSP) can be used to help reduce the potential harm that malicious injected code causes. CSP can also restrict access to undesirable non-script-based files such as CSS and Scalable Vector Graphics (SVG). CSP is great at eliminating the execution of script-based attacks like JavaScript. However, it is insufficient in covering a wide variety of scriptless attacks. CSP is a step in the right direction since it is able to eliminate some side channels, which are data leaks established either by accident by the site or by an attacker, used in scriptless attacks and a few other attack vectors [5].

In [5] the authors propose a solution to a type of scriptless attack labeled double-clickjacking. Double-clickjacking is a form of attack that allows the attacker to leverage pop-up windows and detached views to aquire data leakage ex-

Dataset	XSSFilt	XSSAuditor	NoScript
xssed	399/400	379/400	400/400
cheatsheet	20/20	18/20	20/20

Table 4: Number of attack vectors protected against [7]

ploits and perform clickjacking attacks. Clickjacking attacks decieve the user to click on something different from what they perceive, such as an image or a link that instead runs malicious script when clicked on, potentially allowing confidential information to be revealed. Their solution is a patch created for Firefox that expands on the window object by adding two properties: `isPopup` and `loadedCrossDomain`. Both of these properties return boolean, true/false, values and can only be accessed in read-only mode by a web site at any time. As the names suggest, `isPopup` is true when the GUI, the human-computer interface, window represented by the current DOM, the structure logic for docuements/programs, window object is in a detached view, while `loadedCrossDomain` is true if the current DOM window object was loaded in a cross-domain. Cross-domain means loading information from a site other than the one currently loaded in the windows. With this ability, websites can detect if they are being loaded in a detached view, which can allow it to mitigate different scriptless attack vectors [5].

4.3 Server-side and Hybrid Solutions

In this section we describe a server-side defense against cross-site scripting attacks called SWAP, developed by Peter Wurzinger and his team, and a hybrid approach to managing information traffic flow[9][6].

SWAP operates on a reverse proxy, or a proxy server that retrieves information on behalf of a client, that checks each web response with a so-called JavaScript detection component. Within this component, SWAP puts together a fully functional modified web browser that takes note of any script within the content. In order to find malicious script, SWAP modifies the hosted web application, which encodes all legitimate script calls into unparsable identifiers (script IDs). Thus this approach hides such calls from the JavaScript detection component, and all remaining scripts are assumed to have been injected. If this happens, then SWAP does not send the response, but instead notifies the client of the attempted XSS attack. To determine legitimate script calls, they utilize a modified version of Firefox in the JavaScript detection component. They adapted their version of Firefox in three different positions in order to not overlook legitimate script. The first change notifies the user of scripts that are executed automatically on the loading page. Second, this change notifies of event handlers, most of which are executed on user interactions. The third change allows it to get notified of JavaScript URL link scripts, which are opened when clicked upon. SWAP, however, has a few drawbacks to it. The first is that with the inclusion of an extra step between the client and server, the time it takes to relay the information to its target is increased. This creates a performance penalty due to the need of having to render each page before delivery to the client. SWAP is also limited by only detecting JavaScript XSS attacks [9]. The biggest drawback to SWAP is that since it runs on a reverse proxy it is only useful for non-encrypted content, and in the past few years there has been a push to encrypt as much content as possible.

