Zhen Huang
Nikolay Pavlovich Laptev
Jason Liu

FIRE-FOX XSS PREVENTION

I. INTRODUCTION

In recent years XSS attacks have become more widespread due to the growing popularity of AJAX and other dynamic web paradigms. Because more functionality is controlled by Javascript, any web vulnerability carries with itself a greater penalty for a successful attack.

XSS attacks are difficult to detect because they are often executed as a background process. Furthermore, to successfully detect an XSS attack, we must have a clear idea of what constitutes malicious code. This is a difficult task, because both attack and good code are stored on the same server and have a similar structure.

In this paper we introduce a Firefox plugin that is able to parse an html page before it is loaded and warn the user of any potential XSS attacks. The user then decides what action to take. The contribution of our application is threefold (1) Able to block XSS code before loading the page (2) Allows a finer JavaScript blocking granularity (3) Is easily extendable to defend against new XSS attacks.

We found that our plugin is able to effectively identify and prevent XSS execution without globally disabling JavaScript on client's browser. We also have found that our plugin is robust against a variety of XSS attacks.

II. CROSS SITE SCRIPTING BACKGROUND

There are two types of cross site scripting attacks. The first type is called a *reflective* attack. This attack is carried out by using the GET parameter. The malicious user constructs a URL that embeds evil script as the value of the GET variable. Often-times, the website will display the value of the GET parameter, thereby executing the malicious code. This doctored URL can then be emailed as spam to unsuspecting users. A common use for this attack usually involves theft of cached usernames and passwords. The second type of attacks is called a *stored* attack. The idea behind this attack is that the evil script is actually stored on the server and therefore is executed every time any user visits the web-page. A MySpace profile page is a perfect example where a malicious user can craft a personal web-page where evil javascript is stored, and every guest who visits this web-page is exposed to this XSS code.

Below are two cases where XSS attacks have occurred in the past:

Case Study 1: MySpace XSS Attack

MySpace, a popular social network site was attacked by a self propagating cross site script worm. The attacker was able to automatically add millions of myspace users as friends. The attacker bypassed many of MySpace's security checks and inserted a JavaScript that forced any user that executed it to send friend request to the attacker. In this case, the attacker was able to exploit vulnerability in web browsers such as Internet Explorer and Safari which allow JavaScript execution in the *style* attributes within HTML tags. This is an example of the *stored* vulnerability. The malicious script was part of the attacker's profile page on MySpace.

Example attack code:

```
<div id="mycode" expr="alert('XSS')" style="background:url('java
          script:eval(document.all.mycode.expr)')">
```

In this case, the "javascript" keyword was divided by a newline symbol to bypass one of MySpace's filters. By doing so, the attacker was able to execute any JavaScript expression within the "expr" attribute. The hacker was able to copy this malicious JavaScript to the profile of any user that viewed it, thereby spreading this JavaScript to more and more profiles. Although this attack was done as a practical joke, one could imagine a more serious attack, such as credit card theft or wire transfer fraud.

Case Study 2: Cyberregs site XSS vulnerability

The Cyberregs' home page consists of a login input box. This input box contains input fields that allow the user to enter username and password. After some investigation, the attacker was able to find that GET variable *pg* is displayed on the home page. By using this GET parameter, the attacker was able to force the website to display arbitrary source code on the login page. Any user who viewed the malicious link is at the mercy of the attack. Below is a possible URL that the attacker could use to exploit the vulnerability on Cyberregs website.

Example URL:

```
http://www.cyberregs.com/cgi-exe/cpage.dll?pg=”></form><form
action=”/cgi-exe/cpage.dll” method=”GET” onsubmit=”XSS=new Image;
image.src=’attacker_server’+document.forms(1).unmae.value
+’:’+document.forms(1).pass.value”>
```

The above URL is designed to steal the username and password values of a legitimate user. In order for this to work, the victim must click on the URL. Spam methods are used to mass delivery. In many cases, the email address is spoofed and the message appears to be from the Cyberregs website. This is an example of *reflective* vulnerability. This type of attack is particularly hard to detect for non-technical users if the value of the "*pg*" parameter is encoded in hex. The vulnerability displayed on Cyberregs is very common among other websites. By going to *www.xssed.com/pagerank*, one can find many other websites that have vulnerability similar to Cyberregs'.

