Research

* Cross-site scripting (XSS) is an old but always relevant and dangerous type of attack that affects almost all web applications, old and new. It relies on developers using javascript to improve the end-user experience of their application, but when the javascript isn't handled properly, it can lead to a variety of problems, one of which is XSS.
* The risk of XSS is that the malicious code is usually injected directly into the vulnerable application, rather than a redirect site that the user should be aware of. So, if you frequently visit example.com and someone sends you a link to one of their articles that looks like example.com/this-article-is-good? id=%3Cscript%3Ealert%281%29%3C%2Fscript%3E You'll probably click it because it's something you're used to. What you're not aware of is that some code was injected into the site without your or the site's approval, and that code could steal your session, take screenshots, activate a keylogger, and so on.
* A more dangerous type of XSS vulnerability is the persistent one, where you don't even have to click on a link to execute the code; instead, you simply browse to some page on a trusted site and an attacker's comment containing malicious code saved in the database is displayed on the page, and suddenly you and everyone else who visits that page is triggering something they don't want to trigger.
* One of the most famous examples of XSS is the “Samy“. Samy is one of the fastest spreading malwares in internet history. It abused unsanitized profile posts to inject harmful JavaScript code that was saved to the database and then activated whenever a user viewed that post, thus spreading the worm to themselves and so on, yahoo account hijack via email phishing and XSS, attackers made a page with malicious JavaScript that would steal cookies of visitors. The attack was executed by sending an email with a link to the popular news page article, but the link linked back to the attacker’s site which contained malicious code.
* 3 most common types; reflected, persistent, DOM-based XSS

**Reflected XSS**

* Reflected cross-site scripting (or XSS) arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.
* If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. Amongst other things, the attacker can: Perform any action within the application that the user can perform, view any info that the user can view, modify any information that the user can modify and initiate interactions with other application users, including malicious attacks, that will appear to originate from the initial victim user.
* There are various means by which an attacker might induce a victim user to make a request that they control, to deliver a reflected XSS attack. These include placing links on a website controlled by the attacker, or on another website that allows content to be generated, or by sending a link in an email, tweet, or other message. The attack could be targeted directly against a known user or could be an indiscriminate attack against any users of the application.
* The need for an external delivery mechanism for the attack means that the impact of reflected XSS is generally less severe than [stored XSS](https://portswigger.net/web-security/cross-site-scripting/stored), where a self-contained attack can be delivered within the vulnerable application itself.

How to test for reflective XSS

* Test every entry point. Test separately every entry point for data within the application's HTTP requests. This includes parameters or other data within the URL query string and message body, and the URL file path. It also includes HTTP headers, although XSS-like behaviour that can only be triggered via certain HTTP headers may not be exploitable in practice.
* Submit random alphanumeric values. For each entry point, submit a unique random value and determine whether the value is reflected in the response. The value should be designed to survive most input validation, so needs to be short and contain only alphanumeric characters. But it needs to be long enough to make accidental matches within the response highly unlikely. A random alphanumeric value of around 8 characters is normally ideal. You can use Burp Intruder's number payloads [https://portswigger.net/burp/documentation/desktop/tools/intruder/payloads/types#numbers] with randomly generated hex values to generate suitable random values. And you can use Burp Intruder's [grep payloads settings](https://portswigger.net/burp/documentation/desktop/tools/intruder/configure-attack/settings#grep-payloads) to automatically flag responses that contain the submitted value.
* Determine the reflection context. For each location within the response where the random value is reflected, determine its context. This might be in text between HTML tags, within a tag attribute which might be quoted, within a JavaScript string, etc.
* Test a candidate payload. Based on the context of the reflection, test an initial candidate XSS payload that will trigger JavaScript execution if it is reflected unmodified within the response. The easiest way to test payloads is to send the request to [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater), modify the request to insert the candidate payload, issue the request, and then review the response to see if the payload worked. An efficient way to work is to leave the original random value in the request and place the candidate XSS payload before or after it. Then set the random value as the search term in Burp Repeater's response view. Burp will highlight each location where the search term appears, letting you quickly locate the reflection.
* Test alternative payloads. If the candidate XSS payload was modified by the application, or blocked altogether, then you will need to test alternative payloads and techniques that might deliver a working XSS attack based on the context of the reflection and the type of input validation that is being performed. For more details, see [cross-site scripting contexts](https://portswigger.net/web-security/cross-site-scripting/contexts)
* Test the attack in a browser. Finally, if you succeed in finding a payload that appears to work within Burp Repeater, transfer the attack to a real browser (by pasting the URL into the address bar, or by modifying the request in [Burp Proxy's intercept view](https://portswigger.net/burp/documentation/desktop/tools/proxy/intercept-messages), and see if the injected JavaScript is indeed executed. Often, it is best to execute some simple JavaScript like alert (document. Domain) which will trigger a visible popup within the browser if the attack succeeds.

