Module-3

Browser Security Principles: The Same-Origin Policy (SOP)

1. Defining the Same-Origin Policy

The **Same-Origin Policy (SOP)** is a crucial security mechanism in web browsers that restricts how documents or scripts from one origin can interact with resources from another origin. It prevents malicious websites from accessing sensitive data from other sites visited by the user.

1.1 What is an Origin?

An **origin** is defined by the combination of:

- **Protocol (Scheme)** – HTTP, HTTPS, FTP, etc.
- **Host (Domain Name or IP Address)** – e.g., example.com
- **Port** – Default (80 for HTTP, 443 for HTTPS) or custom

Example:

- https://example.com:443/page1.html
- https://example.com:443/page2.html
  These have the **same origin** because they share the same protocol, domain, and port.

But:

- http://example.com (Different protocol)
- https://example.com:8443 (Different port)
- https://sub.example.com (Different subdomain) These have **different origins** and are restricted by SOP.

1.2 Purpose of SOP

- Prevents **Cross-Site Scripting (XSS)** and **Cross-Site Request Forgery (CSRF)**
- Ensures scripts on one website **cannot access data** from another website unless explicitly allowed
- Protects user data such as cookies, local storage, and session information

---

2. Exceptions to the Same-Origin Policy

While SOP is strict, there are several exceptions and workarounds to allow controlled cross-origin communication:

2.1 Cross-Origin Resource Sharing (CORS)

- A mechanism that allows servers to declare which origins can access their resources
- Uses the Access-Control-Allow-Origin HTTP header to specify permitted origins
- Example:

Access-Control-Allow-Origin: https://trustedwebsite.com

2.2 JSONP (JSON with Padding)

- A technique that bypasses SOP by loading data as a script tag
- Commonly used before CORS became widely supported
- Example:

```
<script src="https://api.example.com/data?callback=myFunction"></script>
```

The server wraps the response in myFunction(), allowing execution in the requesting page.

2.3 PostMessage API

- Allows safe cross-origin communication between windows, iframes, or workers
- Example:

```
window.postMessage("Hello from another origin!", "https://trusted.com");
```

2.4 Content Security Policy (CSP)

- Controls which external resources (scripts, styles, frames, etc.) a webpage can load
- Example:

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

2.5 WebSockets

- WebSockets are **not** restricted by SOP but require an explicit handshake process.

2.6 SameSite Cookies

- Helps prevent CSRF by restricting when cookies are sent with cross-origin requests
- Values: Strict, Lax, None
- Example:

```
Set-Cookie: sessionId=abc123; Secure; HttpOnly; SameSite=Strict
```

---

3. Final Thoughts on the Same-Origin Policy

- **Essential for security:** SOP is fundamental in preventing unauthorized access to user data across different origins.
- **Not a complete solution:** Must be used alongside other security measures like CSP, CORS, and authentication mechanisms.
- **Evolving security landscape:** Modern web security techniques (e.g., sandboxing, isolation models like Site Isolation) enhance protection while allowing safe cross-origin interactions.
- **Web developers must be cautious:** Improper CORS configurations or overly permissive policies can lead to vulnerabilities.

Examples of Same-Origin and Cross-Origin Cases

✅ Same-Origin Example (Allowed Access)

URL 1	URL 2	Same Origin?
https://example.com/page1.html	https://example.com/page2.html	✅ Yes

Reason: The protocol (https), domain (example.com), and port (443 default for HTTPS) are identical.

✅ **JavaScript Example (Allowed Access)**

```
fetch("https://example.com/data.json")  // Allowed
  .then(response => response.json())
  .then(data => console.log(data));
```

Here, the script from example.com can access data.json because both have the same origin.

❌ **Cross-Origin Example (Blocked by SOP)**

URL 1 (Source)	URL 2 (Request)	Same Origin?
https://example.com/index.html	https://api.anotherdomain.com/data	❌ No

Reason: The domain (example.com vs. api.anotherdomain.com) is different. By default, JavaScript from example.com **cannot** access api.anotherdomain.com.

❌ **JavaScript Example (Blocked by SOP)**

```
fetch("https://api.anotherdomain.com/data")  // Blocked by SOP
  .then(response => response.json())
  .catch(error => console.error("Blocked!", error));
```

Browser Console Error:

```
Access to fetch at 'https://api.anotherdomain.com/data' from origin 'https://example.com' has been blocked by CORS policy.
```

This happens because api.anotherdomain.com does not allow requests from example.com.

---

2. Exceptions to the Same-Origin Policy

To enable cross-origin communication securely, several mechanisms exist.

2.1 Cross-Origin Resource Sharing (CORS)

A web server can allow cross-origin requests by setting specific HTTP headers.

✅ **Example: CORS Header Allowing All Origins (Less Secure)**

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

This allows **any website** to request resources from the server. This is risky and should only be used for public APIs.

✅ **Example: CORS Header Allowing a Specific Origin (More Secure)**

```
Access-Control-Allow-Origin: https://example.com
```

```
Access-Control-Allow-Methods: GET, POST
```

This allows only https://example.com to access the resources using GET or POST methods.

