Week 7, Lecture 2

COMP6[48]43

Hamish Cox

Apr 1st, 2025

Lecture 1 Recap

- JS/HTML injection
  - Usually known as Cross-Site Scripting for reasons™.
  - We’ll talk about ‘cross-site’ capabilities next week.
- Stored and Reflected XSS
- DOM-based XSS (extended)

Lecture 2 Overview

- The Basics of Mitigating JS/HTML Injection
- What is special about injecting JS?
- Defense in Depth
- CSP
- How can developers mess this up?

Basic JS/HTML Injection Mitigation

Sanitise your user inputs/templating :)

Demo…

JS injection is ✨ special ✨

Why?

Because it allows us to execute code as a user and on their machine.

We saw this yesterday with being able to fetch a user’s cookie using JS and sending them to a server we control. We could also perform an action as them (e.g. change their password).

vs Server-Side

For server-side vulnerabilities (SQLi, SSTI, Command Injection, etc), the responsibility ends with the developers (of the application, libraries, etc). If they mess up, their server/application is vulnerable.

With JS injection, the code is executing in the browser, and if the developer fails to protect against it, it allows untrusted code to execute on a user’s device. As a result, the browser can take steps to:

- detect and prevent the execution of untrusted JS (tonight).
- mitigate the effects of executing/rendering untrusted JS and HTML (next week, alongside the ‘cross-site’ stuff).

Defense in Depth

Ok, let’s assume that an application has an XSS vulnerability that can affect logged in users.

How could we detect malicious JavaScript/HTML?

Content Security Policy

What’s the simplest way to add a security feature to HTTP/browsers?

We’ve done it before with cookies…

Just add a header.

Content Security Policy

By setting the `Content-Security-Policy` header, we can tell the browser exactly which scripts we intend to execute.

The header allows you to specify the allowed sources for scripts (in order of most to least secure, according to me):

- `'none'`
- `'sha256-<hash>'`
- `'nonce-<nonce>'`
- `https://example.com/some-script.js`
- `https://example.com/some-directory/`
- `https://example.com/`
- `'self'`

Content Security Policy

There are also some more niche values to modify how the CSP is applied:

- `'unsafe-eval'` - prevents dynamic code execution (e.g. `eval`, `Function`, calling `setTimeout` with a string, etc.)
- `'unsafe-hashes'` - allows hash expressions to allow inline event handlers to execute.
- `'strict-dynamic'` - tells the browser to ignore allowlist expressions (i.e. domains/URLs, `'self'`) and only use nonce/hash expressions, but to also trust scripts loaded by trusted scripts.

Content Security Policy

There's also some that really defeat the point:

- `https:` or `http:`
- `'unsafe-inline'`

Content Security Policy

Suppose we want to restrict our website to only execute one script: `/some-script.js`. We would set the CSP to:

```
script-src https://exameple.com/some-script.js
```

Content Security Policy

Why do we need to specify `script-src`?

I know one of you lot will put your hand up - let me ask my rhetorical questions in peace.

CSP isn't just used for scripts - it is named Content Security Policy, after all.

Content Security Policy

You can individually specify rules for:

- Script elements and event handlers separately (`script-elem-src`, `script-attr-src`)
- Fetch/XHR/WebSocket connections (`connect-src`)
- iframes and web/service workers (`worker-src`, `frame-src`, `child-src`)
- Images (`img-src`)
- Styles (`style-src`, `style-elem-src`, `style-attr-src`)
- Fonts (`font-src`)
- About 5 other niche kinds of resource I cbf listing
- All of these together using `default-src` as a fallback

Then theres another 10 directives that restrict other APIs/features of the DOM.

Some Examples

Suppose we have a `/post/<id>` page that renders a post's content as HTML, and we don't protect against XSS:

```
HTTP/1.1 200 OK
Content-Type: text/html
...

<html>
  <!-- ... -->
  cool post!
  <script>alert(1)</script>
  <!-- ... -->
</html>
```

The browser will execute the malicious script.

Content Security Policy

What if we don’t have any JS on this page, so all JS is untrusted?

```
HTTP/1.1 200 OK
Content-Type: text/html
…
Content-Security-Policy: script-src 'none'

<html>
  <!-- ... -->
  cool post!
  <script>alert(1)</script>
  <!-- ... -->
</html>
```

The browser will not execute the malicious script.

But it is worth noting that it will always be invisible in these examples.

