**Cross-site scripting attack**

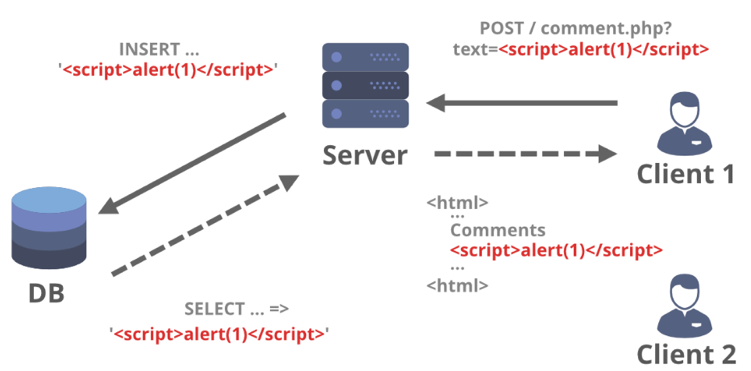
Cross Site Scripting attack is client-side code injection attack. It is a type of injection attack.

Attacker injects malicious codes into a legitimate website. It is also called as XSS.

Execution of XSS attack requires user interaction.

The exploitation of XSS against a user can lead to various consequences such as account compromise, account deletion, privilege escalation, malware infection and many more.

An 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 trusted, and 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 sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.



**Types of XSS:**

There are 3 types of XSS attacks, however we will only explain 2 types of XSS attacks

* Reflected XSS
* Stored XSS

**Reflected XSS:** Reflected (Non-Persistent) XSS attacks occur when the malicious payload is included in the request sent to the vulnerable web application and is then reflected such that the server’s HTTP response consists of the payload.

Attackers leverage social engineering techniques such as phishing attacks to make the victim include the malicious script in their request to the webserver. The victim’s browser then executes the malicious script as the HTTP response.

**How to test?**

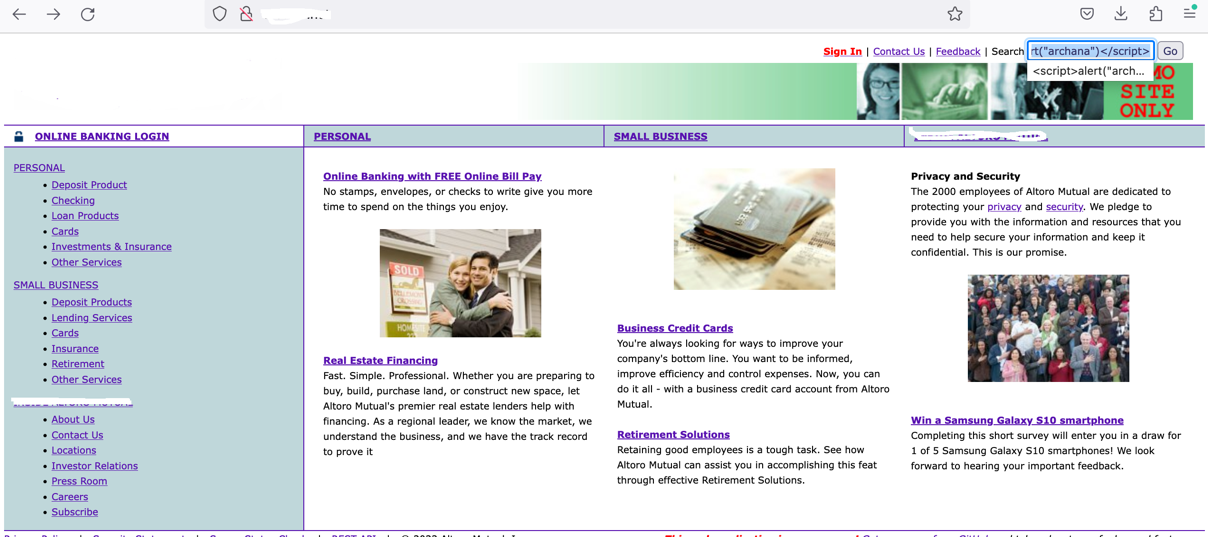
**Using Dynamic Scanner:** we can use dynamic scanner to identify the reflected XSS.

