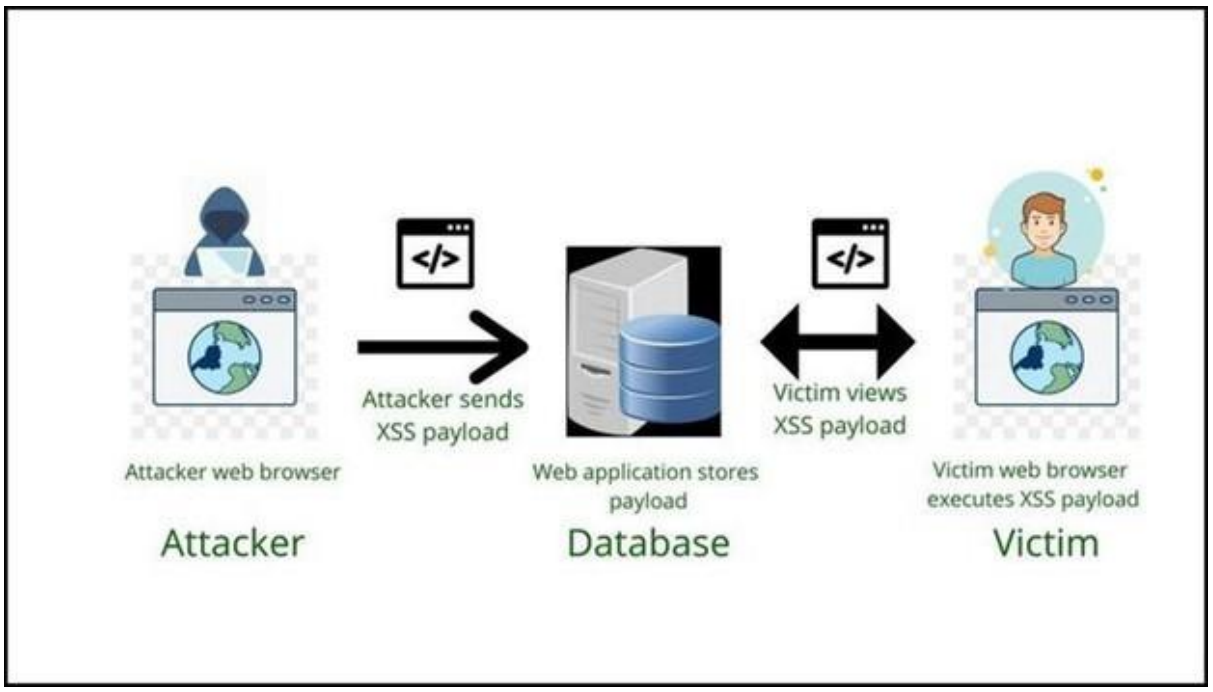
Introduction:

Cross-site scripting, or XSS, is an injection attack that has long been considered a top problem in the cybersecurity world. When an attacker injects malicious java script code into a web application, it is run under the victim's user accounts.

How cross-site scripting attacks work

Cross-site scripting is a client-side attack in which malicious source code, typically in JavaScript, is injected into an application. The malicious code can be inserted in a variety of methods. JavaScript provides the ability to access browser data such as cookies, session tokens, and storage data. This data, like session tokens, can be used to impersonate another individual if obtained. As a result, attackers attempt to gain access to sensitive data by injecting JavaScript and directing it to themselves. Overall, the attacker has access to all of JavaScript's capabilities in this type of assault. Here are some acts that an attacker can perform.

- Stealing sensitive data (personal data, payment data, etc.)
- Redirection to phishing sites
- Performing unauthorized action by stealing session tokens



Here are the different types of XSS attacks.

Reflected XSS

In a Reflected XSS attack, a web application extracts data (such as query strings) from a URL and uses it in some way. The risk comes when an attacker creates a URL with parameters containing an HTML script tag. If the program uses these parameters, such as changing the DOM, the script may run, allowing the attacker to steal data or influence the web application. This attack is significant because the target does not need to be a frequent user of the application, they can simply click on a malicious link to be taken to the compromised page. Furthermore, these attacks can propagate swiftly via open chats or social media, emphasizing the significance of checking links before accessing them.

Stored XSS

A Stored XSS attack involves the attacker attempting to inject malicious scripts into the application's system via input fields. These assaults frequently target forums or chats, where scripts may be easily added and seen by other users. The server administrator is also at risk in WordPress because the editor functions within a regular web interface as part of the content management system, allowing the injected script to be executed. If successful, the attacker exploits the vulnerability created by the failure to correctly escape the data.