III. APPROACH

To protect against XSS attacks, code matching techniques are used. If an instance of malicious code is found in the HTML, necessary action is taken to prevent that code from executing. This is a high level view of our approach.

XSS attack prevention software can be deployed on the client-side or on the server-side:

Client-side validation

Our approach deals with client-side validation. Under this approach, the validator will scan web pages that enter the browser. We built a plug-in for Mozilla FireFox web browser that validates incoming web pages and checks for potentially harmful JavaScript code. Our application will prompt the user for further action once it encounters an instance of problematic JavaScript code. By prompting the user, attacks can be stopped before they are executed without disabling Javascript globally. Our application also remembers the user's decision and automatically stops similar JavaScript from being executed in the future.

This approach stops both *reflective* and *stored* types of XSS vulnerabilities. Our approach gives user full control of script execution by scanning the incoming source code and giving warnings of any potential XSS. The *pros* and *cons* of client-side prevention are as follows:

Pros:

(1) Ability to control script execution
(2) Ability to prevent stored and reflective attacks
(3) Ability to remember user's response
(4) User does not have to disable javascript globally
(5) Easy deployment and installation

Cons:

(1) Difficult to identify malicious and benign scripts
(2) False positives
(3) Hard for the user to tell if a script is malicious by looking at the warning

Server side Validation

A second deployment location of our validator is on the web server side. In this report, we have discussed the idea of doing HTML verification of specific patterns to prevent cross site scripting. One of the difficulties with that solution is that it is hard to identify legit and malicious JavaScript code blocks. One alternative solution that we are presenting is to have the verification process moved to the server-side instead of the client-side.

By having the validation script on the server-side we are able to distinguish user scripts and server scripts easily. By using tags to distinguish between user and server code we can employ pattern matching techniques more efficiently by solely focusing on the user code. This process would scan the appropriate code blocks warning the user of any potential danger as before.

Pros:

(1) Central deployment point (could be in the form of a proxy server).
(2) By focusing on user code, which is likely to contain malicious scripts, pattern matching techniques can be deployed more efficiently.
(3) All other benefits of the client-side approach.

Cons:

(1) Server-side verification puts a burden on the response time of the web server.
(2) Uncertainty of what action to take when server discovered a malicious code block. Several possibilities exist:
    1. Server should deem this page as malicious and send a standard error page to the user requesting the page. This, however, would affect the quality of service for the innocent end user.
    2. Server could block the malicious code block by either commenting it out or “erasing” it from the page. This would potentially eliminate malicious code while still offering service to the rest of the page.
    3. Server prompts the user for what action to take.

In either *client side* or *server side* approaches, one of the difficulties is to determine the patterns that we must be searching for. Although we could program certain known vulnerabilities into the validator, it is still not a proven way to defend against all *possible* XSS attacks. On the other hand, validating a web page with a large amount of JavaScript blocks could be an annoyance to the user as well as a burden on web-page loading performance. We will discuss a few alternative methods that could alleviate some of these limitations in later sections.

IV. PREVIOUS WORK

After doing research, we found a client side program called “No Script” [2] that helps prevent cross site scripting by blocking untrusted JavaScript. This Firefox plug-in uses a white list approach. JavaScript on web pages from untrusted websites are blocked until the user confirms the website. This method could prevent XSS attacks, but it is not robust. In most of the web pages, there are limited locations where attackers could inject malicious script. To block every script on the page seems to be too restrictive. In the next section, we will propose a solution where script blocks are analyzed and only offending script blocks are called into question.

A security server that resides between the web server and the internet cloud was recently suggested [1]. This security server could be used to implement policy based security checks. This solution is similar to the server side approach that we proposed in the previous section. In particular, our approach deals with only the cross site scripting subset of the security issues. Also, our server side approach is much more tightly coupled with the web applications. One of the strength that we hope the system should have is it should leverage the knowledge of the server side code. For example, the system could explicitly tag server side blocks with special markers. When the system is going through script blocks, any block that has the special marker is skipped. Obviously, these markers are also removed before reaching to the client so no outside agent should know the existence of such markers. However, we argue that this system is not domain specific to any web application. The form of tagging could be predefined or it could be any special pattern that is provided to the system during start up. The system will search for that pattern, which it knows how to do very well, and skip the appropriate script blocks.