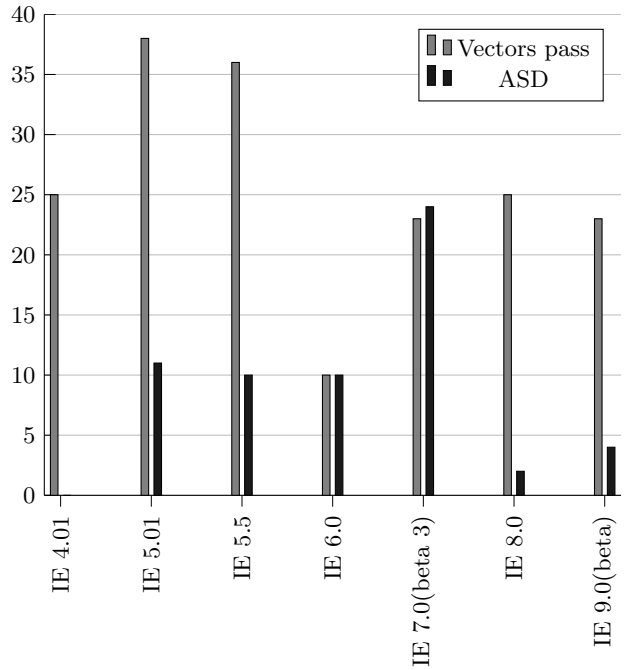

Figure 3: Internet Explorer regression [1].

The hybrid approach that [6] uses static and dynamic approaches to label values during an execution of a program. The static approach to labeling analyzes a program before execution to determine whether all executions are secure, while the dynamic approach monitors a program's execution to determine whether it is secure. This system tracks explicit flow, or clearly demonstrated flow, by updating labels on modified values that influence the final result. When dealing with implicit flow, or information controlled through a programs control flow, they use something that is quite similar to a dynamic control dependence. This is more generally an operation that starts at the predicate (the affirmed statements) to its immediate post-domniator (a statement that the flow of information must pass through in order to reach the end). At runtime they use a stack of security contexts, or a stack of authentication and authorization actions, which determines the label currently in the implicit flow. With this stack, the history of the security contexts can be labeled and monitored. This approach is able to detect XSS attacks when it is a whole-script injection. This approach is unable to deal with scriptless XSS attacks or partial-script attacks. Currently, this hybrid approach runs quite slowly compared to other forms of information flow control.

5. CONCLUSION

In this paper we looked at what an XSS attack is. We discussed less recent solutions to defend against these attacks, but found out that these defenses have flaws in them that can be exploited. We found that a good majority of previous defenses have trouble with partial-script and scriptless injections. Defenses that have this issue include NoScript, IE8, noXSS, and XSSAuditor. Also, we looked at whether

web browser providers have been following a linear regressive path in combating XSS. We found patterns that suggest that web browser providers do not follow a systematic regression strategy in the developement of new versions of their browsers. Due to this neglect in testing previous XSS attacks that were solved in a previous version have the potential to pass in a future version. Next we discussed possible solutions in combating some of the flaws that previous solutions present. We looked at XSSFilt which is a client-side defense that utilizes approximate string matching to locate injected malicious script. Then we looked at some partially completed scriptless attack defenses, which means they do not cover a wide range of scriptless attack vectors, but only specific attack vectors. We then considered a server-side defense called SWAP and a hybrid approach to information traffic flow. These most recent solutions are not enough to combat XSS attacks completly. Each of the solutions previously stated have drawbacks and short comings that prevent them from being able to completly prevent XSS attacks. Additionally, due to the ever evolving nature of XSS attacks new forms of attacks will continue to be created that will be able to circumvent these solutions. In conclusion, continuous research must be done to improve upon these solutions due to the ever evolving nature of XSS attacks.

6. REFERENCES

[1] E. Abgrall, Y. Le Traon, S. Gombault, and M. Monperrus. Empirical investigation of the web browser attack surface under cross-site scripting: An urgent need for systematic security regression testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 34–41, March 2014.

[2] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 91–100, New York, NY, USA, 2010.

[3] L. Digman. Obama site hacked; Redirected to Hillary Clinton, 2008.

[4] S. Gold. eBay downplays significance of 'old school' XSS attack on its auction portal, 2014.

[5] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless attacks: Stealing the pie without touching the sill. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 760–771, New York, NY, USA, 2012.

[6] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for javascript. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, pages 9–18, New York, NY, USA, 2011.

[7] R. Pelizzi and R. Sekar. Protection, usability and improvements in reflected xss filters. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 5–5, New York, NY, USA, 2012.

[8] Wikipedia. Cross-site scripting — Wikipedia, The Free Encyclopedia, 2014.

[9] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. Swap: Mitigating xss attacks using a reverse proxy. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, IWSESS '09, pages 33–39, Washington, DC, USA, 2009.