Tracing out Cross Site Scripting Vulnerabilities in Modern Scripts

Haneet Kour  
M.Tech. Student (4th Sem)  
Department of Computer Science & IT, University of Jammu, J & K  
Email: haneetkour9@gmail.com

Lalit Sen Sharma  
Professor  
Department of Computer Science & IT, University of Jammu, J & K  
Email: lalitsen.sharma@gmail.com

ABSTRACT

Web Technologies were primarily designed to cater the need of ubiquitousness. The security concern has been overlooked and such overlooks resulted in vulnerabilities. These vulnerabilities are being highly exploited by hackers in various ways to compromise security. When vulnerability is blocked, the attacker traces out a different mechanism to exploit it. Cross site scripting (XSS) attack is also an exploitation of one of the vulnerabilities existing in the web applications. This paper traces out the vulnerability in functions and attributes of modern scripts to carry out cross site scripting attack and suggests preventive measures.

Keywords - Cookie, Persistent XSS, Reflected XSS, Web vulnerability.

1. INTRODUCTION

Web Technology has become lingua-franca for companies in software development that allows the design of pervasive applications. Thousands of web applications are developed and accessed by millions of users. Security of these websites is becoming an important concern to ensure the user’s authentication and privacy. For this reason, the invention of effective security mechanisms on the web applications has been an increasing concern. Gartner group has noted that almost 75 percent of attacks are tunneled through web applications. According to the Tower Group, nearly 26 percent of customers don’t use online banking services for security fears and 6 percent do not use due to privacy issues. Over 70% of organizations reported of having been compromised by a successful cyber attack [1]. In June/July 2006, the e-payment web application PayPal had been exploited by the attackers to steal sensitive data (e.g., credit card numbers) from its members during more than two years until PayPal’s developers fixed the XSS vulnerability [2][3]. Cross-Site Scripting attack (XSS) is a code injection attack performed to exploit the vulnerabilities existing in the web application by injecting html tag / javascript functions into the web page so that it gets executed on the victim’s browser when one visits the web page and successfully accesses to any sensitive victim’s browser resource associated to the web application (e.g. cookies, session IDs, etc.). By exploiting XSS vulnerabilities in the scripts (mainly javascript since it is highly used scripting language on the client side by web developers), the attacker targets the organizations that accommodate large online communities of users (i.e. social networking sites, blogs and online news sites) or the organizations that rely on web technology to generate revenue (i.e. providers of online services, services that store personal or financial information such as online payment, banking services, etc.). The time gap between identifying an XSS attack and resolving it, is found to be crucial. According to a study by the Ponemon Institute on the Cost of Cyber Crime, the average time taken to resolve a cyber attack was 32 days with an average cost of $1,035,769 (that is $32,469 per day) for the participating sample of organizations [4].

1.1 Types of XSS attack

The main goal of an XSS attack is to execute malicious JavaScript in the victim's browser to steal victim’s authentication details. It is done in following ways:

- **Persistent XSS or Type 2:**
  The Persistent or Stored XSS attack executed when the malicious code submitted by attacker is saved by the server in the web application repository, and then permanently it will be run in the normal page in victim’s browser. A persistent XSS attack against Hotmail was found on October 2001. In this attack, the remote attacker was allowed to steal .NET Passport identifiers of Hotmail’s users by stealing their associated browser’s cookies [5].

- **Reflected XSS or Type 1:**
  Reflected or non-persistent XSS attack is executed in websites when data submitted by the client is immediately processed by the server to generate results that are then sent back to the browser on the client system. The attacker crafts a url link (containing malicious javascript to redirect the victim’s authentication details to attacker domain) and sends it to the victim. By using social engineering techniques, he provokes the victim to follow this malicious link.

- **DOM-based XSS or Type 0:**
  In this case, the vulnerability exists on the client-side code rather than on the server-side code. It is a case of reflected
XSS where no malicious script inserted as part of the page, the only script that is automatically executed during page load is a legitimate part of the page i.e. legitimate JavaScript and careless usage of client-side data result in XSS conditions [6].

2. AIMS AND OBJECTIVES
The objective of this paper is to trace out the cross site scripting vulnerabilities in the web application to steal user’s authentication details (i.e. cookies, session ID etc). This paper also aims to study how this XSS attack can be mitigated.

3. RELATED WORK
The main goals of XSS attacks are stealing the victim user’s sensitive information and invoking malicious acts on the user’s behalf. A survey has been done on detection and prevention techniques proposed by various researchers to mitigate XSS risks. XSS vulnerabilities can be detected by performing static and dynamic analysis on web application. Many researchers are carrying out their study in this domain [7][8]. Some of them are listed as:

M.T. Louw et. al. [9] introduced a server side prevention technique against XSS attacks. This technique known as BEEP (browser enforced embedded policies) modifies the browser so that it can’t execute the malicious script. Security policies dictate what the server sends to BEEP enabled browser.

