19

SAME-ORIGIN POLICY VULNERABILITIES

Chapter 3 introduced the same-origin policy (SOP), one of the fundamental defenses deployed in modern web applications. The SOP restricts how a script originating from one site can interact with the resources of a different site, and it's critical in preventing many common web vulnerabilities.

But websites often loosen the SOP in order to have more flexibility. These controlled and intended SOP bypasses can have adverse effects, as attackers can sometimes exploit misconfigurations in these techniques to bypass the SOP. These exploits can cause private information leaks and often lead to more vulnerabilities, such as authentication bypass, account takeover, and large data breaches. In this chapter, we'll discuss how applications relax or work around the SOP and how attackers can exploit these features to endanger the application.

Mechanisms

Here's a quick review of how the SOP works. Because of the SOP, a script from page A can access data from page B only if the pages are of the same origin. Two URLs are said to have the *same origin* if they share the same protocol, hostname, and port number. Modern web applications often base their authentication on HTTP cookies, and servers take action based on the cookies included automatically by the browser. This makes the SOP especially important. When the SOP is implemented, malicious web pages won't be able to take advantage of the cookies stored in your browser to access your private information. You can read more about the details of the SOP in Chapter 3.

Practically, the SOP is often too restrictive for modern web applications. For example, multiple subdomains or multiple domains of the same organization wouldn't be able to share information if they followed the policy. Since the SOP is inflexible, most websites find ways to relax it. This is often where things go wrong.

For instance, imagine that you are an attacker trying to smuggle information out of a banking site, *a.example.com*, and find a user's account number. You know that a user's banking details are located at *a.example.com/user_info*. Your victim is logged into the banking site at *a.example.com* and is also visiting your site, *attacker.com*, in the same browser.

Your site issues a GET request to *a.example.com/user_info* to retrieve the victim's personal information. Since your victim is logged into the bank, their browser automatically includes their cookies in every request it sends to *a.example.com*, even if the request is generated by a script on your malicious site. Unfortunately, because of the SOP, the victim's browser won't allow your site to read data returned from *a.example.com*.

But now, say you realize that *a.example.com* passes information to *b.example.com* via SOP bypass techniques. If you can find out the technique used and exploit it, you might be able to steal the victim's private information on the banking site.

The simplest way for websites to work around the SOP is to change the origin of a page via JavaScript. Setting the origin of two pages to the same domain using `document.domain` in the pages' JavaScript will enable the pages to share resources. For example, you can set the domain of both *a.example.com* and *b.example.com* to *example.com* so that they can interact:

```
document.domain = "example.com"
```

However, this approach has its limitations. First, you can only set the *document.domain* of a page to a superdomain; for example, you can set the origin of *a.example.com* to *example.com*, but not to *example2.com*. Therefore, this method will work only if you want to share resources with superdomains or sibling subdomains.

Exploiting Cross-Origin Resource Sharing

Because of these limitations, most sites use Cross-Origin Resource Sharing (CORS) to relax the SOP instead. CORS is a mechanism that protects the data of the server. It allows servers to explicitly specify a list of origins that are allowed to access its resources via the HTTP response header `Access-Control-Allow-Origin`.

For example, let's say we're trying to send the following JSON blob located at *a.example.com/user_info* to *b.example.com*:

```
{"username": "vickieli", "account_number": "12345"}
```

Under the SOP, *b.example.com* won't be able to access the JSON file, because *a.example.com* and *b.example.com* are of different origins. But using CORS, the user's browser will send an `Origin` header on behalf of *b.example.com*:

```
Origin: https://b.example.com
```

If *b.example.com* is part of an allowlist of URLs with permission to access resources on *a.example.com*, *a.example.com* will send the browser the requested resource along with an `Access-Control-Allow-Origin` header. This header will indicate to the browser that a specific origin is allowed to access the resource:

```
Access-Control-Allow-Origin: b.example.com
```

The application can also return the `Access-Control-Allow-Origin` header with a wildcard character (`*`) to indicate that the resource on that page can be accessed by any domain:

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

On the other hand, if the origin of the requesting page isn't allowed to access the resource, the user's browser will block the requesting page from reading the data.

CORS is a great way to implement cross-origin communication. However, CORS is safe only when the list of allowed origins is properly defined. If CORS is misconfigured, attackers can exploit the misconfiguration and access the protected resources.

The most basic misconfiguration of CORS involves allowing the `null` origin. If the server sets `Access-Control-Allow-Origin` to `null`, the browser will allow any site with a `null` origin header to access the resource. This isn't safe because any origin can create a request with a `null` origin. For instance, cross-site requests generated from a document using the `data:` URL scheme will have a `null` origin.

