CONTENT SECURITY POLICY:

CSP policy is supported inside (directive types):

- The HTTP response header (Content-Security-Policy)
- The html tag -> <meta>

CSP's functionality categories:

1. **Limitation of Resource loading:**

Trusted sources to load,

Directives-Set = **{default-src, script-src, style-src, img-src, media-src, font-src, frame-src, object-src, child-src, worker-src ,manifest-src}**

2. **Limitation of Auxiliary URL-based:**

Trusted URLs, which are allowed to interact by the document, in this way some sort of attacks can be prevented, for example:

❑ Clickjacking can be prevented by placing

```
Content-Security-Policy: frame-ancestors <source>;
```

Allowed sources for frame-ancestors, with these sources we can frame the document:

- ' self '
- ' none '
- Host, like: https://website.com
- Scheme, like: https: or http:

❑ Post-XSS can be prevented by placing:

```
Content-Security-Policy: base-uri <source>;
```

And `Content-Security-Policy: form-action <source>;`

For example: a website has the following origin: https:www.website.com : 19

```
<meta http-equiv="Content-Security-Policy" content="form-    action 'none'">

   <form action="javascript:alert(1)" method="post">
         .

         .

    </form>
```

This will cause an Error, because the form-action determines that no source is trusted to be targets of the html tag: <form>

3. **Miscellaneous con nement and hardening options.**

- ❑ Preventing any mixed content bugs and support https by preventing loading any assets over http if the website support https:

All of this could be done by placing the following headers:

```
Content-Security-Policy: block-all-mixed-content;
```

```
Upgrade-Insecure-Requests: 1
```

- ❑ Restriction of the set of plugins, which are able to embedded, by placing the following header:

```
Content-Security-Policy: plugin-types <type>/<subtype>;
```

- ❑ Allowing a sandbox for the requested resource, by placing the following header:

  ```
  Content-Security-Policy: sandbox;
  ```

  Or

  ```
    Content-Security-Policy: sandbox <value>;
  ```

  Set of allowed values = {allow-downloads, allow-downloads-without-user-activation, allow- forms, allow-modals, allow-orientation-lock, allow-pointer-lock, allow-popups, allow-popups-to-escape-sandbox, allow-presentation, allow-same-origin, allow-scripts, allow-storage-access-by-user-activation, allow-top-navigation, allow-top-navigation-by-user-activation, allow-top-navigation-to-custom-protocols}

Script execution controlling:

Again, the syntax of the script-src header is:

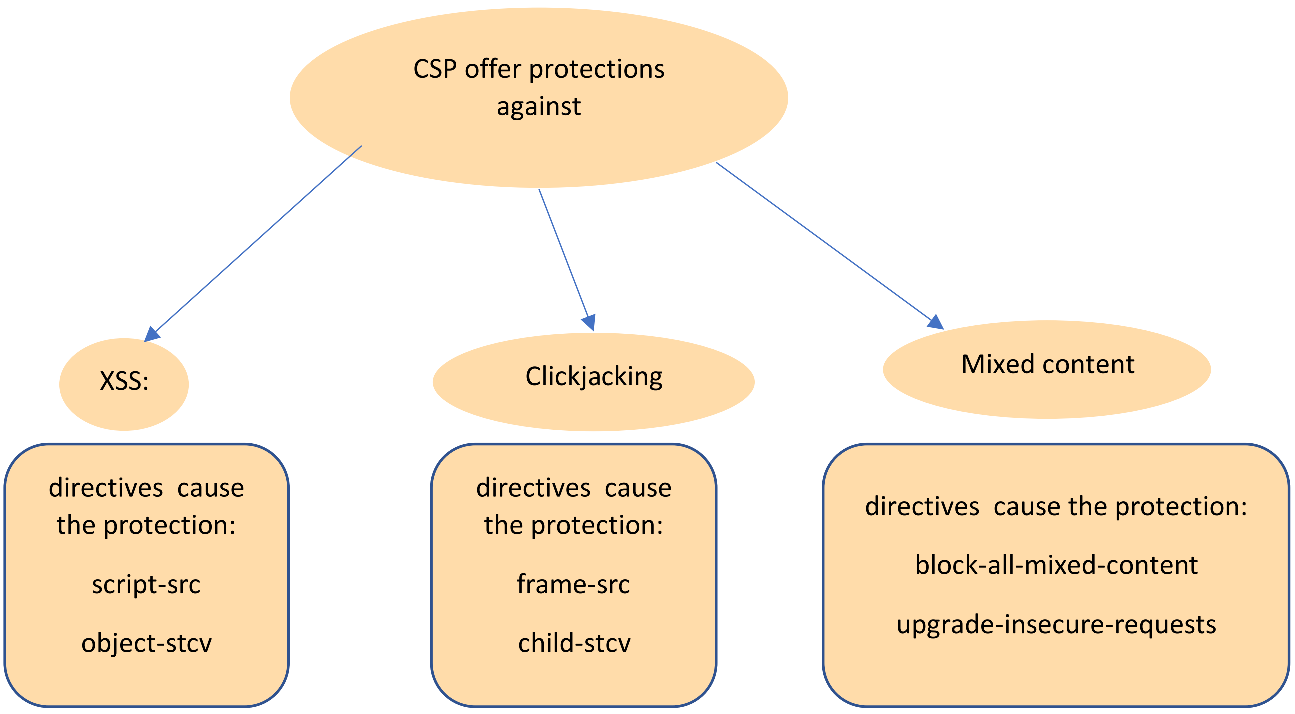
```
Content-Security-Policy: script-src <source>;
```

