Content Security Policy

And mitigating Cross-site Scripting vulnerabilities

Joseph Fields
M.Sc Computer Science - December 2016

Introduction

- HTML and Javascript power billions of websites visited daily by users around the world
- Four major browsers (Chrome, Firefox, IE, Safari) take this code and render it accurately and securely for the user
- HTML (passive, display markup mostly) and Javascript (active scripting language) are delivered in the same document (DOM)

Introduction

```
<html>
<p>I am a paragraph.</p>
<script type="text/javascript">
alert("I am code.");
</script>
```

Figure 1: Example HTML+Javascript Markup

Introduction

Delivering both the display markup in the same document with active scripting presents some not-so unique challenges:

- Separation of concerns
- Principle of least privilege

What is unique about web app security is the massively distributed scale..

(apps being executed simultaneously by millions+ of users)

Introduction

Content delivered in the same document:

- Display markup (HTML)
- Style rules (CSS)
- Javascript (Code)
- User input

Which one doesn’t belong?

Introduction

It is ultimately up to the developer to know how to handle user input..

ESCAPE!

PHP: htmlspecialchars, etc*

*It is EASY to mess this up. Just like encryption, it is best to use a well-vetted library to do this for you safely.

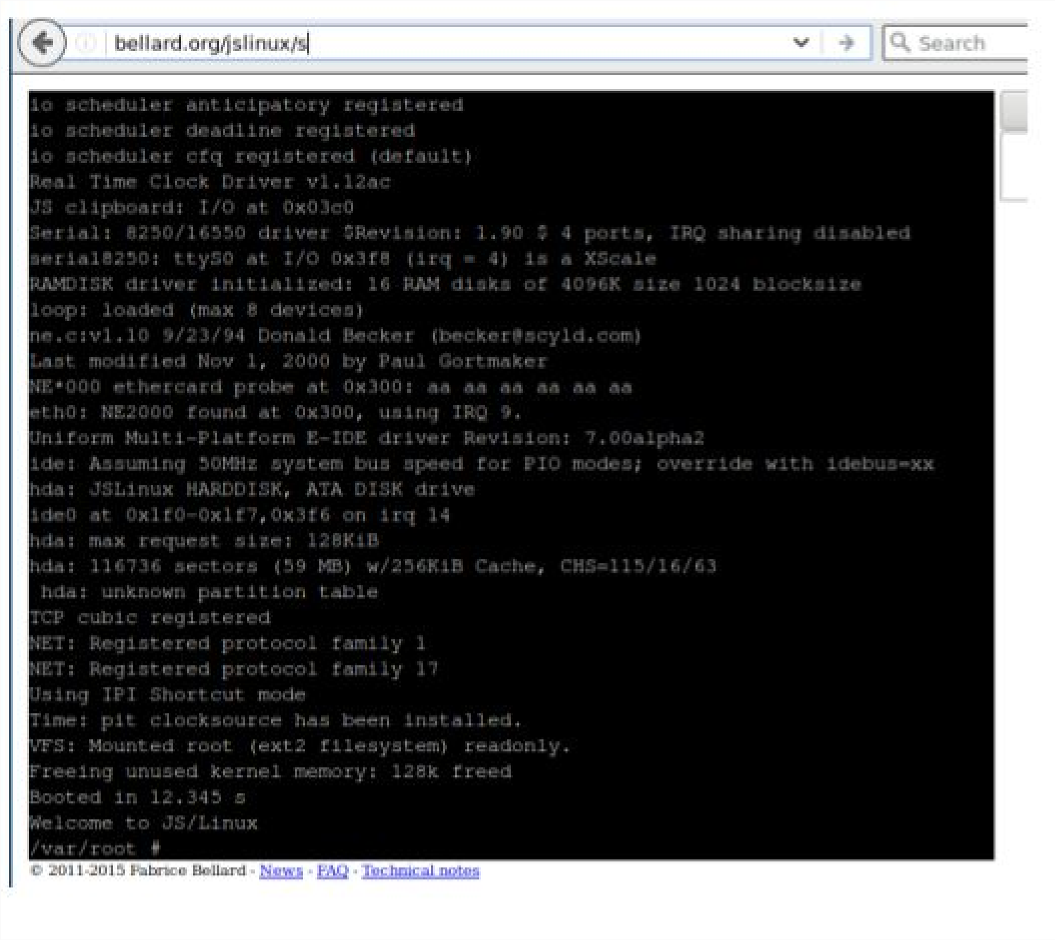
Introduction

Javascript IS executed within a browser-managed sandbox (So, damage to the local OS/files is limited)

Historically, “CORS” policy restricts Javascript permissions to the “same origin”.

However, this is insufficient for the modern web.

Javascript



Note: Within the browser sandbox, you can run Linux!

Cross-Site Scripting

Discovered in ~2000, still going pretty strong in 2016.

