A New Practical and Collaborative Defense Against XSS Attacks

Prateek Saxena  
University of California, Berkeley

Yacin Nadji  
Department of Computer Science  
Illinois Institute of Technology

Dawn Song  
University of California, Berkeley

ABSTRACT

Several remote attacks on the web today exploit the insecurity that comes with embedding untrusted data in trusted content. A specific type of cross site scripting (XSS) attack – reflected XSS attacks – are the most common of these, and plague even the most popular web sites today. Traditional defenses against these attacks rely on filtering user input, which was been shown to be quite difficult in practice. Filtering is entirely reactive to security threats, and more often than not, reacting to new exploits doesn’t happen. Current approaches are effective, but are often difficult to perform widespread implementation.

We propose the concept of client-side tainting of user generated information. Tainted data is quarantined by a novel delimiter scheme which allows for flexible policy enforcement. We confirm that our approach, in addition to being simple to implement, requires only changes to the client browser and defends against over 2000 XSS attacks. We also analyze stored XSS attacks, and what we would need to alter in our approach to combat them as well.

General Terms

Security

Keywords

cross-site scripting, web security, tainting

1. INTRODUCTION

Cross-site scripting attacks have become an immense threat to the millions of Internet users today, who regularly access banking, commercial, government and personal services on a daily basis. Recent reports have shown that in the second half of 2007, that XSS constitutes an alarming 84% of 11,000 reported vulnerabilities. Less than 547 (or 0.5%)[1] have been fixed by the end of the 2007. Clearly, this shows that dealing with XSS has been hard and as seen with other classes of attacks, vulnerabilities remain open for several days even after their disclosure.

So far, defenses against XSS have largely relied on server side filtering and sanitization. Despite the client data being the target of the XSS attacks, there are few solutions that are completely client based. Better mechanisms that allow web site administrators to eliminate XSS bugs in design or at runtime are constantly evolving, but are ineffective for the victim if these are not routinely adopted by the majority of web sites for one reason or another. Therefore, we emphasize that it is invaluable to have defenses that allow a worried user to protect herself in a world where web site administrators take several days to diagnose/fix their websites.

Today, when a user is tempted to click on a web link, she has no option but to completely trust that the web server the link visits has deployed state-of-the-art anti-XSS defense.

XSS vulnerabilities have plagued even the most well-known of the web sites which take appropriate measures by sanitization of untrusted user input. One reason for this is that XSS elimination is seen largely as sanitization problem, today. Sanitization is hard to perform, because of several reasons. First, the attacker has several ways to encode his input giving many polymorphic attack variants possible for a single point of vulnerability. Second, there is a diverse heterogeneity in the actual client environments. Different browsers have different ways in which they parse web pages. Interpretation of the encoding of the web page varies across browsers, even browser versions, making the server side sanitization inconsistent with the client side rendering. Explain with examples. It is natural to ask if we can can utilize what is observed on the client and protect the user against a vulnerable server.

With these limitations in mind, we propose a new high level idea for preventing XSS as a policy enforcement problem on untrusted data on the client side. The idea is to identify parts of the web page data which is derived from untrusted source before it is processed by the client browser. Then, the client web browser can effectively limit the actions that it performs on behalf of untrusted input. This approach overcomes much of the limitations of the server side sanitization based approaches mentioned above. Precisely identifying which data should be marked untrusted is really the responsibility of the web server; but, due to the practical problem of slow vulnerable server patch deployments, we have to approximate the data marking on the client side. In this paper, we show that it is practical to defend against reflected XSS attacks, by correlating the activity seen on the client side only. We also show that a more comprehensive defense against both stored as well as reflected attacks, based on the idea and show that it could work in a backward compatible manner.

A reflected XSS attack is one in which a web server embeds
untrusted input from the user into its output pages, without appropriate sanitization to filter out executable code. Often the attacker can exploit a vulnerable site to serve unintended script code in its output page. As a result, many defenses have focused on identifying the symptoms of the attack, such as identifying cookie stealing. XSS attacks can be devastating in other ways as well. XSS attacks vectors now, are sometimes not based on JavaScript injection, instead they rely on injecting iframes or Flash executables. For instance, in one case, frauds sent phishing mails which use a specially-crafted URL to inject a modified login form (using an iframe) onto the bank’s login page. The vulnerable page is served over SSL with a bona fide SSL certificate given to the vulnerable bank. Security indicators such as yellow “https” URL bars in the web browser, as well as techniques targeted to filter out active JavaScript code, provide no guarantee against invasive XSS attacks that violate the integrity of the HTML page. Show figure.