Content Security Policy

What if we want to load a script to add some interactivity?

```
HTTP/1.1 200 OK
Content-Type: text/html
…
Content-Security-Policy: script-src 'self'

<html>
  <script src="/some-script.js"></script>
  <!-- ... -->
  cool post!
  <script>alert(1)</script>
  <!-- ... -->
</html>
```

The browser will not execute the malicious script.

Content Security Policy

We could also use a hash:

```
HTTP/1.1 200 OK
Content-Type: text/html
...
Content-Security-Policy: script-src 'sha256-abcdef123456'

<html>
  <script src="/some-script.js" integrity="sha256-abcdef123456"></script>
  <!-- ... -->
  cool post!
  <script>alert(1)</script>
  <!-- ... -->
</html>
```

The browser will not execute the malicious script.

Content Security Policy

Or a nonce:

```
HTTP/1.1 200 OK
Content-Type: text/html
…
Content-Security-Policy: script-src 'nonce-randomvalue'

<html>
  <script src="/some-script.js" nonce="randomvalue"></script>
  <!-- … -->
  cool post!
  <script>alert(1)</script>
  <!-- … -->
</html>
```

The browser will not execute the malicious script.

Content Security Policy

What if we want to load a script from another website (perhaps a CDN)?

```
HTTP/1.1 200 OK
Content-Type: text/html
...
Content-Security-Policy: script-src https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js

<html>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>
  <!-- ... -->
  cool post!
  <script>alert(1)</script>
  <!-- ... -->
</html>
```

The browser will not execute the malicious script.

Content Security Policy

We could also be a bit less specific if we wanted/needed:

```
HTTP/1.1 200 OK
Content-Type: text/html
…
Content-Security-Policy: script-src https://cdn.jsdelivr.net/

<html>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></
script>
  <!-- ... -->
  cool post!
  <script>alert(1)</script>
  <!-- ... -->
</html>
```

The browser will not execute the malicious script.

Content Security Policy

But if the allowlisted site allows for arbitrary code uploads, an attacker could just upload their own malicious scripts (or find one they can misuse)…

```
HTTP/1.1 200 OK
Content-Type: text/html
…
Content-Security-Policy: script-src https://cdn.jsdelivr.net/

<html>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>
  <!-- ... -->
  cool post!
  <script src="https://cdn.jsdelivr.net/npm/my-malicious-package/dist/index.js"</script>
  <!-- ... -->
</html>
```

The browser will execute the malicious script.

Content Security Policy

We could also use nonces or hashes as shown previously:

```
HTTP/1.1 200 OK
Content-Type: text/html
...
Content-Security-Policy: script-src 'nonce-randomvalue'

<html>
  <script
    nonce="randomvalue"
    src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"
  ></script>
  <!-- ... -->
  cool post!
  <script>alert(1)</script>
  <!-- ... -->
</html>
```

The browser will not execute the malicious script.

Content Security Policy

What if we want to run an inline script? We can just use a hash...

```
HTTP/1.1 200 OK
Content-Type: text/html
...
Content-Security-Policy: script-src 'sha256-abcdef123456'

<html>
  <script>/* do things */</script>
  <!-- ... -->
  cool post!
  <script>alert(1)</script>
  <!-- ... -->
</html>
```

The browser will not execute the malicious script.

Content Security Policy

We can also use a nonce:

```
HTTP/1.1 200 OK
Content-Type: text/html
…
Content-Security-Policy: script-src 'nonce-randomvalue'

<html>
  <script nonce="randomvalue">/* do things */</script>
  <!-- ... -->
  cool post!
  <script>alert(1)</script>
  <!-- ... -->
</html>
```

The browser will not execute the malicious script.

Content Security Policy

Or there is `'unsafe-inline'` (bad idea):

```
HTTP/1.1 200 OK
Content-Type: text/html
…
Content-Security-Policy: script-src 'unsafe-inline'

<html>
  <script>/* do things */</script>
  <!-- ... -->
  cool post!
  <script>alert(1)</script>
  <!-- ... -->
</html>
```

The browser will execute the malicious script.

Can we bypass this?

It depends!

Bypass Ideas

- Guessable nonces.
- Being able to upload to an allowlisted site, including `'self'`.
- Using `<base>` to change the base URI of the page to bypass `'self'`.
- Somehow injecting `\r\n\r\n` into a response header so that some of the headers (potentially the CSP!) are pushed into the response body.
- Injecting markup to replace `<meta>` tag containing the CSP.
- Vulnerabilities in either libraries used legitimately on the page, or in libraries permitted by the CSP.

We’ll go over these ideas but since I wasn’t sure where we’d get up to I haven’t written slides, so it’s messy demo time!