Another misconfiguration is to set the `Access-Control-Allow-Origin` header to the origin of the requesting page without validating the requestor's origin. If the server doesn't validate the origin and returns an `Access-Control-Allow-Origin` for any origin, the header will completely bypass the SOP, removing all limitations on cross-origin communication.

In summary, if the server sets the `Access-Control-Allow-Origin` header to `null` or to arbitrary origins of the requesting page, it allows attackers to smuggle information offsite:

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

Another exploitable misconfiguration occurs when a site uses weak regexes to validate origins. For example, if the policy checks only if an origin URL starts with *www.example.com*, the policy can be bypassed using an origin like *www.example.com.attacker.com*.

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

An interesting configuration that isn't exploitable is setting the allowed origins to the wildcard (*). This isn't exploitable because CORS doesn't allow credentials, including cookies, authentication headers, or client-side certificates, to be sent with requests to these pages. Since credentials cannot be sent in requests to these pages, no private information can be accessed:

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

Developers can prevent CORS misconfigurations by creating a well-defined CORS policy with a strict allowlist and robust URL validation. For pages containing sensitive information, the server should return the requesting page's origin in the `Access-Control-Allow-Origin` header only if that origin is in the allowlist. For public information, the server can simply use the wildcard * designation for `Access-Control-Allow-Origin`.

Exploiting postMessage()

Some sites work around SOP by using `postMessage()`. This method is a web API that uses JavaScript syntax. You can use it to send text-based messages to another window:

```
RECIPIENT_WINDOW.postMessage(MESSAGE_TO_SEND, TARGET_ORIGIN);
```

The receiving window would then handle the message by using an event handler that will be triggered when the receiving window receives a message:

```
window.addEventListener("message",EVENT_HANDLER_FUNCTION);
```

Since using `postMessage()` requires the sender to obtain a reference to the receiver's window, messages can be sent only between a window and its iframes or pop-ups. That's because only windows that open each other will have a way to reference each other. For example, a window can use `window.open` to refer to a new window it opened. Alternatively, it can use `window.opener` to reference the

window that spawned the current window. It can use `window.frames` to reference embedded iframes, and `window.parent` to reference the parent window of the current iframe.

For example, say we're trying to pass the following JSON blob located at *a.example.com/user_info* to *b.example.com*:

```
{'username': 'vickieli', 'account_number': '12345'}
```

a.example.com can open *b.example.com* and send a message to its window. The `window.open()` function opens the window of a particular URL and returns a reference to it:

```
var recipient_window = window.open("https://b.example.com", b_domain)
recipient_window.postMessage("{'username': 'vickieli', 'account_number': '12345'}", "*");
```

At the same time, *b.example.com* would set up an event listener to process the data it receives:

```
function parse_data(event) {
  // Parse the data
}
window.addEventListener("message", parse_data);
```

As you can see, `postMessage()` does not bypass SOP directly but provides a way for pages of different origins to send data to each other.

The `postMessage()` method can be a reliable way to implement cross-origin communication. However, when using it, both the sender and the receiver of the message should verify the origin of the other side. Vulnerabilities happen when pages enforce weak origin checks or lack origin checks altogether.

First, the `postMessage()` method allows the sender to specify the receiver's origin as a parameter. If the sender page doesn't specify a target origin and uses a wildcard target origin instead, it becomes possible to leak information to other sites:

```
RECIPIENT_WINDOW.postMessage(MESSAGE_TO_SEND, *);
```

In this case, an attacker can create a malicious HTML page that listens for events coming from the sender page. They can then trick users into triggering the `postMessage()` by using a malicious link or fake image and make the victim page send data to the attacker's page.

To prevent this issue, developers should always set the `TARGET_ORIGIN` parameter to the target site's URL instead of using a wildcard origin:

```
recipient_window.postMessage(
"{'username': 'vickieli', 'account_number': '12345'}", "https://b.example.com");
```

On the other hand, if the message receiver doesn't validate the page where the `postMessage()` is coming from, it becomes possible for attackers to

send arbitrary data to the website and trigger unwanted actions on the victim's behalf. For example, let's say that *b.example.com* allows *a.example.com* to trigger a password change based on a `postMessage()`, like this:

```
recipient_window.postMessage(
"{'action': 'password_change', 'username': 'vickieli', 'new_password': 'password'}",
"https://b.example.com");
```

The page *b.example.com* would then receive the message and process the request:

```
function parse_data(event) {
  // If "action" is "password_change", change the user's password
}
window.addEventListener("message", parse_data);
```

Notice here that any window can send messages to *b.example.com*, so any page can initiate a password change on *b.example.com*! To exploit this behavior, the attacker can embed or open the victim page to obtain its window reference. Then they're free to send arbitrary messages to that window.