O.Hallaraker and G.Vigna [10] proposed a mechanism for detecting malicious javascript. The system consists of browser embedded script auditing component and IDS to process the audit logs and compare them to signature of already known malicious behaviour or attacks.

Shasank Gupta et. al. [11] introduced a novel technique called Dynamic Hash Generation Technique that makes cookies worthless for the attackers. This technique is implemented on the server side and its main task is to generate a hash value of name attribute in the cookie and send this hash value to the web browser. With this technique, the hash value of name attribute in the cookie which is stored on the browser’s database is no more valid for the attackers to exploit the vulnerabilities of XSS attacks.

Shasank Gupta and Lalitsen Sharma [12] introduced a technique to mitigate XSS vulnerability by introducing a Sandbox environment on the web browser. Client’s web browser under the protection of a sandbox submits the user-id and password to a web server. Web server will generate the cookie and send this cookie to client’s web browser which is sandbox protected. Now this cookie value will neither leak into the windows nor it can be grabbed by any attacker. On the other hand, sandbox allows the execution of malicious script on the client’s web browser but it cannot give the authority to simply leak the cookie out of this protected environment and hence bypass the XSS attack.

S.Shalini and S.Usha [13] provided a client-side solution to mitigate XSS attack that employs a three step approach to protect cross site scripting. This technique found to be platform independent and it blocks suspected attacks by preventing the injected script from being passed to the JavaScript engine rather than performing risky transformations on the HTML.

Engin Kirda et. al. [14] presented Noxes, a client-side solution to mitigate cross-site scripting attacks. Noxes acts as a web proxy and uses both manual and automatically generated rules to mitigate possible cross-site scripting attempts.

Dr R.P. Mahapatra et. al. [15] presented a technique to protect java web applications from Cross Site Scripting attack (XSS) by applying a framework based on pattern matching approach. The proposed approach consists of Request/Response Analyser and Modifier modules. The Request Analyser/Modifier Module decides whether request is malicious or not and takes decision accordingly. Response analyser and Modifier module deals with the data to be returned the client, it modifies the malicious response to harmless data. Attack Recorder and Response Rejecter Module records the malicious Request/Response for future use. The authors had employed Java Regex for pattern generation and matching the malicious attack signatures.

Kieyzun et. al. [16] devised an automatic technique for creating inputs that expose SQLI and XSS vulnerabilities. The technique generates sample inputs, symbolically tracks tainted data through execution (including through database accesses), and mutates the inputs to produce concrete exploits. This technique creates real attack vectors, has few false positives, incurs no runtime overhead for the deployed application, works without requiring modification of application code, and handles dynamic programming-language constructs. The author also implemented the technique in php, a tool Ardilla. This approach was implemented in a tool called BLUEPRINT that was integrated with several popular web applications.

Stefano Di Paola and Giorgio.F [17] described a universal XSS attack against the Acrobat PDF plugin. When the client clicks the link and the data is processed by the page (typically by a client side HTML-embedded script such as JavaScript), the malicious JavaScript payload gets embedded into the page at runtime.

Shashank Gupta and B.B. Gupta [18] proposed a security model called Browser Dependent XSS Sanitizer (BDS) on the client-side Web browser for mitigating the effect of XSS vulnerability. The authors used a three-step approach to eliminate the XSS attack without degrading much of the user’s Web browsing experience on various modern browsers.

4. EXPERIMENTAL SET UP
In this study, a website in php has been developed and hosted on the local host (XAMPP server). The experiments to exploit XSS vulnerabilities in the website have been performed to steal user’s cookies. The study is focused on persistent and reflected attacks on the websites that maintain user’s authentication state by using cookies. These experiments have been performed on modern browsers (Google Chrome 49, IE 11, Opera 15 and Firefox 44.0.2). The Fig. 1 shows the architecture for exploiting XSS vulnerabilities in the local host.
The vulnerabilities in the web application through tags and attributes in HTML and the functions in javascript are traced out to perform XSS attack by injecting malicious javascript to steal victim’s cookies. The overall analysis of these experiments has been summarized in Table 1. The following javascript code (that provides a hyperlink to redirect the victim’s cookie) is inserted to steal user’s cookie (by getCookie.php file in the attacker domain):

```javascript
<iframe src='javascript:malicious javascript'>
</iframe>
```

This html file contains malicious javascript

<iframe src=http://localhost/attacker/xss.html>
</iframe>

In IE and Opera, cookies are not stolen. But in chrome and firefox, attack is performed.

