Chapter 96: Same Origin Policy & Cross-Origin Communication

Same-Origin policy is used by web browsers to prevent scripts to be able to access remote content if the remote address has not the same **origin** of the script. This prevents malicious scripts from performing requests to other websites to obtain sensitive data.

The **origin** of two addresses is considered the same if both URLs have the same *protocol*, *hostname* and *port*.

Section 96.1: Safe cross-origin communication with messages

The `window.postMessage()` method together with its relative event handler `window.onmessage` can be safely used to enable cross-origin communication.

The `postMessage()` method of the target `window` can be called to send a message to another `window`, which will be able to intercept it with its `onmessage` event handler, elaborate it, and, if necessary, send a response back to the sender window using `postMessage()` again.

Example of Window communicating with a children frame

- Content of `http://main-site.com/index.html`:

```
<!-- ... -->
<iframe id="frame-id" src="http://other-site.com/index.html"></iframe>
<script src="main_site_script.js"></script>
<!-- ... -->
```

- Content of `http://other-site.com/index.html`:

```
<!-- ... -->
<script src="other_site_script.js"></src>
<!-- ... -->
```

- Content of `main_site_script.js`:

```
// Get the <iframe>'s window
var frameWindow = document.getElementById('frame-id').contentWindow;

// Add a listener for a response
window.addEventListener('message', function(evt) {

    // IMPORTANT: Check the origin of the data!
    if (event.origin.indexOf('http://other-site.com') == 0) {

        // Check the response
        console.log(evt.data);
        /* ... */
    }
});

// Send a message to the frame's window
frameWindow.postMessage(/* any obj or var */, '*');
```

- Content of `other_site_script.js`:

```
window.addEventListener('message', function(evt) {

    // IMPORTANT: Check the origin of the data!
    if (event.origin.indexOf('http://main-site.com') == 0) {

        // Read and elaborate the received data
        console.log(evt.data);
        /* ... */

        // Send a response back to the main window
        window.parent.postMessage(/* any obj or var */, '*');
    }
});
```

Section 96.2: Ways to circumvent Same-Origin Policy

As far as client-side JavaScript engines are concerned (those running inside a browser), there is no straightforward solution available for requesting content from sources other than the current domain. (By the way, this limitation does not exist in JavaScript-server tools such as Node JS.)

However, it is (in some situations) indeed possible to retrieve data from other sources using the following methods. Please do note that some of them may present hacks or workarounds instead of solutions production system should rely on.

Method 1: CORS

Most public APIs today allow developers to send data bidirectionally between client and server by enabling a feature called CORS (Cross-Origin Resource Sharing). The browser will check if a certain HTTP header (`Access-Control-Allow-Origin`) is set and that the requesting site's domain is listed in the header's value. If it is, then the browser will allow establishing AJAX connections.

However, because developers cannot change other servers' response headers, this method can't always be relied on.

Method 2: JSONP

JSON with **P**adding is commonly blamed to be a workaround. It is not the most straightforward method, but it still gets the job done. This method takes advantage of the fact that script files can be loaded from any domain. Still, it is crucial to mention that requesting JavaScript code from external sources is **always** a potential security risk and this should generally be avoided if there's a better solution available.

The data requested using JSONP is typically JSON, which happens to fit the syntax used for object definition in JavaScript, making this method of transport very simple. A common way to let websites use the external data obtained via JSONP is to wrap it inside a callback function, which is set via a `GET` parameter in the URL. Once the external script file loads, the function will be called with the data as its first parameter.

```
<script>
function myfunc(obj){
    console.log(obj.example_field);
}
</script>
<script src="http://example.com/api/endpoint.js?callback=myfunc"></script>
```

The contents of `http://example.com/api/endpoint.js?callback=myfunc` might look like this:

```
myfunc({"example_field":true})
```

The function always has to be defined first, otherwise it won't be defined when the external script loads.