Possible sources-keywords:

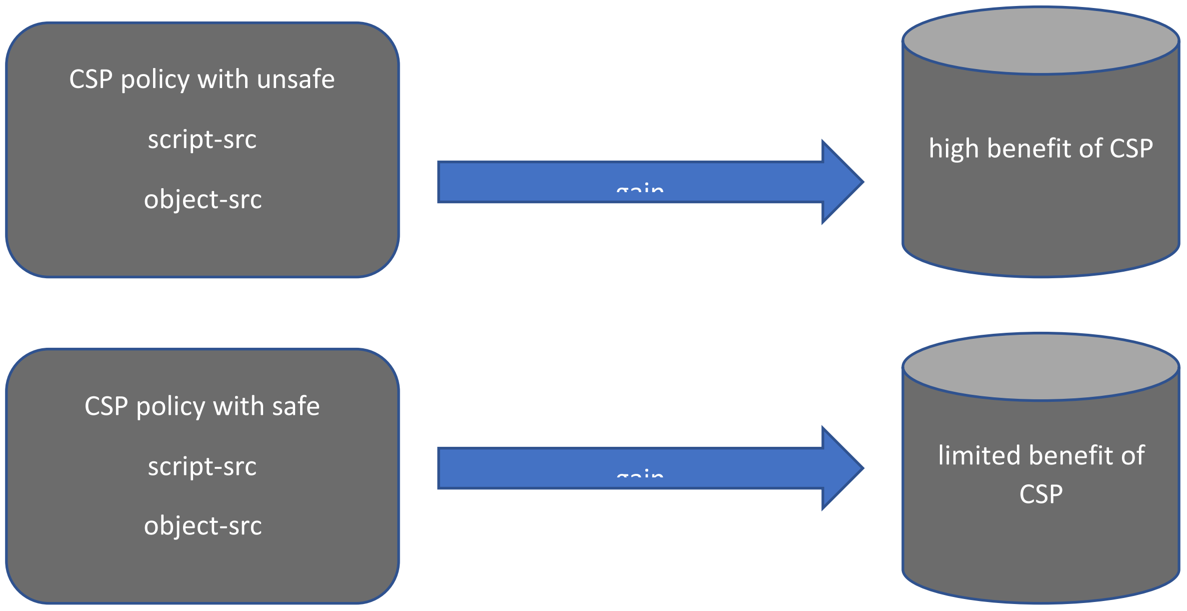
- ♦ unsafe-inline: allows inline scripts blocks and event handlers to be executed, cons: XXS attacks
- ♦ unsafe-eval : allows JavaScript APIs to be executed, like:

  ```
  eval(), Function(), setTimeout(), setInterval()
  ```

