Sensible Defaults for Client Side Security

Jen Zajac @jenofdoom

catalyst
open source technologists

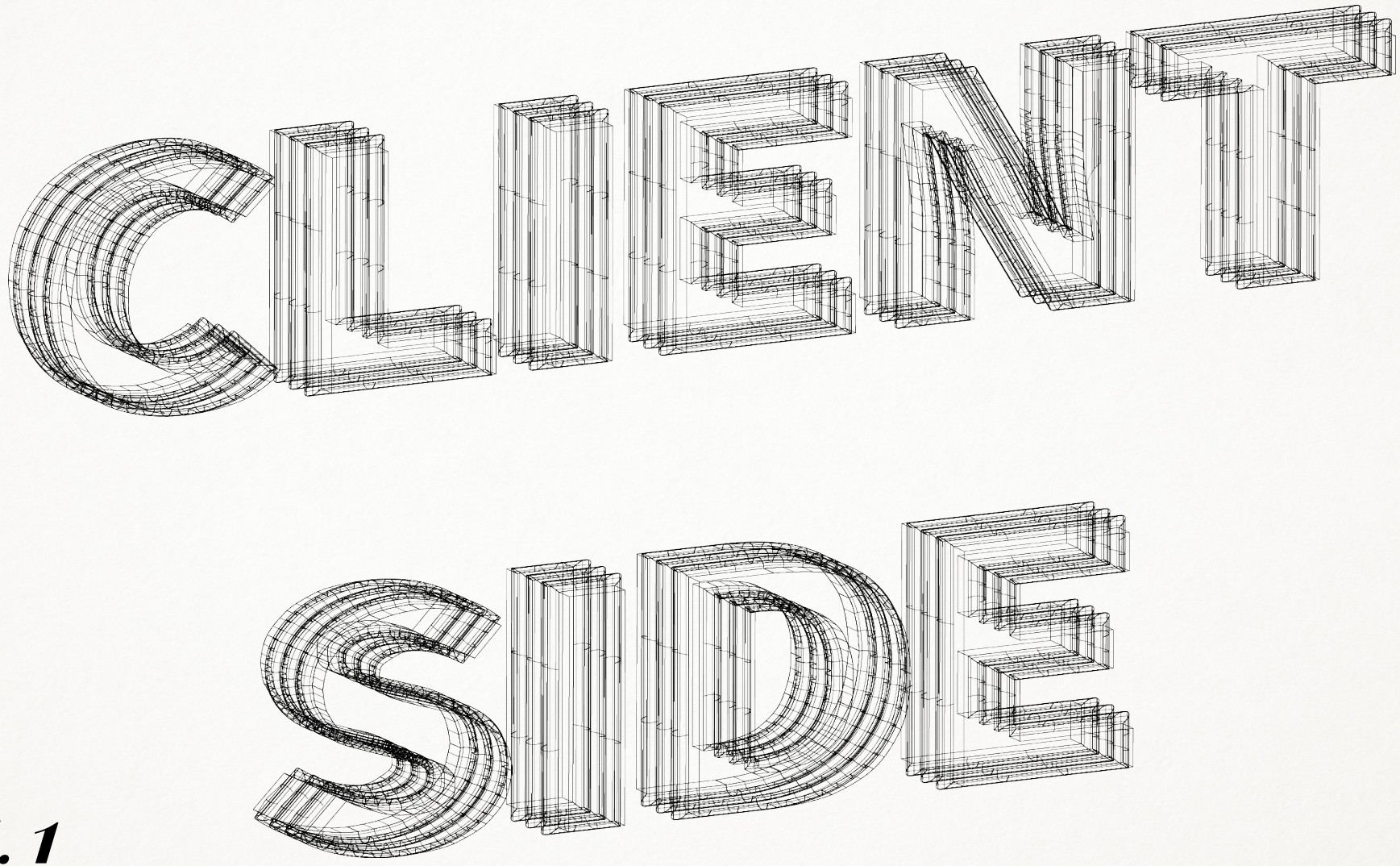

FIG. 1

The original
“*someone else’s computer*”

HTML, CSS, images fonts, and JavaScript

If we insist that our users need JavaScript, we *must* make it secure.

