# **Bypassing Weak CSPs**

Now that we have discussed CSPs, CSP directives, and CSP directive values, let us jump into exploiting and bypassing weak CSPs.

## **Bypassing Weak CSPs**

CSPs can be used to add a defense-in-depth measure to prevent XSS vulnerabilities. However, just because a web application implements a CSP does not automatically mean it is protected from all XSS attacks. If the CSP is weak, it may be possible to bypass it. As such, it is crucial to analyze a web application's CSP for potential bypasses.

Let us start by looking at the following CSP:

Code: http

Content-Security-policy: default-src 'none'; img-src 'self'; style-src \*; font-src \*; script-src 'self' https://\*.google.com;

This CSP allows images to be loaded from the origin itself, styles and fonts from anywhere, scripts from the origin itself, and any subdomain of google.com. All other resources cannot be loaded due to the default-src 'none' directive.

Suppose we attempt injecting a simple alert pop-up in the web application as a proof of concept:

Code: html

<script>alert(1)</script>

Due to the CSP, the alert pop-up is not shown; instead, the browser's JavaScript console will print the following error message:

Code: html

Refused to execute inline script because it violates the following Content Security Policy directive: "script-src 'self' https://\*.google.com". Either the 'unsafe-inline' keyword, a hash ('sha256-bhHHL3z2vDgxUt0W3dWQOrprscmda2Y5pLsLg4GF+pI='), or a nonce ('nonce-...') is required to enable inline execution.

While this defensive technique may seem secure at first glance, it can be bypassed with [JSONP](https://www.w3schools.com/js/js_json_jsonp.asp). JSONP refers to a technique that retrieves data across different origins without issues due to the Same-Origin policy. The basic idea of JSONP is to use script tags to retrieve data across origins since they are excluded from the Same-Origin policy. For instance, assume a web application http://vulnerablesite.htb wants to retrieve data from the endpoint http://someapi.htb/stats, which returns the following JSON data:

Code: json

{'clicks': 1337}

If the API does not have CORS configured, the web application cannot access the response to a cross-origin request due to the Same-Origin policy. However, since script tags are excluded from the Same-Origin policy, the web application can load the data by using the following HTML tag on its page:

Code: html

<script src="http://someapi.htb/stats"></script>

Now, this by itself is not practical, as the web application needs to process the data somehow. Let us assume the web application implements a function called processData for this purpose. But as is, there is no way to pass the received data to this function. That is where JSONP comes into play. If the API supports JSONP, it will read a GET parameter on the endpoint sending the data and adjust the response accordingly. This parameter is often called callback. Assume we call the endpoint http://someapi.htb/stats?callback=processData. This results in the API sending the following response:

Code: js

processData({'clicks': 1337})

The web application can now insert the following script tag on its page:

Code: html

<script src="http://someapi.htb/stats?callback=processData"></script>

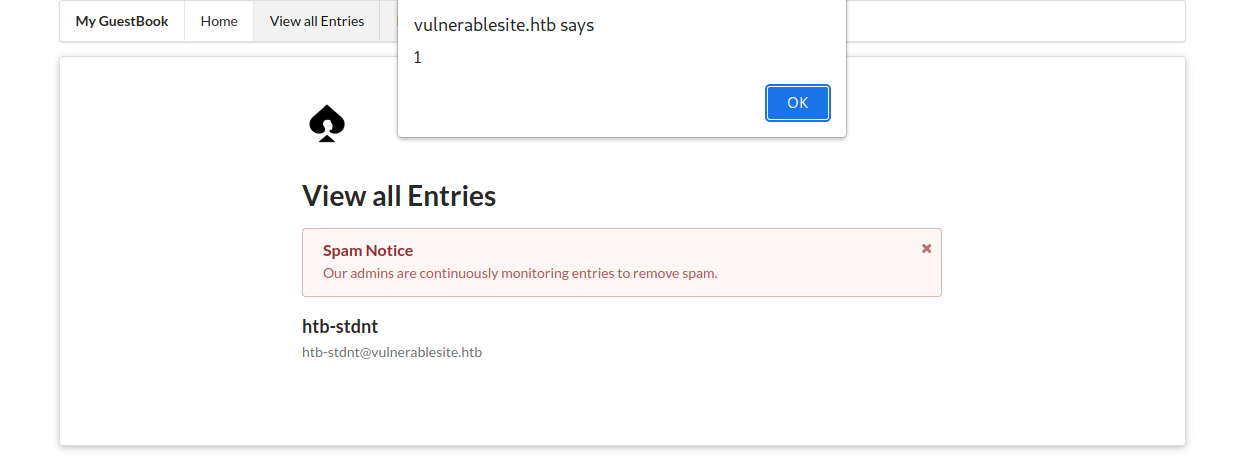
This results in the web application's function processData being called on the data fetched cross-origin from the API without violating the Same-Origin policy or the need for CORS.

Since JSONP endpoints allow the caller to specify a function that is called, they can be used to dynamically create JavaScript code sent out by the domain offering the JSONP endpoint. As such, JSONP can be used to bypass CSPs. Google offers multiple different JSONP endpoints. The [JSONBee](https://github.com/zigoo0/JSONBee) GitHub repository lists many JSONP endpoints that can be used to bypass CSPs. We can use the following Google JSONP endpoint to bypass the CSP above:

Code: html

<script src="https://accounts.google.com/o/oauth2/revoke?callback=alert(1);"></script>

Posting this entry to the guestbook, the alert pop-up is triggered, thus bypassing the CSP:



Another common weakness is the assumption that the 'self' value is automatically safe. For instance, consider the following CSP:

Code: http

Content-Security-policy: default-src 'none'; img-src 'self'; style-src \*; script-src 'self';

This time, scripts can only be loaded from the origin itself. Assuming the origin does not offer a JSONP endpoint, this seems safe. However, consider a scenario where a web application allows users to upload files. If arbitrary file types are allowed, an attacker can upload a .js file. It is then possible to exploit an XSS by loading the uploaded payload from the origin itself:

Code: html

<script src="/uploads/avatag.jpg.js"></script>

Generally, an assessment of a CSP depends on the concrete CSP itself and the web application's functionality. As we have seen, setting the script-src directive to 'self' can be unsafe if the web application implements a file upload functionality. As such, it is crucial to assess the CSP in the context of the concrete web application in which it is implemented