DOM-based XSS

The DOM-based cross-site scripting attack, like the Reflected attack, occurs when the application's DOM is changed. Again, the attacker attempts to inject and execute his code in the web application, for example, by modifying the DOM. As a result, the code originates from a source that he may control and alter as needed. As a result, the general use of JavaScript functions like eval() is improper. eval() allows for parsing a string as JavaScript code, which can be hazardous.

How to prevent XSS attacks?

XSS attacks can be prevented by various methods, but it is always vital to assess the extent to which it is reasonable to restrict something in exchange for lowering the danger of an XSS attack. This section discusses some strategies to prevent cross-site scripting attacks.

Disable JavaScript

As previously stated, XSS attacks may only occur when malicious JavaScript code is run on the client side. As a result, turning off JavaScript in your browser is a secure way to defend yourself from this attack. However, because today's web applications rely significantly on JavaScript, many of them are likely to fail or work only partially.

Dynamic execution with JavaScript

Dynamically run JavaScript code can provide a security concern, as developers and researchers have repeatedly stated, giving the eval () function as an example. This risk is most visible in DOM manipulation, where JavaScript is frequently performed dynamically, such as via the DOM element attribute inner HTML. With inner HTML, an attacker might inject a code snippet including a script tag into the program. If the web application modifies a DOM element using inner HTML and mistakenly communicates harmful code, the script tags will execute it. Because inner HTML is frequently used for text modifications, it is preferable to utilize safer alternatives such as Node which can prevent such attacks while still updating the text.

HTTP-Header X-XSS-Protection

The HTTP header X-XSS-Protection response header is a method of protecting against XSS attacks that is supported by some browsers (Safari, Internet Explorer). When the browser detects an XSS attack, it executes multiple defence measures depending on the parameters in the response header. For example, the defence system may prohibit the web application from loading or remove the insecure areas containing the XSS attack script.

X-XSS-Protection:0 X-XSS-Protection:1 X-XSS-Protection: 1; mode=block

The 0 or 1 indicates whether the protective filtering should be turned off or on (0 = off, 1 = on), and the information following the semicolon specifies how to respond in the event of an attempted injection. So, in the example with the block, not only is the injected script deleted (sanitized), but the page itself is blocked from loading.

Content Security Policy

A Content Security Policy (CSP) protects web applications from injection threats such as XSS. It requires the browser to only load scripts from trusted sites. If a script from an unapproved source is identified, the browser will prevent it. To enable CSP, the server must return a Content-Security-Policy HTTP header, which is often configured by the server administrator to allow only trusted domains, such as the server's own, to load scripts.

“Content-Security-Policy: default-src 'self' “

Client-side XSS prevention

Many online applications are written in JavaScript, hence this is used as an example, although the ideas underlying the security mechanism apply to many programming languages and frameworks. A simple demonstration of an inserted code might look like this

var elment = document. getElementById('elment'); var param = "Parameter"; elment.innerHTML = '<img src="unknown" onerror="alert(param)">';

In this case, the attacker uses the fact that the innerHTML method parses the past statement. Consider the possibility that the attacker could alter or pass that parameter.

Server-side XSS prevention

As previously stated, server-side security can be achieved through HTTP headers, but programmatic protection is also an option. Using the example of WordPress and the computer language PHP, which is used for WordPress development, it makes sense to sanitize data, particularly data from input fields. Because PHP is rendered on the server, the data that will be communicated can be sanitized before it is delivered. WordPress provides several cleansing functions on its own. For example, an input containing a malicious script can be defused using the following function call

```
$title = sanitize_text_field( $_POST['title'] );

update_post_meta( $post->ID, 'title', $title );
```

However, this can also be done in the other direction, i.e. escaping or cleaning when displaying content in the client

```
<?php echo esc_html( $title ); ?>
```

When working with WordPress plugins, it is very crucial to investigate the plugins' sources (if feasible, most of which are open source) to see if they take precautions to prevent code injection. Especially with plugins with many input and management choices, an input field or option can easily be ignored. neglected sanitizing is not uncommon; even some of the most popular plugins neglected it and were exploited, as was the case with WooCommerce.

Conclusion

An attacker can swiftly and simply carry out XSS attacks, causing significant damage. Taking safeguards as a developer is extremely simple and effective, as demonstrated. It is critical to put yourself in the shoes of the attacker and consider how you may proceed and close these vulnerabilities properly. Furthermore, it is critical to raise awareness among web developers about this assault, as it remains one of the most popular, and many applications do not take enough measures.

References:

https://portswigger.net/web-security/cross-site-scripting

https://rb.gy/oax662

https://rb.gy/zsmhru

By

P Suresh

Security Analyst

Vardaan Cybersecurity Private Limited