To prevent this issue, pages should verify the origin of the sender of a message before processing it:

```
function parse_data(event) {
❶ if (event.origin == "https://a.example.com"){

    // If "action" is "password_change", change the user's password
  }
}
window.addEventListener("message", parse_data);
```

This line ❶ verifies the origin of the sender by checking it against an acceptable origin.

Exploiting JSON with Padding

JSON with Padding (JSONP) is another technique that works around the SOP. It allows the sender to send JSON data as JavaScript code. A page of a different origin can read the JSON data by processing the JavaScript.

To see how this works, let's continue with our previous example, where we're trying to pass the following JSON blob located at *a.example.com/user_info* to *b.example.com*:

```
{"username": "vickieli", "account_number": "12345"}
```

The SOP allows the HTML `<script>` tag to load scripts across origins, so an easy way for *b.example.com* to retrieve data across origins is to load the data as a script in a `<script>` tag:

```
<script src="https://a.example.com/user_info"></script>
```

This way, *b.example.com* would essentially be including the JSON data block in a script tag. But this would cause a syntax error because JSON data is not valid JavaScript:

```
<script>
    {"username": "vickieli", "account_number": "12345"}
</script>
```

JSONP works around this issue by wrapping the data in a JavaScript function, and sending the data as JavaScript code instead of a JSON file.

The requesting page includes the resource as a script and specifies a callback function, typically in a URL parameter named `callback` or `jsonp`. This callback function is a predefined function on the receiving page ready to process the data:

```
<script src="https://a.example.com/user_info?callback=parseinfo"></script>
```

The page at *a.example.com* will return the data wrapped in the specified callback function:

```
parseinfo({"username": "vickieli", "account_number": "12345"})
```

The receiving page would essentially be including this script, which is valid JavaScript code:

```
<script>
    parseinfo({"username": "vickieli", "account_number": "12345"})
</script>
```

The receiving page can then extract the data by running the JavaScript code and processing the `parseinfo()` function. By sending data as scripts instead of JSON data, JSONP allows resources to be read across origins. Here's a summary of what happens during a JSONP workflow:

1. The data requestor includes the data's URL in a script tag, along with the name of a callback function.
2. The data provider returns the JSON data wrapped within the specified callback function.
3. The data requestor receives the function and processes the data by running the returned JavaScript code.

You can usually find out if a site uses JSONP by looking for script tags that include URLs with the terms *jsonp* or *callback*.

But JSONP comes with risks. When JSONP is enabled on an endpoint, an attacker can simply embed the same script tag on their site and request the data wrapped in the JSONP payload, like this:

```
<script src="https://a.example.com/user_info?callback=parseinfo"></script>
```

If a user is browsing the attacker's site while logged into *a.example.com* at the same time, the user's browser will include their credentials in this request and allow attackers to extract confidential data belonging to the victim.

This is why JSONP is suitable for transmitting only public data. While JSONP can be hardened by using CSRF tokens or maintaining an allow-list of referer headers for JSONP requests, these protections can often be bypassed.

Another issue with JSONP is that site *b.example.com* would have to trust site *a.example.com* completely, because it's running arbitrary JavaScript from *a.example.com*. If *a.example.com* is compromised, the attacker could run whatever JavaScript they wanted on *b.example.com*, because *b.example.com* is including the file from *a.example.com* in a `<script>` tag. This is equivalent to an XSS attack.

Now that CORS is a reliable option for cross-origin communication, sites no longer use JSONP as often.

Bypassing SOP by Using XSS

Finally, XSS is essentially a full SOP bypass, because any JavaScript that runs on a page operates under the security context of that page. If an attacker can get a malicious script executed on the victim page, the script can access the victim page's resources and data. Therefore, remember that if you can find an XSS, you've essentially bypassed the SOP protecting that page.

Hunting for SOP Bypasses

Let's start hunting for SOP bypass vulnerabilities by using what you've learned! SOP bypass vulnerabilities are caused by the faulty implementation of SOP relaxation techniques. So the first thing you need to do is to determine whether the target application relaxes the SOP in any way.

Step 1: Determine If SOP Relaxation Techniques Are Used

You can determine whether the target is using an SOP-relaxation technique by looking for the signatures of each SOP-relaxation technique. When you're browsing a web application, open your proxy and look for any signs of cross-origin communication. For example, CORS sites will often return HTTP responses that contain an `Access-Control-Allow-Origin` header. A site could be using `postMessage()` if you inspect a page (for example, by right-clicking it in Chrome and choosing **Inspect**, then navigating to **Event Listeners**) and find a `message` event listener (Figure 19-1).

