Using Burp to Find Cross-Site Scripting Issues

Cross-Site Scripting (XSS) is the most prevalent web application vulnerability found in the wild. XSS often represents a critical security weakness within a application. It can often be combined with other vulnerabilities to devastating effect. In some situations, an XSS attack can be turned into a virus or self-propagating worm.

XSS vulnerabilities occur when an application includes attacker-controllable data in a response that is sent to the browser without properly validating or e the content. Cross-site scripting attacks may occur anywhere that an application includes in responses data that originated from any untrusted source. A attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be truste will execute the script because it thinks the script came from a trusted source. The malicious script can access any cookies, session tokens, or other sen information used with that site.

XSS vulnerabilities come in various different forms and may be divided in to three varieties: reflected (non-persistent), stored (persistent) and DOM-base

Using Burp Scanner to Automatically Test for XSS

The articles below describe how to use Burp Scanner to automatically detect different types of XSS vulnerabilities:

Using Burp Scanner to Find Cross-Site Scripting (XSS) Issues

Using Burp Scanner to Test for DOM-Based XSS

Understanding XSS: The Same-Origin Policy

Security on the web is based on a variety of mechanisms, including an underlying concept of trust known as the same-origin policy. This mechanism is implemented within browsers and is designed to prevent content that came from different origins from interfering with one another. The same-origin polic essentially states that content from one site (such as `https://bank.example1.com`) can access and interact with other content from that site, while c from another site (`https://malicious.example2.com`) cannot do so, unless it is explicitly granted permission.

If the same-origin policy did not exist, and an unwitting user browsed to a malicious website, script code running on that site could access the data and functionality of any other website also visited by the user. Due to the same-origin policy, if a script residing on `https://malicious.example2.com` q `document.cookie`, it will not obtain the cookies issued at `https://bank.example1.com`, and a potential hijacking attack would fail. However, when attacker exploits an XSS vulnerability, they are able to circumvent the same-origin policy. As far as the user's browser is concerned, the attacker's malicic JavaScript was sent to it by `https://bank.example1.com`. As with any JavaScript received from a website of the "same origin", the browser execute script within the security context of the user's relationship with `https://bank.example1.com`. Although the script has originated elsewhere, it can gai access to the cookies issued by `https://bank.example1.com`. This is also why the vulnerability itself has become known as cross-site scripting.

Manually Detecting XSS

When manually testing for XSS issues, first you must identify instances of reflected input, then manually investigate each instance to verify whether it is a exploitable. In each location where data is reflected in the response, you need to identify the syntactic context of that data. You must find a way to modify input such that, when it is copied into the same location in the application's response, it results in execution of arbitrary script. In the articles below, we pr some general examples of testing for reflected and stored XSS, followed by some more in-depth approaches for detecting XSS in different HTML contex

Using Burp to Manually Test for Reflected XSS

Using Burp to Manually Test for Stored XSS

Exploiting XSS - Injecting into Direct HTML

Exploiting XSS - Injecting into Tag Attributes

Exploiting XSS - Injecting into Scriptable Contexts