Server side defense techniques have largely been the focus of previous techniques. Automatic escaping/quoting features in languages such as PHP that make input data safe for use in HTML output queries using special escape sequences, are some of the earliest mechanisms for combating XSS attacks. However, since there are many contexts where untrusted data could be used, and different level of content richness in web input is permitted by web applications, the one-size-fits-all approach did not scale. Moreover, it left holes in the sanitization. For instance, htmlspecialchars does not stop an attack that injects HTML attributes. See Figure 1 for an example of HTMl attributes.

```html
<input type=text value= value => awesome onmouseover=alert(1)
<input type=text value=awesome onmouseover=alert(1)
```

Figure 1: Attribute Injection

Several other techniques such as input filtering can be applied to input variables with the aim of remove dangerous symbols or ensuring that some sanitization is done on user input. All of these are problematic because they do not give a strong guarantee that malicious action would finally not occur on the client side. The context in which untrusted data is used is often unknown at the point of sanitization; victim client environments may vary from the assumptions made by the sanitization code about the same (such as assumed charset encoding for the untrusted input). These factors make XSS mitigation a practically, serious problem. Server side tainting based solutions against XSS have somewhat relied on blacklisting characters in tainted outputs, which has obvious problems. Plus, policies vary based on where the untrusted data is emitted in the output, which makes policy configuration a headache for the web developer.

We evaluate our defense technique against 850 real world XSS attacks on popular web sites, and provide a analytical evaluation of the capabilities of the defense as well. We show that the technique can prevent 800 out of 850 attack samples, with no false positives. Moreover, our techniques incur less than 5% overheads on the server and the clients in our experiments.

2. MOTIVATION

2.1 Current Defenses

The fundamental problem behind reflected XSS attacks is that today’s web programming model places the complete onus of web input filtering/sanitization on the developer.

Server Side Taint-enhanced Policies.

This approach allows policies to focus on untrusted input, but still relies on blacklisting dangerous characters that is highly based on context and susceptible to heterogeneity in clients. Besides, it adds performance overheads on the server which may cause the server to be bottlenecked.

Server Side Sanitization.

A big problem with focusing all defenses on the server side is the heterogeneity of the way browsers render pages. Table 1 shows based on the list of XSS attacks from [2] how many are suitable attack vectors for each browser. Without the server knowing with 100% accuracy what browser a client connects with, it becomes cumbersome to successfully filter all input to allow for both correct rendering and security against XSS attacks.

<table>
<thead>
<tr>
<th>Browser</th>
<th>Number of Attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Explorer 7.0</td>
<td>49</td>
</tr>
<tr>
<td>Internet Explorer 6.0</td>
<td>89</td>
</tr>
<tr>
<td>Netscape 8.1-IE Rendering</td>
<td>89</td>
</tr>
<tr>
<td>Netscape 8.1-Gecko Rendering</td>
<td>47</td>
</tr>
<tr>
<td>Firefox 2.0</td>
<td>45</td>
</tr>
<tr>
<td>Firefox 1.5</td>
<td>50</td>
</tr>
<tr>
<td>Opera 9.02</td>
<td>61</td>
</tr>
<tr>
<td>Netscape 4</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1: Browser Specific XSS Attacks

Sensitive Information Flow Tracking.

The first problem with purely client side sensitive information flow tracking, is that this is legitimately allowed by applications and not restricted to attack cases. Without a separate mechanism to deteremine individual policies, it is hard to enforce by tracking sensitive information without numerous false positives.

Second, the inherent assumption in the work by Vogt et al.[5], is that the end goal of an XSS attack is to covertly steal sensitive information such as cookies. However, XSS attacks can really be used for several other arbitrary actions, such as setting the price of objects to zero, injected/overlaying new iframes and so on. Third, they too focus on the JavaScript based attacks. If only sensitive information such as cookies are protected, attackers are still free to inject arbitrary flash and HTML elements to navigate frames, reset passwords or other attacks to users’ accounts.