V. IMPLEMENTATION

We have implemented a plugin for FireFox (based on the no-script code) that is able to stop the execution of an XSS script embedded in a web-page. Our script simply warns the user about a possible XSS attack, whereby a user can choose to continue loading the page or stop its execution. Our plugin is also able to remember user's previous responses, so if a previously seen XSS attack is detected again, user's previous decision is used as an action for our script. We now demonstrate how our plugin is able to defend against three possible XSS attacks, (1) XSS attack which relies on a malicious code located in an XML file, (2) Cookie stealing attacks, and (3) Attacks which supply a malicious script in the URL of a page.

Type I

Assume that we have the following 3 lines in our HTML page:

```
<div style="-moz-binding:url('test.xml#xss'); color: blue;" ></div>
<div style="-moz-binding:url('test.xml#xss'); color: blue;"></div>
<div style="-moz-binding:url('test2.xml#xss'); color: blue;"></div>
```

Figure 1: Code block which is stored in the HTML page containing a hidden XSS script (stored in the .XML file)

The potentially malicious XML file consists of the following:

```
<?xml version="1.0"?>
<bindings xmlns="http://www.mozilla.org/xbl">
  <binding id="xss">
    <implementation>
      <constructor><![CDATA[document.write('XML Attack(2)']]></constructor>
    </implementation>
  </binding>
</bindings>
```

Figure 2: XML file containing a potential XSS code.

This is a benign XML file that only outputs a string onto a page, however clearly an attacker can supply a much more elaborate script which does more damage to the client (e.g. steals private information).

When this script is executed with our plugin enabled, we get the following result:

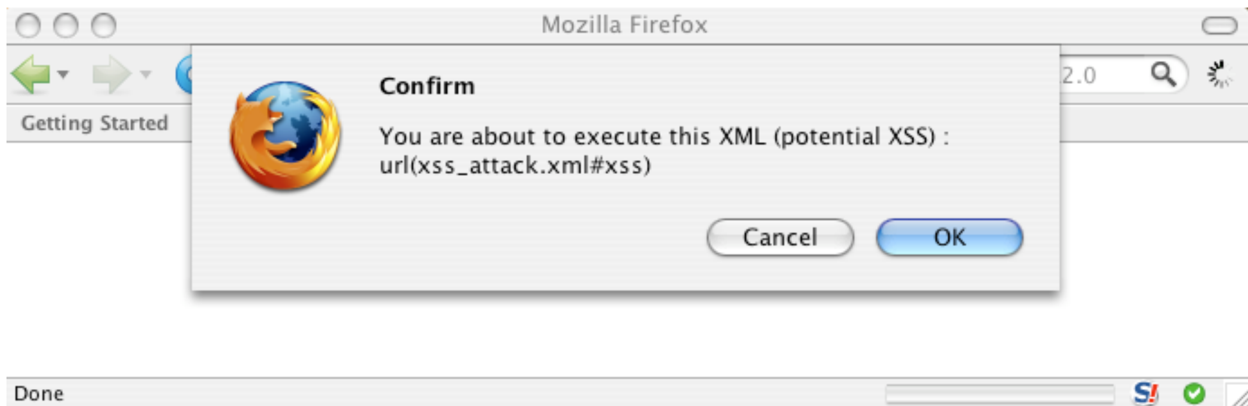

Figure 3: A warning is displayed to the user indicating a possible XSS attack hidden in the XML file.

If the user decides to click OK, the script will be executed, and what was in the XML file will display on the user's screen.

