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.
What we’ll be talking about

01 WHAT IS CSP
02 WHAT’S IN A POLICY?
03 COMMON MISTAKES
04 BYPASSING CSP
05 A NEW WAY OF DOING CSP
06 THE FUTURE OF CSP
07 SUCCESS STORIES
08 Q & A
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...

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.

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

CSP 2 specification: https://www.w3.org/TR/CSP/
CSP 3 draft: https://w3c.github.io/webappsec-csp/
WHAT’S IN A POLICY?

CSP directives
Many, for many different problems.

default-src
base-uri
connect-src
img-src
font-src
child-src
script-src
media-src
frame-ancestors
style-src
object-src
plugin-types
report-uri

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

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

Content-Security-Policy

default-src 'self';
script-src 'self' yep.com;
report-uri /csp_violation_logger;

money.example.com
<img src="cat.png">
CSP allows

money.example.com
<script src="/yep.com/x.js">
CSP allows

money.example.com
""'"'"""'alert(42)"'/script>
CSP blocks

yep.com
CSP blocks

attacker.com
CSP blocks

source not whitelisted

money.example.com/csp_violations_logger
inline script not allowed

DEMO
**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.
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

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

Trivial mistakes

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

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

Bypasses

```
'>'<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

";'><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>

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';
```

Same for `object-src`.

Bypass

```html
"'><script src="/user_upload/evil_cat.jpg.js"></script>
```
BYPASSING CSP [1/5]

Whitelist bypasses

JSONP-like endpoint in whitelist

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

Bypass

"">\n<\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!

DEMO
AngularJS library in whitelist

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

<div ng-app ng-csp>{{1336 + 1}}</div>

Also works without user interaction, e.g. by combining with JSONP endpoints or other JS libraries.
```

```html
"<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)>
```
BYPASSING CSP [4/5]
AngularJS is a problem

Powerful JS frameworks are 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.
BYPASSING CSP [5/5]

Path relaxation

Path relaxation due to open redirect in whitelist

```
object-src 'none';
```

Spec: "To avoid leaking path information cross-origin (as discussed in Homakov's Using Content-Security-Policy for Evil), the matching algorithm ignores path component of a source expression if the resource loaded is the result of a redirect."
CSP EVALUATOR
"A Tool to Rule Them All"

Paste CSP

default-src 'self' * 127.0.0.1 https://[2a00:79e0:1b:2:66b:5fd9:dc72:f00e]foobar https://someDomainNotGoogle.com;
img-src 'self' https://data:; 
report-uri https://csp.withgoogle.com/csp/test1

- Check
- Example
A NEW WAY OF DOING CSP

Strict nonce-based CSP

Strict **nonce-based** policy

```plaintext
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

CSP allows

money.example.com

CSP allows

yep.com
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;
```

---

```html
<img src="cat.png">

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

<script nonce="r4nd0m"
src="//yep.com/x.js">

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

"'"'><script>alert(42)
```

**DEMO**
From the CSP3 specification

The 'unsafe-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 'unsafe-dynamic'

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

```html
script-src 'nonce-r4nd0m' 'unsafe-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.
- **unsafe-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 'unsafe-dynamic'.

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

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

CSP3 compatible browser (unsafe-dynamic support)
script-src 'nonce-r4nd0m' 'unsafe-dynamic' 'unsafe-inline' https;
object-src 'none';

CSP2 compatible browser (nonce support) - No-op fallback
script-src 'nonce-r4nd0m' 'unsafe-dynamic' 'unsafe-inline' https;
object-src 'none';

CSP1 compatible browser (no nonce support) - No-op fallback
script-src 'nonce-r4nd0m' 'unsafe-dynamic' 'unsafe-inline' https;
object-src 'none';

Dropped by CSP2 and above in presence of a nonce
Dropped by CSP3 in presence of 'unsafe-dynamic'
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.

THE GOOD, THE OK, THE UGLY

Nonce support

'unsafe-dynamic' support

::(}
SUCCESS STORIES
'unsafe-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/unsafe-dynamic
Q & A
We would love to get your feedback!

QUESTIONS?
@mikispag
@we1x
#{unsafedynamic}
{lwe,mikispag}@google.com