Browser “noexecute” Policies.

Simply disallowing all scripting is an unreasonable solution as well. For example, in [3] developers are given the opportunity to prevent certain nodes in the DOM from having execute permission, particularly in the context of JavaScript. This would not prevent against Flash code injection, or injection of HTML elements, such as iframes. If iframes are allowed to be arbitrarily inserted by attackers, exploits such as those that stole bank account information[4] are still vul-
nerable.

Document Structure.
A common theme in reflected XSS attacks is that in order to execute the arbitrary code, the structure of the document must be altered in order to allow the client’s browser to render and execute the code properly; this is not true in all cases, however. In the event that user input is immediately reflected back to the user without being nested in an HTML element, or inside a tag (e.g. `<input type="user-input"`), the document’s structural integrity does not need to be broken in order to execute an XSS attack.

There are numerous ways that an attacker can attempt to execute arbitrary code by breaking the document’s structure:

1. Breaking out of string literals:
   In such attacks, there is improper sanitization to prevent the untrusted input from escaping the string literal scope. In this example, USERDATA is untrusted user data first closes the string quotes, and then exploits the context in which it is being embedded to insert a malicious event handler for the mouse over event.
   ```html
   <input type="text" value="USERDATA" size=10>
   <input type="text" value="abc"
   onmouseover=javascript:sendCookie();" size=10>
   ```

2. Breaking out of HTML nodes in the DOM:
   In such attacks the untrusted input escapes the scope of the HTML node that it is embedded in. In the shown instance below, the untrusted code adds a new script tag which gets interpreted by the browser as an embedded script tag.
   ```html
   <input type="text" value=USERDATA >
   <input type="text" value=abc><script>
   sendCookie(document.cookie); </script> }
   ```

3. Inclusion/Insertion:
   Consider a vulnerable search engine that echos a user search request back in a “No documents found” error message. If it fails to sanitize the input search query, the attacker may simply include a script tag (or a embedded flash movie tag) to execute code, without having the untrusted content span outside the enclosing node’s content.

4. HTML response splitting:
The defenses against these attacks are limited:

- **Server side sanitization.** Sanitization is hard because it varies by where the input is emitted out. This is a well understood problem. A second reason is the heterogeneity in client environments and browser specific quirks make it hard for the server to determine how the web server will interpret a given untrusted data. For example, in 2007, Google was found susceptible to two severe XSS vulnerabilities that arose because the server response lacks charset encoding enforcement. Specifically, script, when given a bad URL as argument returned a 404 Not Found page with missing character set encoding:

   A UTF-7 encoded attack payload is inappropriately filtered on the server side, as Google web server assumed a different charset encoding than UTF-7. On the client side, Microsoft IE, when configured for auto-selecting character encoding, uses UTF-7 automatically if it finds a UTF-7 string in the first 4096 characters of the response’s body and therefore interprets the malicious payload. This example illustrates how web server sanitization may be inconsistent with the actual rendering on the client side.

- **Client side policies.** A new shift in XSS defense paradigm is to have the server specify policies and the client enforce, due to the above mentioned problems. For instance, BEEP[3] allows whitelisting all JavaScript code that executes on a web page, and, marking certain regions of the web page as “no-execute” which allows the browser to ensure that no scripts are executed in those regions. This has three limitations:

  1. Specifying policies, such that untrusted content can not escape the scope of the policy-confined node, is hard. The authors of BEEP noticed this, and proposed a fix. Unfortunately, as we show now, even the fixed version is insecure and can be evaded by a careful attacker. BEEP suggests that...
  2. Executing untrusted code is not limited to JavaScript. Other mime types and active content like Flash, Quicktime can use scripting as well. BEEP does not nullify attacks based on these vectors.

- **Client side sensitive information flow tracking.** Vogt. et al[5] showed a purely client-side defense that relies on tracking cross-domain transfer of sensitive data such as cookies. However, as the authors found.

  Proposed defenses are limited: BEEP can deal with IFRAME injection or Flash. Sensitive information flow tracking. Painting on server side and blacklisting. Server side whitelist generation.