JavaScript Example (Allowed with CORS):

```
fetch("https://api.anotherdomain.com/data", {
  method: "GET",
  headers: {
    "Origin": "https://example.com"
  }
})
  .then(response => response.json())
  .then(data => console.log(data));
```

If the server has configured CORS properly, this request will succeed.

2.2 JSONP (JSON with Padding) [Older Technique]

JSONP is a workaround for SOP by using <script> tags (which are not restricted by SOP).

✅ **Example: JSONP Request to Fetch Data from Another Origin**

```
<script src="https://api.anotherdomain.com/data?callback=myFunction"></script>
<script>
  function myFunction(data) {
    console.log("Received data:", data);
  }
</script>
```

The server returns:

```
myFunction({"message": "Hello, world!"});
```

This method is rarely used today because of security concerns (e.g., it can be exploited for XSS attacks).

2.3 PostMessage API (Cross-Origin Communication Between Windows or Iframes)

The postMessage API allows controlled communication between different origins.

✅ **Example: Sending a Message from an Embedded iFrame**

Parent Page (example.com)

```
const iframe = document.getElementById("myIframe");
iframe.contentWindow.postMessage("Hello from parent!", "https://anotherdomain.com");
```

Inside the iFrame (anotherdomain.com)

```
window.addEventListener("message", (event) => {
  if (event.origin === "https://example.com") {
    console.log("Message received:", event.data);
  }
});
```

This method is commonly used for secure cross-origin interactions, such as embedding widgets or payment gateways.

2.4 WebSockets (Not Restricted by SOP)

WebSockets provide real-time communication and are not blocked by SOP.

✅ **Example: WebSocket Connection**

```
const socket = new WebSocket("wss://chatserver.com");


socket.onopen = () => {
  console.log("Connected to WebSocket server");
  socket.send("Hello Server!");
};


socket.onmessage = (event) => {
  console.log("Received:", event.data);
};
```

Since WebSockets require a **handshake**, they are not subject to SOP restrictions like HTTP requests.

2.5 SameSite Cookies (Mitigating CSRF Risks)

SOP does not protect against **Cross-Site Request Forgery (CSRF)**, so cookies should be configured properly.

✅ **Example: Setting a Secure Cookie with SameSite Attribute**

```
Set-Cookie: sessionId=abc123; Secure; HttpOnly; SameSite=Strict
```

- SameSite=Strict: Prevents cookies from being sent with cross-origin requests.
- HttpOnly: Prevents JavaScript access to the cookie.
- Secure: Ensures cookies are only sent over HTTPS.

---

3. Final Thoughts on the Same-Origin Policy

✅ **Key Takeaways:**

- **SOP is essential for web security** to prevent unauthorized data access between sites.
- **CORS, JSONP, PostMessage, and WebSockets** allow controlled cross-origin communication.
- **Misconfigured CORS policies** can lead to security risks like **data leaks or CSRF**.
- **Modern techniques** like postMessage, SameSite cookies, and CSP help balance security and flexibility.

Scenario:

We have two domains:

1. siteA.com – A website that has sensitive user data.
2. siteB.com – A malicious website trying to access data from siteA.com.

The browser's **Same-Origin Policy** will block siteB.com from accessing resources from siteA.com via JavaScript.

Example: Attempted Cross-Origin Request (Blocked by SOP)

Legitimate Page (siteA.com)

```
<!-- siteA.com/index.html -->

<!DOCTYPE html>

<html>

<head>

   <title>Secure Site</title>

</head>

<body>

   <h1>Welcome to Site A</h1>

   <p id="secretData">User Secret: 12345XYZ</p>

</body>

</html>
```

Malicious Page (siteB.com)

```
<!-- siteB.com/hack.html -->

<!DOCTYPE html>

<html>

<head>

  <title>Hacker Site</title>

  <script>

    // Attempt to fetch data from siteA.com

    fetch("https://siteA.com/index.html")

    .then(response => response.text())

    .then(data => {

      console.log("Hacked Data:", data); // SOP will block this

    })

    .catch(error => console.error("Blocked by SOP:", error));

  </script>

</head>

<body>

  <h1>Malicious Site</h1>

</body>

</html>
```

Expected Outcome:

- The browser blocks the request with a **CORS error**, preventing siteB.com from reading the contents of siteA.com due to **Same-Origin Policy**.
- The attacker cannot access the user's private data.

How to Allow Cross-Origin Access (Using CORS - Controlled Exception)

If siteA.com wants to allow access to its data from other origins, it must explicitly enable **CORS (Cross-Origin Resource Sharing)**:

Site A - Enabling CORS via Server Headers

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

Access-Control-Allow-Methods: GET, POST
```

or in an Express.js server:

js

```
app.use((req, res, next) => {
    res.header("Access-Control-Allow-Origin", "*"); // Allow all origins
    res.header("Access-Control-Allow-Methods", "GET, POST");
    next();
});
```

This allows controlled cross-origin access while maintaining security.