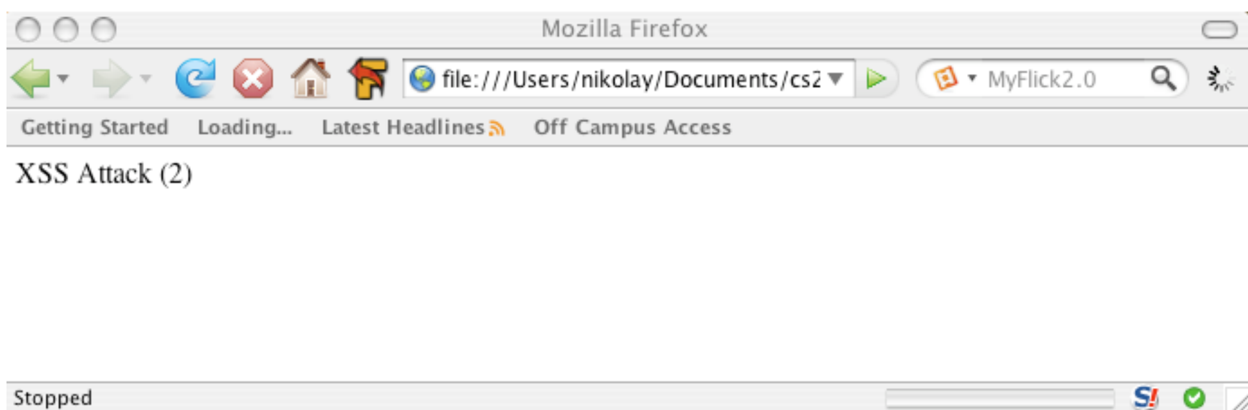

Figure 4: If the user answers 'OK' to a possible XSS warning, an XML file is executed.

In order to display the alert, we employ regular expressions which scan HTML code matching any possible XML vulnerabilities. Figure 5 presents a snippet of code where these regular expressions are used. In this particular case, we go over all of the tags in our HTML code and try to match any style tags which contain *xml* keyword as one of their properties. If we find such an HTML element, we add it to the message array which we will display to the user later in the execution. We also must keep track of where exactly in the code the found element appears (which element number) so that if the user decides against executing of the potentially dangerous HTML element, we can quickly disable it by accessing its element number and setting that element to *null.*

```
processScriptStyleElements: function(document, sites) {
    var count=0;
    sites.styles = document.getElementsByTagName("*");
    var i;
    for (i=0; i< sites.styles.length; i++) {
        myRe = new RegExp ("xml", "g");
        if (myRe.exec(sites.styles[i].style.MozBinding)) {
             sites.xss_message[count] = sites.styles[i].style.MozBinding;
             count++;
        }
     }

 },
```

Figure 5: One where of the places in our code where we look for potential XSS vulnerabilities.

Figure 5 shows a simple regular expression matching our keyword of interest, and upon finding it we check if its style property (MozBinding - which a property in the style tag that can be used to embed an *XML* document) contains a given regular expression. Our plugin also contains other regular expressions which match other parts of the HTML document that may contain XSS code.

Next we present code snippet which is responsible for displaying the message containing the name of a possibly malicious XML document, remembering user's response and disabling the appropriate HTML element.

```
if (sites.xss_message) {
   var tmp;
   var tmp2;
   var seen = 0;
   var i = 0;
   for (tmp=0; tmp < sites.xss_message.length; tmp++) {
      seen = 0;
      for (tmp2 = 0; tmp2 < sites.xss_message_seen.length; tmp2++) {
          if (sites.xss_message_seen[tmp2] == sites.xss_message[tmp]) {
              seen = 1;
          }
      }
   if (!seen) {
      if(!confirm("You are about to execute this XML (potential XSS) : " +
sites.xss_message[tmp] )) {
      sites.xss_message_seen[i] = sites.xss_message[tmp];
      var tmp3 =0;
      for (tmp3 = 0; tmp3 < sites.styles.length; tmp3++){
          if (sites.styles[tmp3].style.MozBinding ==
  sites.xss_message[tmp]){
              sites.styles[tmp3].style.MozBinding = null;
              }
          }
          i++;
      }
      }
   }
}
```

Figure 6: Piece of code responsible for displaying the warning message to the user, remembering user's response and finally disabling the appropriate HTML element.

In Figure 6, for each possible XSS vulnerabilities, we ask the user if its execution is desired. We remember user's response, and either execute the page, or halt execution.