<table>
<thead>
<tr>
<th>XSS attack vectors</th>
<th>Attack performed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;script&gt;---malicious javascript code---&lt;/script&gt;</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>&lt;script src=http://localhost/attacker/xss.js&gt;</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>&lt;img src=javascript:malicious code&gt;</code></td>
<td>No</td>
</tr>
<tr>
<td><code>&lt;img src=path event-attribute=malicious javascript&gt;</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>&lt;img src=path alt=javascript:malicious code&gt;</code></td>
<td>No</td>
</tr>
<tr>
<td><code>&lt;div style=&quot;background : url (malicious javascript)&quot;&gt;</code></td>
<td>No</td>
</tr>
<tr>
<td><code>&lt;iframe src=javascript:malicious code&gt;</code></td>
<td>In IE and Opera, cookies are not stolen. But in chrome and firefox, attack is performed.</td>
</tr>
<tr>
<td><code>&lt;iframe src=http://localhost/attacker/xss.html&gt;</code></td>
<td>This html file contains malicious javascript</td>
</tr>
<tr>
<td><code>&lt;link rel=stylesheet href=javascript:malicious code&gt;</code></td>
<td>No</td>
</tr>
<tr>
<td><code>&lt;object data=javascript:malicious code&gt;</code></td>
<td>No</td>
</tr>
<tr>
<td><code>&lt;object type=x-scriptlet data=javascript:malicious code&gt;</code></td>
<td>No attack in IE</td>
</tr>
<tr>
<td><code>&lt;a href=# onclick=javascript:malicious javascript&gt;</code></td>
<td>In other browsers, script is executed but victim’s cookies are not stolen</td>
</tr>
<tr>
<td><code>&lt;div style=&quot;width:expression(malicious javascript;&quot;&gt;</code></td>
<td>No</td>
</tr>
<tr>
<td><code>&lt;input type=image src=javascript:malicious code&gt;</code></td>
<td>Cookies are retrieved by attacker</td>
</tr>
<tr>
<td><code>&lt;script&gt;------ XMLHttpRequest object code------&lt;/script&gt;</code></td>
<td>No</td>
</tr>
</tbody>
</table>

**Encoding**: Encoding of the user input is done by the function htmlspecialchars(‘user-input’) in php to mitigate XSS attack. It escapes user input so that the browser interprets it only as data, not as code. This function converts characters like < and > into &lt; and &gt; respectively. Although, the attacker posts the malicious code, but htmlspecialchars( ) encodes all the code before inserting it into the database of web application. Thus, the script does not get executed.
Sanitization: Sanitization function (that removes all the html tags from the user input) `filter_var("user-input",FILTER_SANITIZE_STRING)` in php is used to prevent the insertion of malicious code into the database of web application, thus mitigating XSS attack.

Regular Expressions Matching: The regular expressions for the possible malicious javascript code (to carry out XSS) have been defined. When the user enters the input, then it is matched with all predefined regular expressions to check whether it is valid or not. The function `ereg("predefined regular expression", "user-input")` is used to perform validation of user input. This method employs black listing techniques.

6. RESULT AND DISCUSSION

By performing these experiments on the local host, various ways have been traced out to execute javascript in victim’s browser. The value of event attributes, in html tags, has been set to malicious javascript code to carry out XSS attack and the attack became successful. Also, the ‘src’ attribute of some html tags (`<img src=javascript:code>`, `<iframe src=javascript:code>`, `<input type=image src=javascript:code>`, `<object data=javascript:code>`) set to malicious javascript. The script does not get executed in case of `<img>`, `<input type=image>` and `<object>` tag in modern browsers. But these browsers support the execution of javascript through ‘src’ attribute in `<iframe>` tag. Although IE11 and opera15 allow the execution of javascript yet the XSS attack is denied but the attack becomes successful in case of chrome and firefox. It occurs due to DOM issues.
code (that can be misused by hackers to steal user’s authentication details). It causes overburden on the developer’s side.

- If the attacker inserts the malicious code that is not in the list of predefined regular expressions templates, then this code can be bypassed and it gets executed on the victim’s browser.

7. CONCLUSION

By now there have been a variety of defensive techniques to prevent XSS. These techniques are implemented on the client-side or server-side to protect web users from XSS injection attack. Still XSS is emerging as one of the top 10 web application vulnerabilities leading to security breach. A weak input validation on the web application causes the stealing of cookies from the victim’s web browser. The hackers are becoming powerful day by day to develop new approaches to carry out XSS attack. Cross-site Scripting (XSS), the top most vulnerability in the web applications, demands an efficient approach on the server side as well as client side to protect the users of the web application.

REFERENCES