Dynamic scanner has a signatures to detect the reflected xss vulnerability present in your application.

XSS signatures will be injected in all area of your application [Input fields, hidden fields, URL paths etc.]

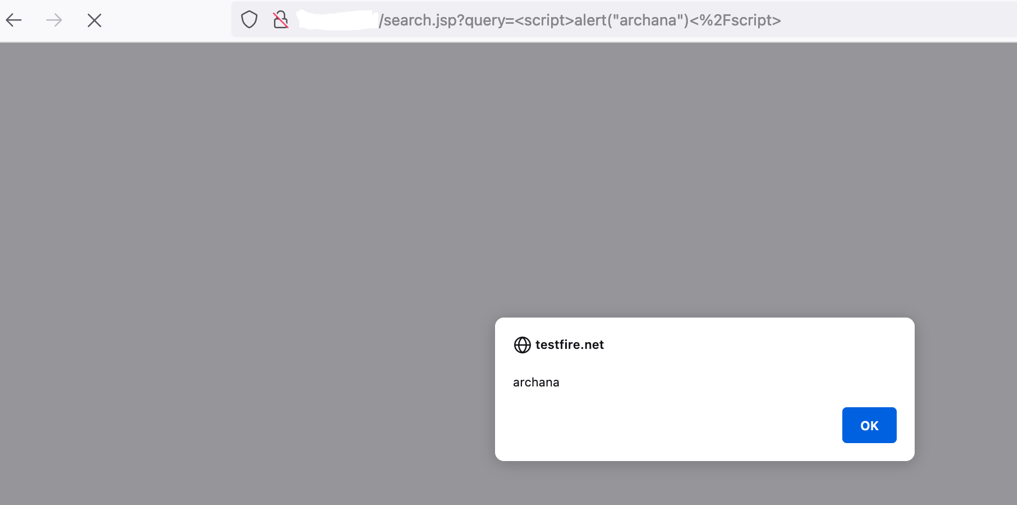
**Manual testing:** XSS payloads can be injected manually to identify if XSS vulnerabilities are present in your application.

Check the below pictures, where we injected a <script>alert("archana")</script> payload.

****

Once we injected the XSS payload we see the below result and see reflected XSS is present in this application.

We have also injected the XSS payload in the URL as well and we see same result.

****

**Code Scanning[SAST] :** You can perform code scanning by using any good code scanner.

Code scanners: Checkmarx, Fortify etc.