  Important thing is to ensure that the attacker can not break out of the confines of the web application.

  Document markup languages have no mechanisms that allow HTML document generators to specify constraints on the intended structure of the untrusted input. Web content delivery languages, such as HTML, are not equipped with mechanisms that preserve web document’s integrity.

3. **OUR APPROACH**

Our approach is to identify parts of the web page data which is derived from an untrusted source before it is processed by the client browser. With this information successfully obtained, we can force the client browser to handle this untrusted data in a way to mitigate XSS attacks. A perfect approach would be to have the server identify these data and pass the information to the client for enforcement. Unfortunately, due to the difficulty associated with rolling out
new security features at the server level, focusing entirely on the client is a more practical solution.

We are able to, with high accuracy, approximate which data should be untrusted by inspecting the information sent by the client to the server in POST/GET requests. Our proposed implementation uses a proxy server to collect the data sent to the server, and insert delimiters around the tainted data based on the server’s response. This information is then returned to the client for policy enforcement. An illustration of this can be seen in Figure 2. This allows for a practical solution that only requires individual client involvement, however, can be easily extended into a full client/server cooperative approach for additional confidence in the delimiter insertion.

4. IMPLEMENTATION

Our implementation of client-side XSS mitigation relies on two main portions: a way of inserting delimiters to mark tainted data and a browser capable of parsing HTML with inserted delimiters and negating the attack.

4.1 Proxy-based Delimiter Insertion

After properly isolating any potentially malicious input, the browser is able to handle each tainted input before the document is rendered, allowing numerous policies to be enforced as desired. In our sample implementation, we use the policy that any tainted user input should be treated as a string.

We wrote a small Ruby web proxy based on the WEBrick HTTP Server Toolkit [6]. When given an HTTP request that contains user queries (i.e. GET/POST requests) the proxy surrounds all instances of the tainted data with delimiters, allowing for policy enforcement. For example, the snippet of PHP code in Figure 3 is something you would find on a typical dynamic website. The client sends information as a GET request (http://site.com/?user=username), this information is processed on the server, and returned to the user by echoing out an HTML page. Using our proxy, the HTML snippet in Figure 4 would be returned to the user, leaving the browser to enforce the policy based on the inserted delimiters.

Delimiter Specification.

The delimiters are randomly generated sequences of seven numbers and lowercase letters (called the delimiter key) surrounded by the ASCII control characters start of text (^B) and end of text (^C), respectively. The ASCII control characters signify to the browser the beginning and end of the delimiter key to aid in parsing. The delimiter serves as a way of preventing attackers from breaking out of the document structure and altering the HTML page. There are $36^7$ different possible keys, meaning an attacker has a $1.276 \times 10^{-29}$% chance of correctly guessing the key and initiating a reflected XSS attack.

4.2 Browser Modifications

We modified the KDE HTML Library, KHTML, to support our additions and tested them against the browser Konqueror. The KHTML library consists of ~178,000 lines of C++ code. Due to its small size, and the practicality of our approach, we only needed to add 190 lines of code to implement the basic policy of “treat all tainted user input as a string”. As KHTML is parsing an HTML document, it builds a tree of the Document Object Model (DOM) where each node represents an HTML element. When the policy encounters text that is surrounded by our delimiters, it treats it all as one node, which forces the browser to treat everything inside it as benign text. Figures 5, 6, and 7 illustrate this.

Figure 5 shows a DOM tree for a very basic web page containing a form. Assume the left-most <input> tag corresponds to the basic PHP form in Figure 3. If we submit the input value ""<script>alert(1)</script><iframe src="evil.com"></iframe>"", we end up with the DOM tree in Figure 6. Without proper sanitization, an attacker is allowed to alter the structure of the tree, allowing new nodes to be created and evaluated by the browser. If this same

![Figure 2: Proxy Delimiter Insertion](image-url)

![Figure 3: Sample PHP Page](image-url)

![Figure 4: Sample Taint Delimiter Rendering](image-url)
scenario is rendered with our client-side tainting protection, however, the entire user input is confined to one node and the attack is unable to launch. The generated DOM tree can be seen in Figure 7.