“http://vuln.host/index.php?name=<script>location.href=’http://evil.host/’;</script>”

<html><body><p>Hello, <?php echo $_GET[‘name’]; ?></p></html>

Cross-Site Scripting

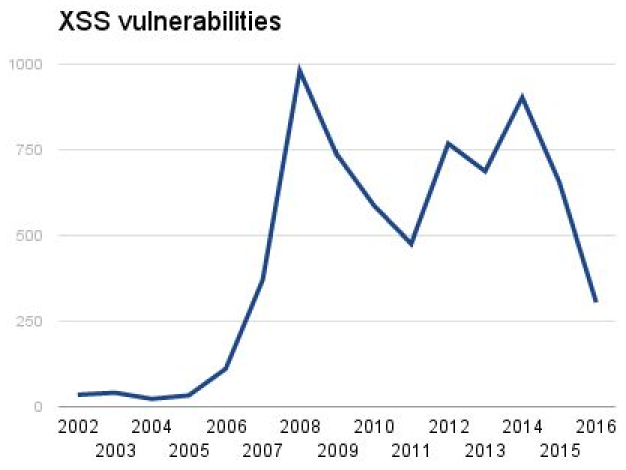
“http://vuln.host/index.php?name=<script>location.href='http://evil.host/';</script>”

The ORIGIN of the script in the “name” parameter is NOT vuln.host, yet it bypasses CORS.

(OK - technically it is, it is just not escaped as user input.)

Cross-Site Scripting

Discovered in ~2000, still going pretty strong in 2016.



Content Security Policy

CSP1 published by W3C in 2012, and CSP2 in 2014. CSP3 is being developed.

A mechanism to set granular policies for where resources can be loaded from and whether they can be executed as script.

- Set via **HTTP Header** (Content-Security-Policy: XXX)
- Or HTML META tag: <meta http-equiv="Content-Security-Policy" content="XXX">

HTTP Header is the recommended method, META tags are not guaranteed to be applied to an entire document.

Content Security Policy 2

Syntax: directive value1 value2;

Values can be **‘self’**, **‘none’**, a **fully-qualified URL**, a **wildcard URL**, random **nonce-**, or cryptographic **sha256-** hash.

Content Security Policy 2

Directives:

base-uri, child-src, connect-src, default-src, font-src, form-action, frame-ancestors, img-src, media-src, object-src, plugin-types, script-src, and style-src.

Font-src, img-src, media-src, style-src restrict the URIs from which content (font, images, audio/video, CSS) is loaded.

Content Security Policy 2

Directives:

Default-src specifies the default policy.

Base-uri restricts the <base /> tag (which would affect relative URLs throughout the page)

Child-src restricts frame and iframe content locations.

Frame-ancestors - restricts the URLs (“the parents”) that can embed frames within a page.

Content Security Policy 2

Directives:

Connect-src restricts javascript and worker network connections.

Form-action - restricts the URLs that forms can be submitted to.

Object-src - restricts the URLs that can serve third-party objects (Flash, etc)

Plugin-types - restricts the type of plugins that can be invoked.

Content Security Policy 2

NOTE: object-src is important because several types of third-party objects can also execute Javascript within the overall document context.

This is a common method to bypass CSP policies, and it is recommended to disable object's completely by specifying 'none'.

Content Security Policy 2

Directives:

Script-src - Restricts the URLs that can load and execute scripts.

‘**Strict-dynamic**’ can be added to propagate the policy through to scripts loaded by the root script.

‘**Unsafe-inline**’ and ‘**unsafe-eval**’ can be specified to allow inline scripts and JS eval(). **Without a nonce, this effectively disables XSS protection**

Sandbox - Enables a more restrictive sandbox (disables script, popups, etc)

Content Security Policy 2

CSP can be enabled in “report only” mode by changing the Header name to:

“Content-Security-policy-Report-Only”

Report-uri - will POST a JSON object to the specified URL when a violation of any defined policy occurs.

Content Security Policy 2

Reports:


```
{
  "csp-report": {
    "document-uri": "http://example.com/signup.html",
    "referrer": "",
    "blocked-uri": "http://example.com/css/style.css",
    "violated-directive": "style-src cdn.example.com",
    "original-policy": "default-src 'none'; style-src cdn.example.com; report-uri /_/csp-reports"
  }
}
```

Real-world CSP:

Courtesy of PayPal.com

content-security-policy:
default-src 'self' https://*.paypal.com https://*.paypalobjects.com https://nexus.ensighten.com 'unsafe-inline';

frame-src 'self' https://*.paypal.com https://*.paypalobjects.com https://www.youtube.com/embed/ https://www.paypal-donations.com https://www.paypal-donations.co.uk https://*.qa.missionfish.org https://www.youtube-nocookie.com https://www.xoom.com;

script-src 'self' https://*.paypal.com https://*.paypalobjects.com https://www.youtube.com/iframe_api https://s.ytimg.com/yts/jsbin/ https://*.t.eloqua.com https://img.en25.com/i/elqCfg.min.js https://nexus.ensighten.com/ 'unsafe-inline' 'unsafe-eval';

connect-src 'self' https://*.paypal.com https://*.paypalobjects.com https://*.salesforce.com https://*.force.com https://*.eloqua.com https://nexus.ensighten.com https://storelocator.api.where.com https://api.paypal-retaillocator.com https://nominatim.openstreetmap.org https://www.paypal-biz.com
https://app.getsentry.com/;

img-src 'self' * data:;
object-src 'self' https://*.paypal.com https://*.paypalobjects.com;
font-src 'self' https://*.paypalobjects.com;"

Content Security Policy 2

Example CSP Violation:

Problems

- Ineffective Policies
- Inline is bad.
- Wildcards are bad.
- Whitelists (especially for public CDNs) are bad.
- CSP can get unwieldy for big web apps QUICK
- Low adoption

Problems: Ineffective Policies

Google researchers studied many CSP policies ~ a year ago (paper published 2016), **found 94% did it wrong.**

Meaning, the policies were "trivial to bypass"

Problems: Inline scripts

Inlined Javascript (or stylesheets) are discouraged for many reasons. They also make security very difficult:

<p>Hello, <script>alert(“Bad code.”);</script></p>

<script>alert(“Good code.”);</script>

How can anyone know which script is OK to run, and which isn’t?

Problems: Inline scripts

Google recommended to use random nonces for this reason, if inlining is absolutely necessary:

CSP: script-src ‘self’ ‘nonce-random-number’ ‘unsafe-inline’;

<p>Hello, <script>alert(“Bad code.”);</script></p>

<script nonce=”random-number”>alert(“Good code.”);</script>

The attacker shouldn’t be able to know the nonce beforehand, so this is better

Problems: Whitelists

Whilelists have several problems:

- Only as secure as the target. Many don't know or care about CSP.
- Wildcards just mean more targets could potentially be used to exploit XSS on your site. (DNS Poisoning?)

Scenarios:

- Load jQuery via CDN whitelist that also hosts AngularJS. Attacker injects AngularJS, and can run arbitrary code.
- Inject arbitrary code via JSONP callback in vulnerable CDN code.

Problems: Policy Maintenance

Large web applications usually have many dependencies. This means the security policies will be that much more complex. Lots of refactoring is usually needed to even begin to implement CSP policies, otherwise the app will break.

One solution proposed by Google is ‘**strict-dynamic**’ which allows whitelisting a trusted root script which can load, and propagate, this trust to subsequently loaded scripts.

Obviously a vulnerability in the root script means XSS and arbitrary execution.

Problems: Policy Maintenance

Google also just released (this year) two Chrome extensions (CSP Evaluator, CSP Mitigator) to help implement and test policies.

https://security.googleblog.com/2016/09/reshaping-web-defenses-with-strict.html

These should also help developers with implementing and maintaining CSP.

Problems: CSP2 Browser Support

CSP is only good when the browser fully supports it.

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			49						
			51						
			52					4.4	
			53	9.1		9.3		4.4.4	
11	14	[7] 49	54	10	41	10	all	53	53
		[7] 50	55	TP	42				
		[7] 51	56		43				
		[7] 52	57						

Problems: Adoption

If nobody uses it, it can’t help. (One paper cited as low as 39%)

Also, there is less of a reason for companies other than Google to devote the resources to support CSP in their browsers (Looking at you, Microsoft).

Only with time … as the specification matures and more tooling is developed to make using CSP less painful will this issue go away.

Conclusion

CSP can’t (not intended to) fix everything

At best, a backup tool to help mitigate issues that arise

Remember: If an attacker can modify the web app files, they can modify the CSP.

Security is Hard :(

Conclusion: Further reading

A long list of potential XSS attack vectors (don't have nightmares!):

https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Thank You!

Questions?

References

[1] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1365–1375. ACM, 2016.
[2] WorldWide Web Consortium. Content security policy level 3, https://www.w3.org/TR/CSP/, 2016.
[3] Google. Reshaping web defenses with strict content security policy, https://security.googleblog.com/2016/09/reshaping-web-defenses-with-strict.html, 2016.
[4] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In Proceedings of the 2006 ACM symposium on Applied computing, pages 330–337. ACM, 2006.
[5] Hao Chen Matthew Van Gundy. Noncespaces: Using randomization to defeat cross-site scripting attacks. El Sevier.
[6] Kevin Spett. Cross-site scripting. SPI Labs, 1:1–20, 2005.
[7] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In Proceedings of the 19th international conference on World wide web, pages 921–930. ACM, 2010.
[8] Michael Hicks Trevor Jim, Nikhil Swamy. Defeating script injection attacks with browser-enforced embedded policies (beep). ACM.
[9] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1376–1387. ACM, 2016