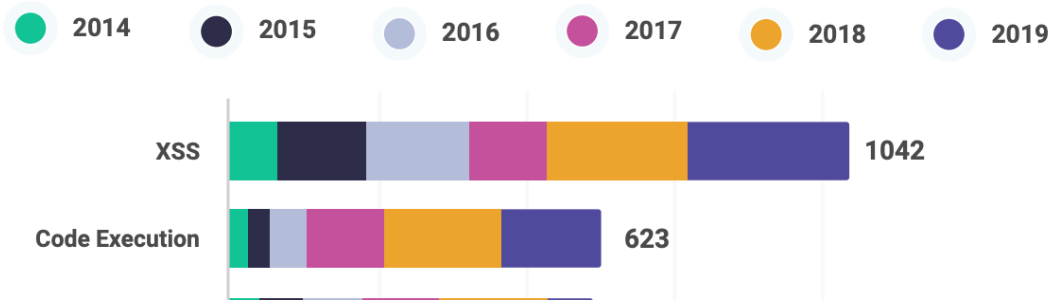
snyk.io

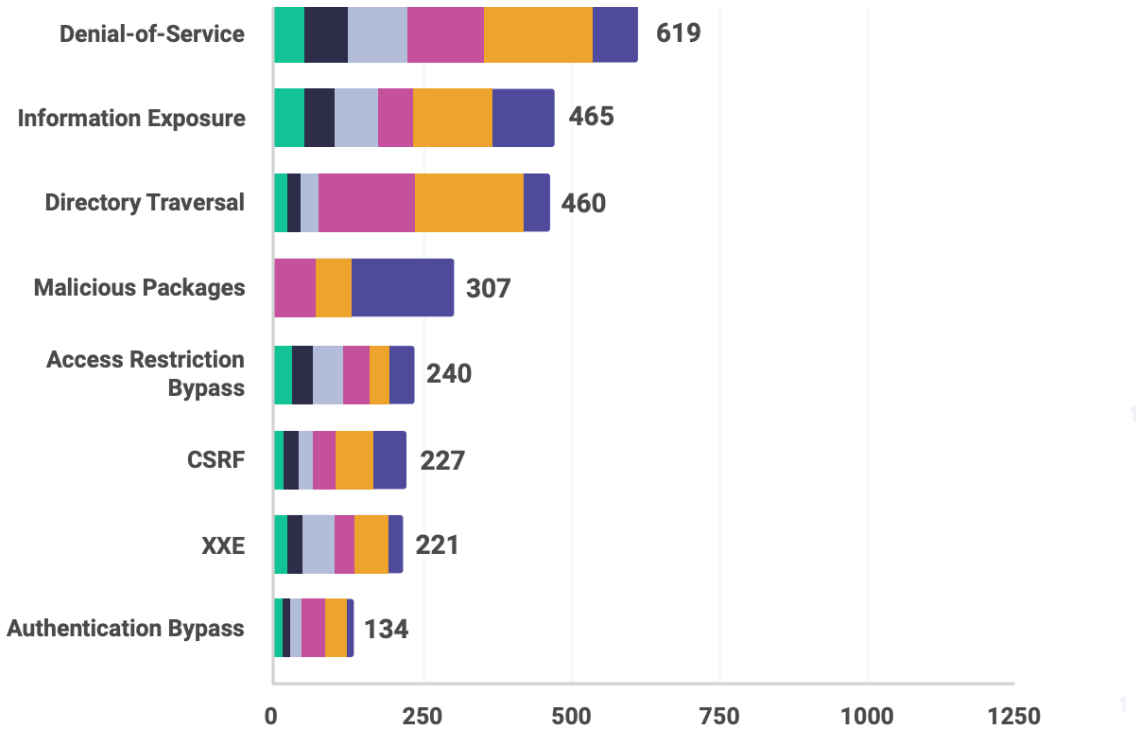
What is cross-site scripting | How to prevent an XSS attack | Snyk

11-14 minutes

Cross-site scripting—referred to as XSS—is an application vulnerability that has the potential to wreak havoc on applications and websites. XSS is so rampant and potentially harmful that it continues to be included in the Open Web Application Security Project (OWASP) list of top 10 vulnerabilities.

In fact Cross-Site Scripting is the most common vulnerability discovered since 2014 and in 2019 it continues to appear in the Top 3 of reported vulnerabilities.





Top disclosed vulnerabilities – State of Open source Report 2020 by Snyk.

What is Cross-Site Scripting (XSS)?

Cross-site scripting is a website attack method that utilizes a type of injection to implant malicious scripts into websites that would otherwise be productive and trusted. Generally, the process consists of sending a malicious browser-side script to another user. This is a common security flaw in web applications and can occur at any point in an application where input is received from the browser and used to create output without first validating or encoding the data.