Type II

Assume that now our code consists of the following:

```
<script>
document.cookie = "name: myCookie; domain: nikolaylaptev.com";
alert("cookie " + document.cookie);
</script>
```

Figure 7: Potentially malicious cookie access.

In Figure 7, we have a potentially malicious cookie access. When it is executed with our plugin enabled, we get result in Figure 8.

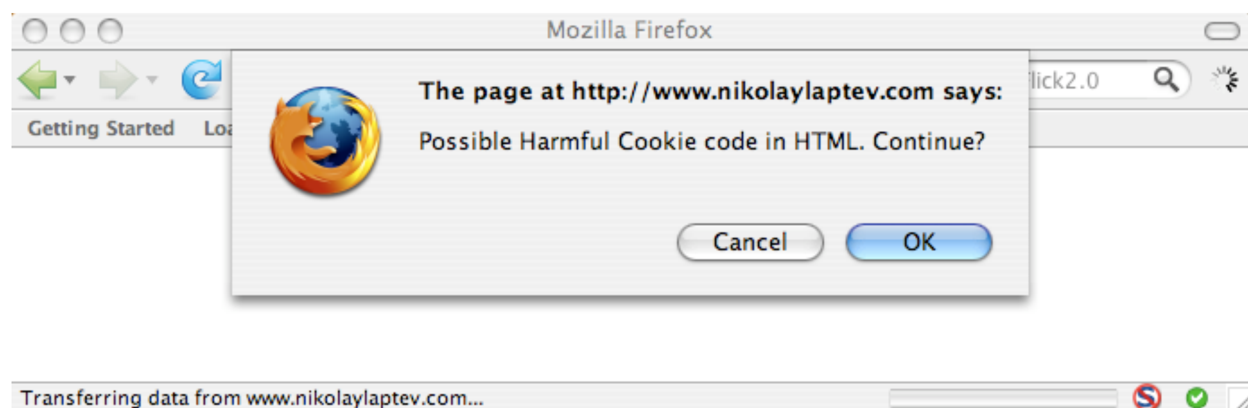

Figure 8: A warning that tells the user of a potentially malicious cookie access.

If the user decides to execute the code, the value of the cookie is displayed, as shows in Figure 9.

Figure 9: The value of the cookie is displayed if execution is granted.

This type of attack is different from the previous case, where we had an XSS attack located in a separate XML file. Here, an attack is embedded in the HTML page itself. In order to defend against this attack, our plugin makes an AJAX call to the page of interest and checks for any potentially malicious cookie accesses then. Figure 10, presents a snippet of that code.

```
http.open('get', window.location, false);
      .......
      .......
http.send(null);
function processResponse() {
//check if the response has been received from the server
    if(http.readyState == 4){
        //read and assign the response from the server

    myRe = new RegExp ("cookie", "g");
    if (myRe.exec(http.responseText) ) {
        if(!confirm("Possible Harmful Cookie code in HTML. Continue?"))
          window.stop();
    }
      ............
      ............
```

FIgure 10: An AJAX call to a site of interest to check for potentially harmful cookie access.

Figure 10 shows how we open a page which is currently is being accessed by the user's browser (window.location). We also must make sure that this AJAX call is blocking, which is guaranteed by the third parameter (false). Once again, if a particular regular expression is detected, a warning is shown, and appropriate action can be taken by the user without actually loading the page in the main browser.

Type III

Assume that user has clicked on the link presented in Figure 11.

www.nikolaylaptev.com/cs236/index.php?user=<script>alert(1);</script>

Figure 11: A potentially malicious script is passed in the URL.

When clicking on the above script with our plugin enabled, result depicted in Figure 12 is produced.