Currently, we only have implemented a policy to treat all tainted user input as a string, however, the approach lends itself to more flexible policies which we intend to investigate at a later time.

5. EVALUATION

In order to properly evaluate our approach we needed a way to automate the proxy/client experience and generate a large enough dataset to show how effective our approach was. We modified our proxy to, when given an HTML page and a sample attack string, generate a new HTML file with the delimiters inserted. We could then automatically have our modified browser visit these websites, to see if the attacks were nullified. In accordance with our original hypothesis, our approach mitigated a large chunk of reflected XSS attacks, and with continued work, could easily increase in effectiveness.

5.1 Effectiveness

To verify our approach’s effectiveness, we constructed a dataset of 5,343 real-world XSS attack samples from XSSed.com[7]. The website contains archived HTML of each attack allowing individuals to learn from the mistakes of others, and also keeps track of each website to determine if the exploit has been fixed. To avoid unimportant websites, we targeted only XSSed’s “special” website exploits, which limits us to popularly ranked and government websites. After downloading each exploited website and its accompanying attack, we ran our modified proxy to produce delimited versions that a client would be served if visiting the actual website with the proxy.

Of the 5,343 websites analyzed, 56% of the attacks were nullified. 4% of the websites had improperly placed delimiters, showing that although guessing for the location of tainted input is imperfect, it is an incredibly close approxi-
5.2 Overhead

When using the proxy to browse, we saw an overall increase in rendering of 6%. Switching over to a server/client cooperative approach, in which the server places the delimiters and the client enforces the policy, would lessen the time wasted at the client’s proxy level. Unfortunately, the performance hit for adding the delimiters for every page serve— which would be different for each unique query— could prove to be too high for widespread implementation.

When simply viewing a pre-delimited HTML file, we observed no significant increase in rendering time. This further reinforces our approach as being a practical solution to defending the majority of reflected XSS attacks.

6. CONCLUSIONS

This paper has presented a practical, easy to implement and effective way to defend against reflected XSS attacks. The homogeneity of web browser rendering schemes and the difficulty in forcing a drastic change in the way web servers generate responses make our client only approach easy to implement. XSS attacks primarily are focused on harming the user, so it makes sense to put the defense in the hands of the client. Coupled with the general slow response of server side XSS exploit patching, allowing the user control over their own browsing security is a natural choice.

6.1 Continued Work

We intend to continue with this project in order to enhance the effectiveness of our approach, and better understand the potential of extending it to deal with other XSS attack classes.

- Improved Effectiveness
  Our approach, while effective, could be substantially improved. Browser’s often try to render malformed HTML documents, which makes altering the parsing scheme more difficult. In addition to this, data returned from the web server immediately undergoes tokenization and parsing, making it difficult to correctly handle delimiter parsing, as it may fall in between reads from the server.

- False Positives and False Negatives
  In any approach aimed improving security, the number of false positives and negatives plays a vital role in the effectiveness of a system. In our theoretical evaluation of the approach, we believe the number of false positives would be very few, depending on the policy implemented. To our knowledge, very few sites rely on user input needing to be executed as it is reflected back, making the likelihood of running into many false positives quite slim. We intend to test this functionality by writing a web crawler to compare the rendered pages using our proxy/patched browser and an unpatched browser. False negatives are more difficult to detect, as we cannot be sure of the unintended effect our approach may have on websites. An attack that executes by injecting code into the JavaScript code of the website would be unable to be detected by our approach currently. The script is only executed when the page’s script is executed, which is too late for our approach to take effect.

- Exploration of Client/Server Tandem Approach
  Having the server place the delimiters and the client enforce the policy offers a number of potential improvements. We are guaranteed that the placement of any taint on user input is correct, and we can also begin to improve our approach’s scope by mitigating stored XSS attacks. The approach is not without its problems however. Stored XSS attacks will often require different policies to be enforced, and taint must be propagated at the database level as well to ensure we do not misclassify stored information. Finally, the performance cost may be too great for popular servers, making the approach impractical for widespread implementation.

In the following weeks, we intend to further explore these in order to improve our approach.

Acknowledgments.

This work was supported in part by NSF SUPERB-CISE grant.
7. REFERENCES