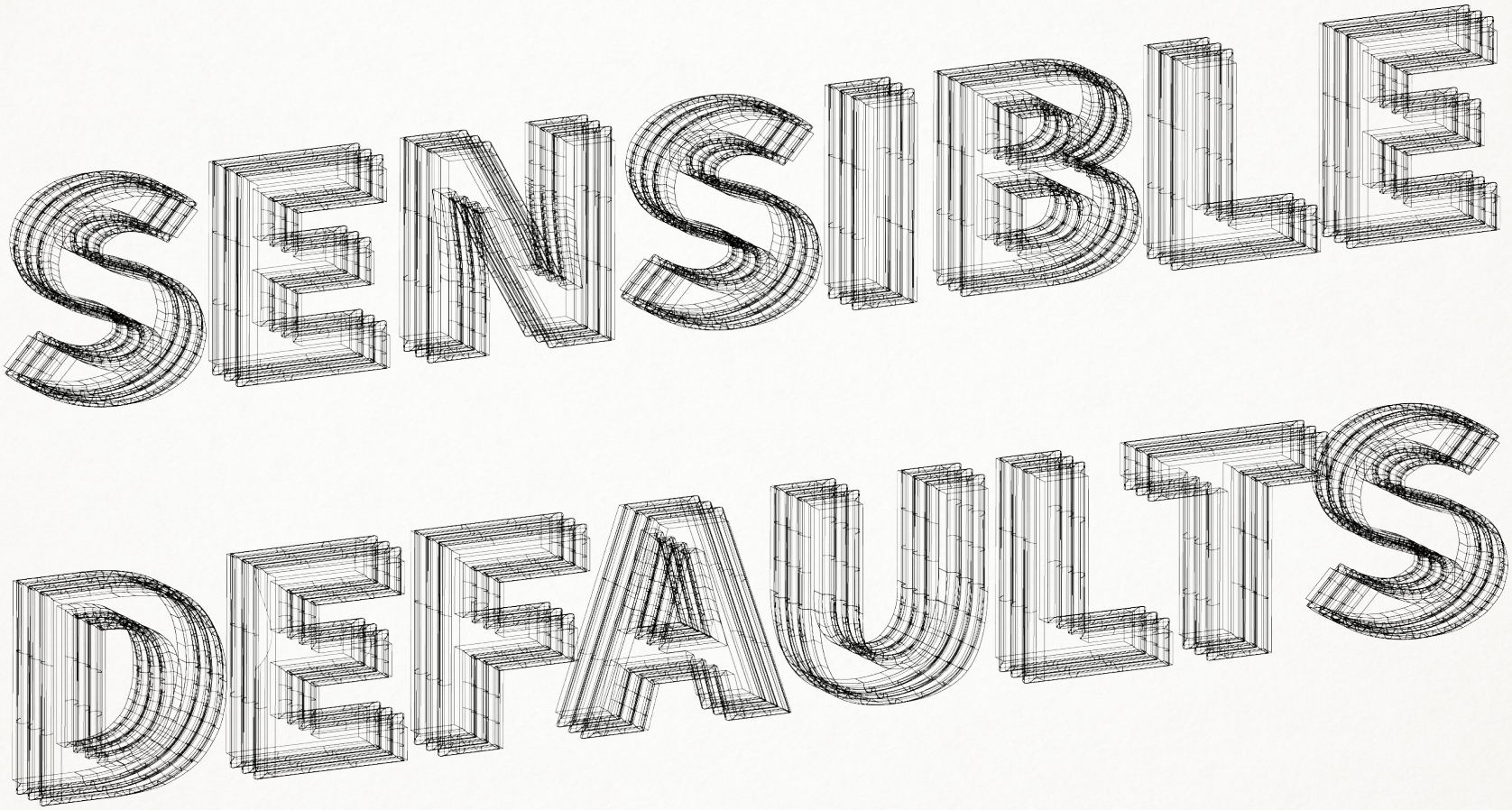

FIG. 2

What is a default?

Default values are standard values that... make [our project] as accessible as possible... without necessitating a lengthy configuration process

– Somebody on Wikipedia
https://en.wikipedia.org/wiki/Default_(computer_science)

Why “sensible” defaults?

Frivioulous defaults wouldn’t make much sense ;)

Reduce the congnitive overload of descision making

Set our projects up to be secure by convention

... even if those choices are both clear and useful, the act of deciding is a cognitive drain. And not just while [we]'re deciding... even after we choose, an unconscious cognitive background thread is slowly consuming/leaking resources, "Was that the right choice?"

– Kathy Sierra

http://seriouspony.com/blog/2013/7/24/your-app-makes-me-fat

Beyond configuration

“Zero configuration” trend in the JavaScript world

We’re not there yet for security…

Is there such a thing as a style guide for security?

Make your own checklist

1. Dependency management
2. Content Security Policy
3. Storing data locally
4. Escaping

FIG. 3

If we're standing on the shoulders of giants, we'd better measure them

How do we measure?