In some cases, browser-side script can also introduce this vulnerability allowing an attacker to exploit it without the target user making a request to the web application.

Attackers exploit XSS by crafting malicious code that can be routed to another user, is then executed by the unsuspecting browser. Since the script is most often included in the content of the web application's response, it is executed and has the same access as if the script is legitimate. Access may be allowed to session tokens, cookies, and even confidential or sensitive information the browser has access to on that site, even rewriting the HTML page content.

Safeguard against XSS attacks

Automatically find, prioritize and fix vulnerabilities in the open source dependencies used to build your cloud native applications

How does Cross-site Scripting work?

Exploiting cross-site scripting vulnerabilities is a relatively simple task for attackers. By injecting a

malicious script into unprotected or un-validated browser-supplied input, the attacker causes the script to be returned by the application and executed in the the browser. This could allow it to take control of the application's functionality, manipulate data, or plant additional malicious code.

What are the types of XSS attacks?

There are three main types of XSS attacks: reflected XSS, stored XSS, DOM-based XSS.

Reflected XSS Attacks

In reflected XSS attacks, the malicious script is injected into an HTTP request (usually by specifically crafted link supplied to the user). As the simplest variety, it uses input parameters in the HTTP request that can be easily manipulated to include the damaging script content. The malicious script is then reflected from the server in a HTTP response and gets executed in the victim's browser.

In this case, it acts like a stored XSS without

actually storing malicious data on the server.

Here is an example of Reflected XSS Attack:

Let’s say example.com/profile contains a name parameter. The URL for the request would look like this: https://example.com /profile?user=Tammy. Based on this input, the web application would thus respond with “Hi Tammy” at the top of the page. If the parameters are not validated to ensure it only contains expected data, an attacker could have a user visit a malicious version of the URL like this: https://example.com /profile?user<script>some_malicious_code</script>.

When the response is sent to the browser, it includes that malicious script, which is then executed in the browser, likely without the user even knowing. This is an example of a reflected XSS attack, as the malicious code is immediately “reflected” back to the user making the request.

Stored XSS Attacks

In what is known as a stored or persistent XSS attack, malicious content is delivered directly,

along with the server's response when the user loads a web page. Thus the content is already stored in the website's database (hence the name for such attacks). Users then simply enter the hacked web page and fall victim to such attacks.

Every single user who opens such a compromised website is thus at risk of having their personal data stolen, and so this could be considered the most dangerous type of XSS attack.

But how does the malicious content get into the database to begin with? In most cases, it is introduced through unprotected web page forms in which user input is not properly validated and sanitized. If the data entered by a hacker is not validated on both the client and server sides, it will be saved in the database. For example, such input might include a comment text area, post text editor, personal data editor, or others forms.

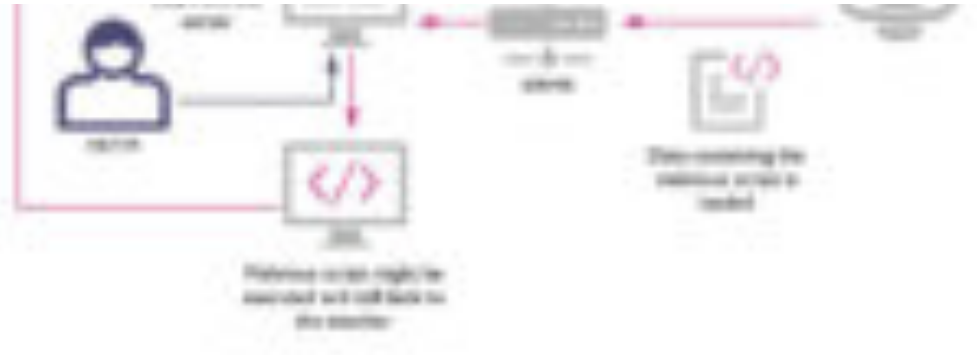


Figure 1: Stored or persistent XSS attack flow

Once an attacker manages to send malicious content to the server and that content appears unfiltered on a web page, all users become potential victims.The common remedy for stored XSS attacks is to sanitize input on both the front end and back end of the application. Sanitizing involves a combination of validating data and escaping special characters or completely filtering those out and is a common best practice for web and JavaScript security.

DOM-Based XSS Attacks

Document Object Model (DOM) is an interface that enables applications to read and manipulate the structure of a web page, its content, and style. In a DOM-based XSS attack, the vulnerability lies in the browser-side script code and can be exploited without any server

interaction at all, modifying the environment in the unsuspecting victim's browser.

Sometimes, DOM-based XSS attacks are similar to reflected attacks. The above example of a reflected XSS attack can be applied in this case with a single assumption: The web application reads data directly from a query string.

Let's say the application uses the query parameter "name" in order to instantly display the user's name on the screen while waiting for the rest of a page to load. Without proper validation, this can yield the same result as with a reflected attack, if the hacker is successful in making the victim open a suspicious link.