And a site could be using JSONP if you see a URL being loaded in a `<script>` tag with a callback function:

```
<script src="https://a.example.com/user_info?callback=parseinfo"></script>
<script src="https://a.example.com/user_info?jsonp=parseinfo"></script>
```

If you see clues of cross-origin communication, try the techniques mentioned in this chapter to see if you can bypass the SOP and steal sensitive info from the site!

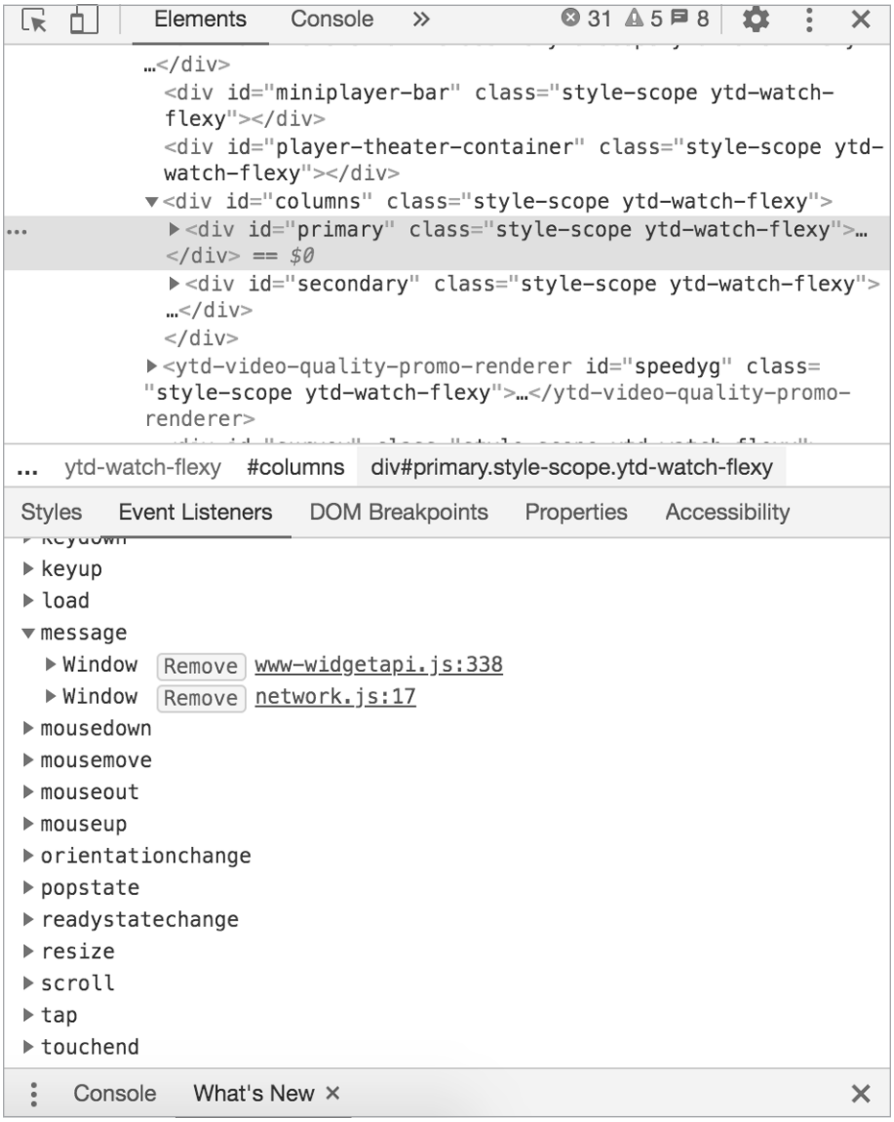

Figure 19-1: Finding the event listeners of a page in the Chrome browser

Step 2: Find CORS Misconfiguration

If the site is using CORS, check whether the `Access-Control-Allow-Origin` response header is set to `null`.

```
Origin: null
```

If not, send a request to the site with the origin header `attacker.com`, and see if the `Access-Control-Allow-Origin` in the response is set to `attacker.com`. (You can add an Origin header by intercepting the request and editing it in a proxy.)

```
Origin: attacker.com
```

Finally, test whether the site properly validates the origin URL by submitting an Origin header that contains an allowed site, such as *www.example.com.attacker.com*. See if the `Access-Control-Allow-Origin` header returns the origin of the attacker's domain.

```
Origin: www.example.com.attacker.com
```

If one of these `Access-Control-Allow-Origin` header values is returned, you have found a CORS misconfiguration. Attackers will be able to bypass the SOP and smuggle information offsite (Figure 19-2).