**Persistent**

* Stored cross-site scripting (also known as second-order or persistent XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

Impact of persistent XSS

* If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. The attacker can carry out any of the actions that are applicable to the impact of [reflected XSS vulnerabilities](https://portswigger.net/web-security/cross-site-scripting/reflected).
* In terms of exploitability, the key difference between reflected and stored XSS is that a stored XSS vulnerability enables attacks that are self-contained within the application itself. The attacker does not need to find an external way of inducing other users to make a particular request containing their exploit. Rather, the attacker places their exploit into the application itself and simply waits for users to encounter it.
* The self-contained nature of stored cross-site scripting exploits is particularly relevant in situations where an XSS vulnerability only affects users who are currently logged in to the application. If the XSS is reflected, then the attack must be fortuitously timed: a user who is induced to make the attacker's request at a time when they are not logged in will not be compromised. In contrast, if the XSS is stored, then the user is guaranteed to be logged in at the time they encounter the exploit.

How to find and test for this

Many stored XSS vulnerabilities can be found using Burp Suite's [web vulnerability scanner](https://portswigger.net/burp/vulnerability-scanner).

Testing for stored XSS vulnerabilities manually can be challenging. You need to test all relevant "entry points" via which attacker-controllable data can enter the application's processing, and all "exit points" at which that data might appear in the application's responses.

Entry points into the application's processing include:

* Parameters or other data within the URL query string and message body.
* The URL file path.
* HTTP request headers that might not be exploitable in relation to [reflected XSS](https://portswigger.net/web-security/cross-site-scripting/reflected).
* Any out-of-band routes via which an attacker can deliver data into the application. The routes that exist depend entirely on the functionality implemented by the application: a webmail application will process data received in emails; an application displaying a Twitter feed might process data contained in third-party tweets; and a news aggregator will include data originating on other web sites.

The exit points for stored XSS attacks are all possible HTTP responses that are returned to any kind of application user in any situation.

The first step in testing for stored XSS vulnerabilities is to locate the links between entry and exit points, whereby data submitted to an entry point is emitted from an exit point. The reasons why this can be challenging are that:

* Data submitted to any entry point could in principle be emitted from any exit point. For example, user-supplied display names could appear within an obscure audit log that is only visible to some application users.
* Data that is currently stored by the application is often vulnerable to being overwritten due to other actions performed within the application. For example, a search function might display a list of recent searches, which are quickly replaced as users perform other searches.

To fully identify links between entry and exit points, each variation would need to be tested separately, by entering a specific value into the entry point, navigating directly to the exit point, and determining whether the value appears there. This approach, however, is impractical for applications with more than a few pages.

Instead, a more realistic approach would be to go through the data entry points one by one, entering a specific value into each one and then monitoring the application's responses for cases where the submitted value appears. Relevant application functions, such as comments on blog posts, can be given special consideration. When the submitted value is observed in a response, you must determine whether the data is being stored across multiple requests or simply reflected in the immediate response.

When you have identified links between entry and exit points in the application's processing, each link needs to be specifically tested to detect if a stored XSS vulnerability is present. This involves determining the context within the response where the stored data appears and testing suitable candidate XSS payloads that are applicable to that context. At this point, the testing methodology is broadly the same as for finding [reflected XSS vulnerabilities](https://portswigger.net/web-security/cross-site-scripting/reflected).

***Blind xss (persistent XSS)***

* A type of stored XSS

• But you can’t see the alert(0)

pop up

• This is because it only pops

up on a page you can’t see

• Instead, you use XSS hunter

or crafted payloads to check if

the XSS ever fires

# Server XSS

Server XSS occurs when untrusted user supplied data is included in an HTML response generated by the server. The source of this data could be from the request, or from a stored location. As such, you can have both Reflected Server XSS and Stored Server XSS.

In this case, the entire vulnerability is in server-side code, and the browser is simply rendering the response and executing any valid script embedded in it.

**DOM XSS**

* DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution, such as eval() or innerHTML. This enables attackers to execute malicious JavaScript, which typically allows them to hijack other users' accounts.
* To deliver a DOM-based XSS attack, you need to place data into a source so that it is propagated to a sink and causes execution of arbitrary JavaScript.
* The most common source for DOM XSS is the URL, which is typically accessed with the window. location object. An attacker can construct a link to send a victim to a vulnerable page with a payload in the query string and fragment portions of the URL. In certain circumstances, such as when targeting a 404 page or a website running PHP, the payload can also be placed in the path.