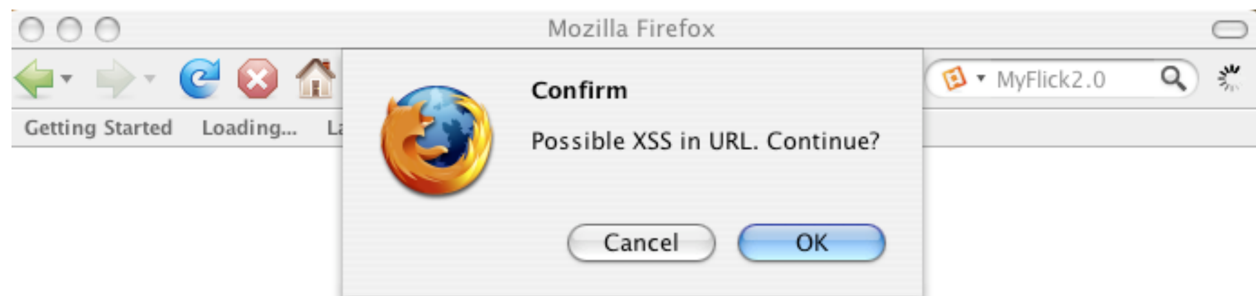

Figure 12: A warning is displayed about a potentially malicious URL.

If the user decides to click OK, the embedded script is executed, as shown in Figure 13.

Figure 13: If user clicks OK, the embedded script is executed.

In order to block against URL attacks, we must check the URL before it is being passed into the browser and data fetched. In order to do that, our plugin checks URL before initializing content in the document, as shown in Figure 14.

```
initContentWindow: function(window) {
  myRe = new RegExp ("script", "g");
  if (myRe.exec(window.location) ) {
      if(!confirm("Possible XSS in URL. Continue?"))
           window.stop();
  }
```

Figure 14: Code that checks the URL before content is initialized.

As Figure 14 presents, URL is checked before initializing content, which enabled us to prevent execution of the web-page altogether.

VI. FUTURE WORK

With the emergence of Web 2.0, current websites require more and more user participation than the previous simple presentations. Collaborative contribution is the very essence of the Web 2.0. Every user can contribute his/her content after a simple registration and others can view it. These websites include online social networks, blogs, and Wikipedia. While these websites provide great user experience, they are also vulnerable to XSS attacks. Malicious users could include a piece of JavaScript code in their online submission to obtain the cookie information or other sensitive information of the user who views their page.

Usually, these websites provide their own HTML tag sets. Users can display various types of content to the viewers without allowing the execution of JavaScript. These html tags are then translated to the appropriate HTML elements, which are parsed and presented to the viewer via Web Brower. In this way, these websites try to eliminate all the malicious JavaScript Code. Indeed, most of the tags are filtered if the tags and the translation rules are carefully designed. However, this method cannot protect the websites from potential XSS attack. There is always a way for the hacker to inject the malicious JavaScript code. Consider the Facebook case. Facebook provides a set of custom defined tags called “The Facebook Markup Language” (FBML). It works well in most cases. Several attributes of the fb:swf tag which allows users embed flash swf’s in their content, however, were un-sanitized. One of them is related to the imgstyle attribute.

```
<fb:swf imgsrc="http://abc.ucla.edu/test.gif"
swfsrc="http://abc.ucla.edu/test.swf" waitforclick="true"
imgstyle="background-color: expression(eval(unescape(‘JavaScript
Here’)));/>
```

With this XSS hole, the attackers can perform any action a Facebook user would normally be able to, but in an automated and silent manner without his/her knowledge. These actions could be profile modification, or add-friend which is similar to the MySpace XSS worm case.

Our proposed approach is based on tag checking. Because the html code is parsed and presented to the end user, the filter operation is performed on either server or client side. With the pattern matching, we cannot examine every possibility. Even if all the possibilities are figured out and we possess a huge list of pattern rules, it is resource consuming to check against each rule in the list.

Another problem with the tag-checking method is that it is hard to tell which piece of JavaScript code is server-provided and which is user-provided. Nowadays JavaScript is widely used in most of the websites. In order to make the page more user-friendly, the Ajax technology is popularized, which makes the webpage full of JavaScript code. The malicious user-provided JavaScript, deeply buried in the server-provided JavaScript code, makes itself difficult to be identified by our tag-checking method. It cannot be solved with more interaction, for example, users are asked when a piece of JavaScript code is to be executed. There are so much JavaScript execution before a page is presented to the end users that users would be exhausted with the endless popup boxes asking for confirmation. Indeed, XSS JavaScript codes are not always malicious. Consider the Google Adsense, the gadget itself is JavaScript code injected into web pages to obtain the keywords for advertisement use.