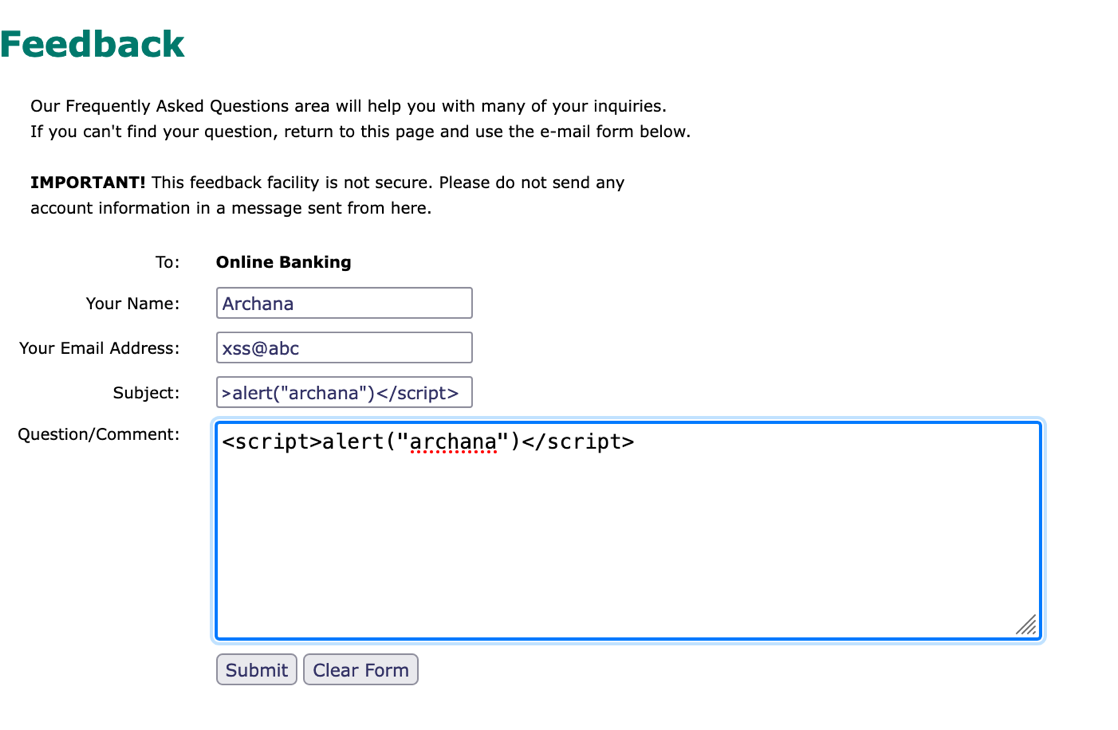
**Stored XSS:** An attacker uses Stored XSS to inject malicious payload, into the web application. When application is vulnerable to stored XSS, then the injected malicious code will be permanently stored by the target application, for example within a database or server.

For example, an attacker may enter a malicious script into a user input field such as a blog comment field or in a forum post of feedback section.

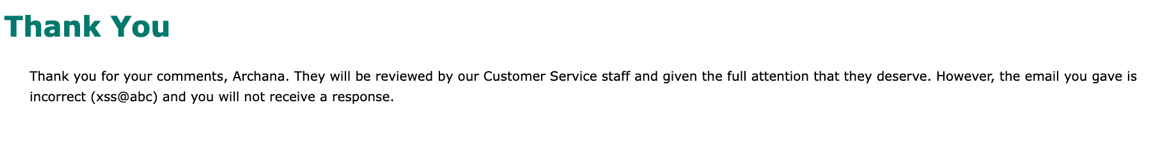
When a victim opens the affected web page in his browser, the stored malicious payload will be executed while page is loading.

Stored XSS is more dangerous, any user who is opening the affected page the payload/code will be executed.

So in the below website in the Feedback section we have injected the <script>alert("archana")</script> payload



This application is vulnerable to Stored XSS, the payload which we have injected will be executed when any other user will open this page.



**How to prevent XSS:**

1. Input Validation: To prevent XSS attacks, your application must validate all the input data, make sure that only the allowlisted data is allowed

* If a user submits a URL that will be returned in responses, validating that it starts with a safe protocol such as HTTP and HTTPS. Otherwise someone might exploit your site with a harmful protocol like javascript or data.
* If a user supplies a value that it expected to be numeric, validating that the value actually contains an integer.
* Validating that input contains only an expected set of characters.

1. Output Encoding: ensure that all variable output in a page is encoded before it is returned to the user.

In an HTML context, you should convert non-whitelisted values into HTML entities:

* < converts to: &lt;
* converts to: &gt;

In a JavaScript string context, non-alphanumeric values should be Unicode-escaped:

* < converts to: \u003c
* converts to: \u003e

1. Mitigating XSS using [content security policy](https://portswigger.net/web-security/cross-site-scripting/content-security-policy) (CSP)

Content security policy (CSP) is the last line of defence against cross-site scripting.

If your XSS prevention fails, you can use CSP to mitigate XSS by restricting what an attacker can do.

CSP lets you control various things, such as whether external scripts can be loaded and whether inline scripts will be executed.

To deploy CSP you need to include an HTTP response header called Content-Security-Policy with a value containing your policy.

An example CSP is as follows:

default-src 'self'; script-src 'self'; object-src 'none'; frame-src 'none'; base-uri 'none';