As with stored XSS, to prevent reflected and DOM-based attacks, developers should implement data validation and avoid displaying raw user input, despite the presence or absence of communication with the server.

What are the Cross-site Scripting attack vectors?

There are several vectors commonly utilized in

XSS attacks:

- **<script> tag:** A script tag can be used to reference external JavaScript code, making this the most straightforward XSS point. Attackers can also embed the malicious code within the script tag.
- **JavaScript events**: Another popular XSS vector used by attackers, event attributes, can be applied in a variety of tags. Such attributes as “onerror” and “onload” are examples.
- **<body> tag:** Event attributes can also be the source of the script when provided through the “body” tag.
- **<img> tag:** Depending on the browser in use, this attribute may be useful by attackers to execute the JavaScript code.
- **<iframe> tag:** Especially effective for phishing attacks, this vector allows the XSS attack to embed another HTML page into the current page.
- **<input> tag:** Some browsers allow manipulation through this vector, which can be used to embed a script.

- **<link> tag:** This tag can contain a script, instead of the normal use of linking to external style sheets.
- **<table> tag:** Where the background attribute normally refers to an image, this could be compromised to refer to the offending script.
- ** tag:** This tag also contains a background reference and can be used in the same way as <table> to refer to a script.
- **<object> tag:** Scripts from an external site can be included using this tag.

  While there are other vectors XSS attackers use in their efforts to steal information and compromise websites, these are some of the most commonly used methods. Developers must adhere to proper escaping and sanitizing practices to guard against such attacks.

What is the impact of XSS vulnerabilities?

XSS has the potential to wreak havoc on

applications and websites. The fact that XSS has been present in every OWASP top 10 list illustrates the need to protect web applications from this vulnerability.

Depending on the attacker and their malicious intent, XSS attacks can result in different impacts including:

- hijacking a user's session, utilizing credentials to access other sites or redirect the user to unintended websites,
- altering website pages or inserting sections into a web page,
- executing scripts to extract sensitive information from cookies or databases,
- if the victim has administrative rights, the attack could extend to the server side, causing further damage or retrieving additional sensitive information.

How to test Cross-site Scripting?

Testing for XSS vulnerability begins at the design phase, taking best practices into account from

the beginning. Website designers should build in security measures, not add them after-the-fact. Initial tests include bench scans of code to identify the use of common XSS attack vectors that present potential vulnerabilities. This allows mitigation of the weaknesses before actual application testing begins.

Key steps in testing for XSS vulnerabilities for critical web applications include:

- utilizing a code scanning tool to detect vulnerabilities that allow code corrections during the development process,
- implementing automated testing functionality that will reveal potential vulnerabilities thoroughly and quickly.

Test your code for XSS vulnerabilities

Automatically find, prioritize and fix vulnerabilities in the open source dependencies used to build your cloud native applications

What is a Cross-site Scripting example?

Cross-site scripting can be exploited when a web application uses data supplied by the browser to create responses to user requests. A very simplistic example would be a case where a web application makes use of a parameter in the URL to provide a customized response to the user.

Let’s say exmple.com/profile contains a name parameter. The URL for the request would look like https://example.com/profile?user=Tammy. The web application responds with “Hi Tammy” at the top of the page based on this input.

If the user parameter is not validated to ensure it only contains expected data, an attacker could have a user visit a malicious version of the URL that looks like this:

https://example.com /profile?user<script>some_malicious_code</script>.

When the response is sent to the browser, it includes that malicious script which is then executed in the browser, likely without the user even knowing. This is an example of a reflected XSS attack. The malicious code is immediately “reflected” back to the user making the request.

How to prevent Cross-site Scripting?

There are several key action items for preventing XSS attacks: enhance education and awareness of your developers, screen and validate the data input and scan code for vulnerabilities.

- **Education and awareness:** Make sure all developers, website designers, and QA teams are aware of the methods hackers use to exploit vulnerabilities and provide guidelines and best practices for coding. This includes proper escaping/encoding techniques for the application environment (JavaScript, HTML, etc.).
- **Sanitize input:** Whether for internal web pages or public websites, never trust the validity of user input data. Screen and validate any data fields, especially if it will be included as HTML output.
- **Scan code:** Implement software that scans code for vulnerabilities, including cross-site scripting, to ensure that best practices are being followed and minimizes exposure from XSS and other OWASP-listed vulnerabilities.
- **Content Security Policy:** Use Content Security

Policy (CSP) to define what a website can do, and by this reduce the risk of an XSS attack. By using CSP, XSS can be blocked entirely (by blocking all in-line scripts) or be reduced to much lower risk.

OWASP provides additional guidance on how to prevent XSS vulnerability with a comprehensive cheat sheet.