Many methods are proposed to prevent the XSS attacks. The data tethering method [3] specifies a set of sensitive DOM attributes, which cannot be modified or sent to third-party server. It has two shortcomings. First, it is runtime checked where all the variables which have accessed the sensitive data are marked as sensitive. Therefore, it is resource-consuming to check the exponentially increasing number of variables. Second, the data tethering method cannot prevent the attack which performs actions such as adding a friend in an online social network, because these actions are just normal operations and do not access the sensitive DOM attributes.

The best and cleanest way to solve this is to sandbox the user-provided content. XSS code could be contained in all user-provided content. There are several ways to isolate the user-provided content. The first way is to create an attribute “trusted” for each HTML element to denote whether the contents of this element are trusted. For the content in the un-trusted element, it cannot access other elements beyond the scope of its containing element. The JavaScript code in the un-trusted element cannot access any global functions or variables including DOM attributes, such as cookie information. The XMLHttpRequest method, which is the only way for the JavaScript to send out information via network, cannot be called within un-trusted elements. In this way, we sandbox the user-provided content by surrounding it with un-trusted element such in the form like this: <div trusted = “true”>. This solution can be seen as web browser enforcement. It takes effect only with the support of web browser. During the parsing and rendering of the html code, web browser intercepts the access and execution attempts from the un-trusted element and makes the decision based on the rule mentioned above. This support only needs little modification of the current web browser and can safely include third-party content without jeopardizing the source page. This design is also downward compatible because lower-version web browser just simply discards this attribute.

Another approach is to disable JavaScript within the scope of selected content in a web page. For the current web browser, the JavaScript in the page

are disabled or enabled as a whole. In order to prevent the execution of XSS code, we have to disable the JavaScript in whole page, which results in a fragmentary webpage. The JavaScript is disabled in the user-provide content, while the JavaScript code is enabled for the server-provided element. A finer granularity of the JavaScript control is achieved. This approach also needs the support of the browser and only needs little modification of the current design. This is especially true for the Firefox because it can be controlled by the JavaScript execution engine SpiderMonkey.

If it is hard or impossible to persuade the current browser designer to support the features discussed above, we can also sandbox the user-provided content by putting the user-provided content in a different domain. Consider the MySpace case. We put the user-provided content on the domain *myspacearticle.com* while keeping the server-provided content on the original domain *myspace.com*. Before the whole page is presented to the end user, the user-provided content is fetched from the domain *myspacearticle.com* and put in an iframe. The iframe is integrated into the final web page and shown. In this way, the user-provided content is sandboxed. Suppose there is a piece of malicious code embedded in the user-provided content, which tries to obtain the user cookie information. However, the user cookie is within the scope of domain *myspace.com*, thus cannot be accessed from domain *myspacearticle.com*. This method seems simple and efficient. However, it is not an elegant solution to present a page full of iframes. Furthermore, it is hard to synchronize the content on two domains when updating.

VII. CONCLUSION

In this paper we have presented a pattern matching, client-side approach to preventing XSS attacks. We have demonstrated the robustness of our implementation against several malicious scripts. We have found that by warning the user and prompting for an action to take, we were able to keep global javascript enabled, while still protecting against XSS attacks.

Although tag-checking method can filter out all the XSS codes if carefully designed, we cannot expect to find all the vulnerabilities. A better way is to sandbox the user-provided content. We can enhance the web browsers so that it can support a new attribute *trusted*, which distinguishes the user-provided content from the server-provided content. The content in the element with the un-trusted attribute cannot access the global functions or variables. We could also put the third-party content on a different domain. In this way, the user-provided content is isolated.

References

1. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis, Philip Vogt, Florian Nentwich, Nenand Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna, Network and Distributed Systems Security Symposium, 2007.
2. http://www.cs.virginia.edu/felt/secure_mashups.pdf
3. Abstracting Application-Level Web Security, David Scott and Richard Sharp, WWW 20002.
4. A Web developers guide to cross-site scripting, S Cook
5. The Evolution of Cross-Site Scripting Attacks, D Endler
6. Preventing cross-site scripting vulnerability, D Hu, SANS, 2004