Test for DOM XSS

Testing HTML sinks

* To test for DOM XSS in an HTML sink, place a random alphanumeric string into the source (such as location.search), then use developer tools to inspect the HTML and find where your string appears. Note that the browser's "View source" option won't work for DOM XSS testing because it doesn't take account of changes that have been performed in the HTML by JavaScript. In Chrome's developer tools, you can use Control+F (or Command+F on MacOS) to search the DOM for your string.
* For each location where your string appears within the DOM, you need to identify the context. Based on this context, you need to refine your input to see how it is processed. For example, if your string appears within a double-quoted attribute then try to inject double quotes in your string to see if you can break out of the attribute.
* Note that browsers behave differently with regards to URL-encoding, Chrome, Firefox, and Safari will URL-encode location. Search and location. Hash, while IE11 and Microsoft Edge (pre-Chromium) will not URL-encode these sources. If your data gets URL-encoded before being processed, then an XSS attack is unlikely to work.

Testing JavaScript execution sinks

* Testing JavaScript execution sinks for DOM-based XSS is a little harder. With these sinks, your input doesn't necessarily appear anywhere within the DOM, so you can't search for it. Instead, you'll need to use the JavaScript debugger to determine whether and how your input is sent to a sink.
* For each potential source, such as location, you first need to find cases within the page's JavaScript code where the source is being referenced. In Chrome's developer tools, you can use Control+Shift+F (or Command+Alt+F on MacOS) to search all the page's JavaScript code for the source.
* Once you've found where the source is being read, you can use the JavaScript debugger to add a break point and follow how the source's value is used. You might find that the source gets assigned to other variables. If this is the case, you'll need to use the search function again to track these variables and see if they're passed to a sink. When you find a sink that is being assigned data that originated from the source, you can use the debugger to inspect the value by hovering over the variable to show its value before it is sent to the sink. Then, as with HTML sinks, you need to refine your input to see if you can deliver a successful XSS attack.

Testing for DOM XSS using DOM Invader

* Identifying and exploiting DOM XSS in the wild can be a tedious process, often requiring you to manually trawl through complex, minified JavaScript. If you use Burp's browser, however, you can take advantage of its built-in DOM Invader extension, which does a lot of the hard work for you.

# Client XSS

Client XSS occurs when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call. A JavaScript call is considered unsafe if it can be used to introduce valid JavaScript into the DOM. This source of this data could be from the DOM, or it could have been sent by the server (via an AJAX call, or a page load). The ultimate source of the data could have been from a request, or from a stored location on the client or the server. As such, you can have both Reflected Client XSS and Stored Client XSS.

This redefines XSS into two categories: Server and Client.

Server XSS means that the data comes directly from the server onto the page. For example, the data containing the unsanitized text is from the HTTP response that made up the vulnerable page.

Client XSS means that the data comes from JavaScript which has manipulated the page. So it is JavaScript that has added the unsanitized text to the page, rather than it being in the page at that location when it was first loaded in the browser.

**How to prevent XSS attacks?**

* **Filter input on arrival.** At the point where user input is received, filter as strictly as possible based on what is expected or valid input.
* **Encode data on output.** At the point where user-controllable data is output in HTTP responses, encode the output to prevent it from being interpreted as active content. Depending on the output context, this might require applying combinations of HTML, URL, JavaScript, and CSS encoding.
* **Use appropriate response headers.** To prevent XSS in HTTP responses that aren't intended to contain any HTML or JavaScript, you can use the Content-Type and X-Content-Type-Options headers to ensure that browsers interpret the responses in the way you intend.
* **Content Security Policy.** As a last line of defense, you can use Content Security Policy (CSP) to reduce the severity of any XSS vulnerabilities that still occur.

[What is reflected XSS (cross-site scripting)? Tutorial & Examples | Web Security Academy (portswigger.net)](https://portswigger.net/web-security/cross-site-scripting/reflected)

[The Ultimate Beginners Guide to XSS Vulnerability (brightsec.com)](https://brightsec.com/blog/cross-site-scripting-xss/)

[What is stored XSS (cross-site scripting)? Tutorial & Examples | Web Security Academy (portswigger.net)](https://portswigger.net/web-security/cross-site-scripting/stored)

[What is DOM-based XSS (cross-site scripting)? Tutorial & Examples | Web Security Academy (portswigger.net)](https://portswigger.net/web-security/cross-site-scripting/dom-based)

[owasp - Server XSS vs. client XSS - Stack Overflow](https://stackoverflow.com/questions/28392997/server-xss-vs-client-xss)