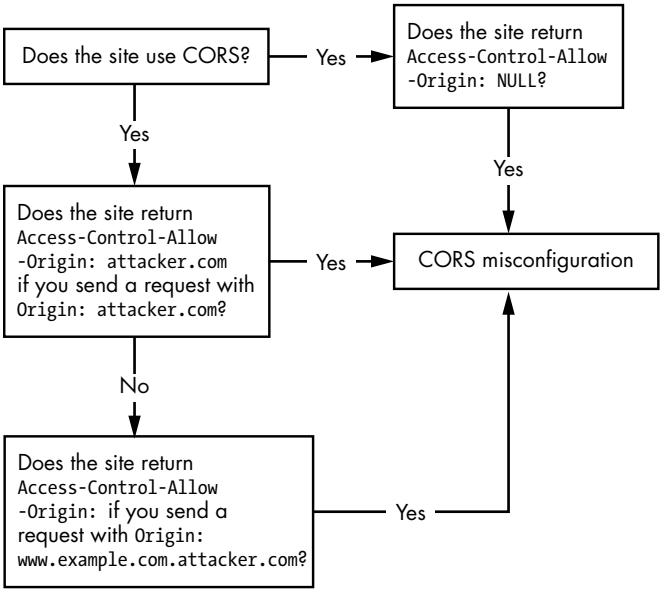

Figure 19-2: Is the site vulnerable to a CORS misconfiguration vulnerability?

Step 3: Find postMessage Bugs

If the site is using `postMessage`, see if you can send or receive messages as an untrusted site. Create an HTML page with an iframe that frames the targeted page accepting messages. Try to send messages to that page that

trigger a state-changing behavior. If the target cannot be framed, open it as a new window instead:

```
var recipient_window = window.open("https://TARGET_URL", target_domain)
recipient_window.postMessage("RANDOM MESSAGE", "*");
```

You can also create an HTML page that listens for events coming from the target page, and trigger the postMessage from the target site. See if you can receive sensitive data from the target page.

```
var sender_window = window.open("https://TARGET_URL", target_domain)

function parse_data(event) {
  // Run some code if we receive data from the target
              }
window.addEventListener("message", parse_data);
```

Step 4: Find JSONP Issues

Finally, if the site is using JSONP, see if you can embed a script tag on your site and request the sensitive data wrapped in the JSONP payload:

```
<script src="https://TARGET_URL?callback=parseinfo"></script>
```

Step 5: Consider Mitigating Factors

When the target site does not rely on cookies for authentication, these SOP bypass misconfigurations might not be exploitable. For instance, when the site uses custom headers or secret request parameters to authenticate requests, you might need to find a way to forge those to exfiltrate sensitive data.

Escalating the Attack

An SOP-bypass bug often means that attackers can read private information or execute action as other users. This means that these vulnerabilities are often of high severity before any escalation attempts. But you can still escalate SOP-bypass issues by automation or by pivoting the attack using the information you've found. Can you harvest large amounts of user data by automating the exploitation of the SOP bypass? Can you use the information you've found to cause more damage? For example, if you can extract the security questions of a victim, can you use that information to completely take over the user's account?

Many researchers will simply report CORS misconfigurations without showing the impact of the vulnerability. Consider the impact of the issue before sending the report. For instance, if a publicly readable page is served with a null `Access-Control-Allow-Origin` header, it would not cause damage

to the application since that page does not contain any sensitive info. A good SOP-bypass report will include potential attack scenarios and indicate how attackers can exploit the vulnerability. For instance, what data can the attacker steal, and how easy would it be?

Finding Your First SOP Bypass Vulnerability!

Go ahead and start looking for your first SOP bypass. To find SOP-bypass vulnerabilities, you will need to understand the SOP relaxation techniques the target is using. You may also want to become familiar with JavaScript in order to craft effective POCs.

1. Find out if the application uses any SOP relaxation techniques. Is the application using CORS, `postMessage`, or JSONP?
2. If the site is using CORS, test the strength of the CORS allowlist by submitting test `Origin` headers.
3. If the site is using `postMessage`, see if you can send or receive messages as an untrusted site.
4. If the site is using JSONP, try to embed a script tag on your site and request the sensitive data wrapped in the JSONP payload.
5. Determine the sensitivity of the information you can steal using the vulnerability, and see if you can do something more.
6. Submit your bug report to the program!