- ♦ nonce-random-value : allows any script that has the correct random-value to be executed
- ♦ sha256-nGADFW23F : allows any inline script that has the same digets in the policy to be executed


CSP offer protections against
XSS:
Clickjacking
Mixed content
directives cause the protection:
script-src
object-stcv
directives cause the protection:
frame-src
child-stcv
directives cause the protection:
block-all-mixed-content
upgrade-insecure-requests

Defending against XXS:



We can assume that a CSP policy offers a protection from content-injection attacks, if any only if the following three conditions are fulfilled:

- script-src and object-src are defined in the policy (or default-src).
- script-src :
  - 1) If unsafe-inline (unless accompanied by a nonce) is not included in source list (this mean inline scripts blocks and event handlers cannot be executed).
  - 2) Data are not allowed in source.

- Endpoints and unsafe libraries are not included in script-src and object-src.

Script execution bypasses:

JSONP:

GET-Parameters of URLs: callback, cb, json, jsonp.

let assume that there is a server, which can enable JSONP capabilities, but to enable it the server will expects a parameter 'callback', for example we have the following JSNOP request:

http://www.website.net/ahmad?callback=func

Func = function(info) {

Alert(info.name)

};

Without JSONP => output: {name: 'Ahmad'}

With JSONP => output: func({name: Ahmad});

The important disadvantage of JSONP is that we will lose a lot of control of the request.

A policy with JSONP interface, which is included in the domain whitelisted can be used by an attacker to load the endpoint as a script with an attacker-controlled callback.

Bypassing CSP could be done, when the famous library "angular.js" is loaded.

MIME types or Multipurpose Internet Mail Extensions (Hint: they are used by the browsers):

Syntax: *type/subtype*

For example : image/png

To see all types visit:

https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

A script can be executed if the following conditions are fulfilled:

1- Response => parsed as a JavaScript without any syntax errors
2- Response => contain attacker-controlled data before the first runtime error

Responses, which can cause bypassing CSP (response types):

◊ CSV-data, where attacker-controlled contents are included.
◊ Error messages echoing request parameters
◊ User file uploads, even if their contents are properly HTML-escaped or sanitized.

If an application has CSP policy with endpoints => this will lead to policy bypassing.

This security problem will also affect all domains in script-src, which include third parties and CDNs.

Bypassing CSP caused by path restrictions:

For example:

The CSP determines which origins are trusted to execute:

Content-Security-Policy: script-src website.com parially-trusted.com/modul/playwright.js

The redirection is able to use but to load resource from the whitelisted origins, which has not the same path in CSP policy:

Redirect is a parameter for the server.

<script src="// website.com?redirect=partially-trusted.org/hacker/byebyedude.js">

IsCSPReportOnly (hint: it is not supported inside <meta>):

To send a report (in JSON format) via http Post request to a specific url.

Syntax:

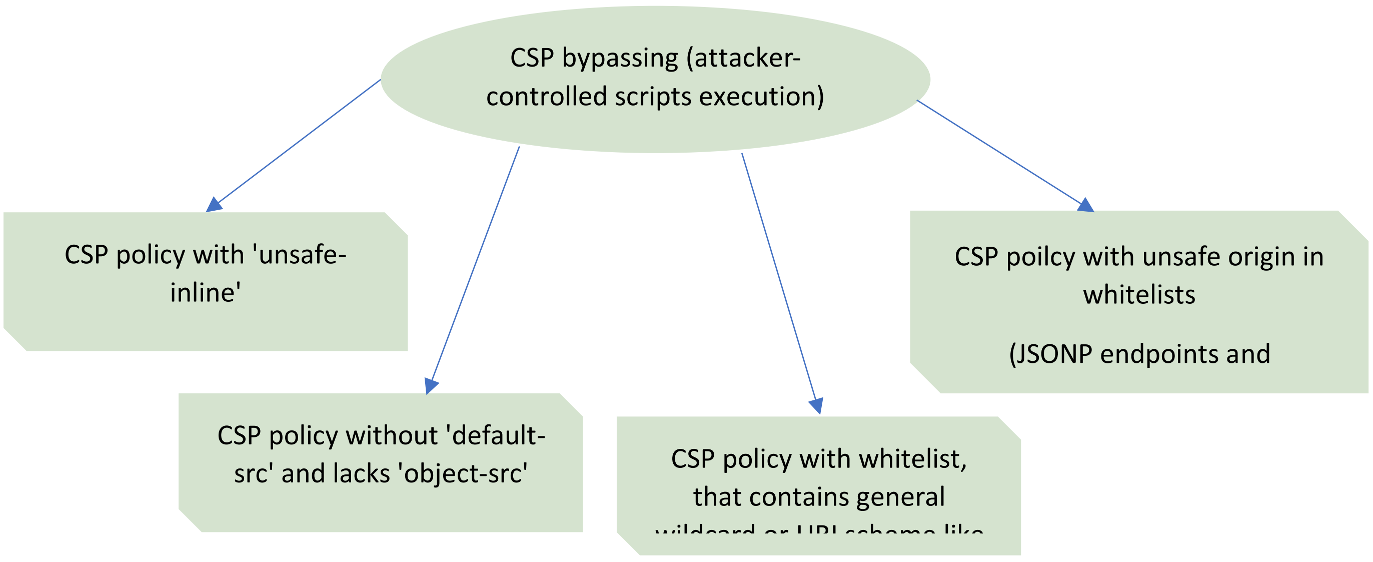
```
Content-Security-Policy-Report-Only: <policy-directive>; <policy-directive>
```

For example:

```
Content-Security-Policy-Report-Only: default-src https:; report-uri /csp-violation-report-endpoint/
```

Or inside the CSP header with the directive "report-uri"

```
content-security-policy : report-uri https://www.instagram.com/security/csp_report/; default-src 'self' https://www.instagram.com;
```


CSP bypassing (attacker-controlled scripts execution)
CSP policy with 'unsafe-inline'
CSP policy without 'default-src' and lacks 'object-src'
CSP policy with whitelist, that contains general
CSP poilcy with unsafe origin in whitelists
(JSONP endpoints and

Most of the policies do not offer the required protection that prevent XXS attacks.

Maintaining the security of a policy is hard, when the policy contains a very long whitelist, and to solve this difficult problem we must use nonce, with nonce scripts can be whitelisted individually, because of the random unpredictable value of the nonce, it's impossible for the attacker to inject a valid script pointing to the JSNOP endpoint.

The user can receive tow policies in the same http response header due to capability of the browsers to enforce multiple policies, for example:

Content-Security-Policy:

<!-- whitelist - based CSP -->

script-src ‘self’ https://website.com

default-src 'self’

<!-- nonce - based CSP -->

script-src 'nonce-fuzzer11'

Dynamic Scripts:

Impacts of adding ‘strict-dynamic’:

- ❖ Additional scripts via *non-"parser-inserted"* script elements are allowed now to execute,

for example, we have the a CSP with following script-src header:

script-src ‘nonce-fuzzy11’ ‘strict-dynamic’; default-src ‘self’;

<script-src=”/dynamic.js” nanoce=”fuzzy11”></script>

Because of the keyword ‘strict-dynamic’ /dynamic.js is now able load additional scripts:

var script = document.createElement(‘script’);

script.src = https://ahamd.com/run.js;

document.body.appendchild(s);

We notice the URL of the additional scripts is not in the script-src whitelist.

- ❖ Other script-src whitelist entries are ignored.

The advantage is that, to call createElement() the JavaScript must be executed firstly and with the random unguessable value of nonce, the attacker will not be able predict the value of nonce and hence the attacker cannot do any injections-attacks.

The following set of sources {‘unsafe-inline’, ‘self’, Host based source lists, Protocol source lists} will be ignored by the browser which supports “strict-dynamic” when it’s used.

Where can we insert the keyword strict-dynamic ?

In the following directives:

- script-src
- default-src

Parser-inserted scripts:

Usage of APIs: ‘strict-dynamic’ will block any scripts that have been added using document.write(),

In spite of the fact that these additional scripts point to a whitelisted source.