Use a package manager like **npm** so we have a manifest of dependencies and versions

Use metrics like how popular the library is and how often it receives updates
https://npmcompare.com/

Finding the right balance

We *don't* want our app to break from constant updates

We *do* want to get critical security updates

Right now this requires decision making squishy humans

Assisting the squishy humans with tooling

Automated regression tests so you can update with more confidence

Tools to check for outdated packages and known security vulns

npm-update-checker

```
Using /home/jenz/Code/js-build-pipelines-training/webpack-example/package.json
[..................] - :
 babel-preset-env     ~1.3.3  →   ~1.4.0
 babel-preset-react  ~6.23.0  →  ~6.24.1
 css-loader          ~0.27.3  →  ~0.28.0
 eslint              ~3.18.0  →  ~3.19.0
 file-loader         ~0.10.1  →  ~0.11.1
 react               ~15.4.2  →  ~15.5.4
 react-dom           ~15.4.2  →  ~15.5.4
 react-router-dom     ~4.0.0  →   ~4.1.1
 webpack              ~2.3.3  →   ~2.4.1

Run ncu with -u to upgrade package.json
```

Retire.js

https://www.npmjs.com/package/retire

https://nodesecurity.io/advisories/

Alternative tools:

1. nsp (free CLI tool)
2. snyk (commercial only)

Enjoy the firehose of everything being awful

Once you've investigated a problem that a dependency has:

1. Update to newer fixed version
2. Or, avoid that dep altogether
3. Or, document an exception in `.retireignore.json` if not relevant

Set up for easy use

```
npm install --save-dev retire npm-check-updates
```

```
"scripts": {

  "deps:check": "./node_modules/.bin/npm-check-
updates && ./node_modules/.bin/retire",

  "deps:update": "./node_modules/.bin/npm-check-
updates -u && npm install &&
./node_modules/.bin/retire",
```

Verify CDN resources

With subresource integrity:

```
<script
src="https://code.jquery.com/jquery-
3.2.1.min.js" integrity="sha256-
hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNg
t4=" crossorigin="anonymous"></script>
```

Make your own: **https://www.srihash.org/**

FIG. 4

uh, isn’t that server-side?

yes, but

it *strongly affects* the client side

so f/e devs should both know and care about it

(it's also your *last line* of protection against XSS)

Set up your CSP policy

Test during dev too – you don't want a rude awakening later when stuff doesn't work

Lock it down really hard to start with, and open it up later if it becomes absolutely necessary

In your server config

Apache

```
Header set Content-Security-Policy
"<stuff in here>"
```

nginx

```
add_header Content-Security-Policy
"<stuff in here>";
```

Suggested starter rules

```
default-src 'none';
script-src 'https://yourdomain';
connect-src 'https://yourdomain';
img-src 'https://yourdomain' data:;
style-src 'https://yourdomain';
font-src 'https://yourdomain';
object-src 'https://yourdomain';
```

CSP continued

IE doesn’t support it (properly) but Edge does

You can send reports of fails to a log

You can do legacy inline JS via nonces

Version 3 of the standard gives us some new fancy features

CSP resources

Reference guide:
https://content-security-policy.com

Chrome plugin for testing new policies against existing sites:
https://goo.gl/Nkghf5

Tool to evaluate your policy:
https://csp-evaluator.withgoogle.com

FIG. 5

User Sessions

Session cookies or JWT?

Session cookies

This form of auth has been around for ages.

It’s not bulletproof but its drawbacks are well understood.

Most frameworks implement this well.

Hands off the cookies

Assuming you have a backend handling auth, the client side has no need to access your sesssion cookies.

Make them `HttpOnly`

Secure *all* your cookies

Assuming your site is https (it is, right?) then you should always set the `Secure` flag to `true`

Consider setting `hostOnly` and `domain` as appropriate too, although these can be spoofed...

Better CSRF protection

There is a new flag that can more reliably (where supported) ensure cookies don't get sent along with malicious requests:

`SameSite`

https://scotthelme.co.uk/csrf-is-dead/

JSON Web Tokens (JWT)

Recent-ish standard for representing a claim between two or more systems.

https://jwt.io/introduction

Although many of the early critical failures in JWT are fixed, it's still new and not as well understood

JWTs *are* good as single use tokens

https://goo.gl/qcJfDa

It's a standard *not* a storage mechanism

Still need to choose a storage method:

- localStorage
- sessionStorage
- as a cookie

JWT in localStorage

Token would persist until deleted (although hopefully you'd give it an expiry!)

You **must** make sure you use HTTPS

Only avaiable via same domain, but readable via JS === vulnerable to XSS

JWT in sessionStorage

This is the same as localStorage, except it would be cleared on the user closing the window

Still vulnerable to XSS

JWT in a cookie

If you make the cookie HttpOnly, much less vulnerable to XSS

You are vulnerable to cross domain attacks but these can be mitigated via CSRF token use

Much smaller storage area though

Stick with user sessions unless you have a specific need for JWT

Other User Data?

Things to consider:

Store locally, or on the server? Set it to expire or persist?

If locally, store in:

- localStorage?
- sessionStorage?
- in a cookie?

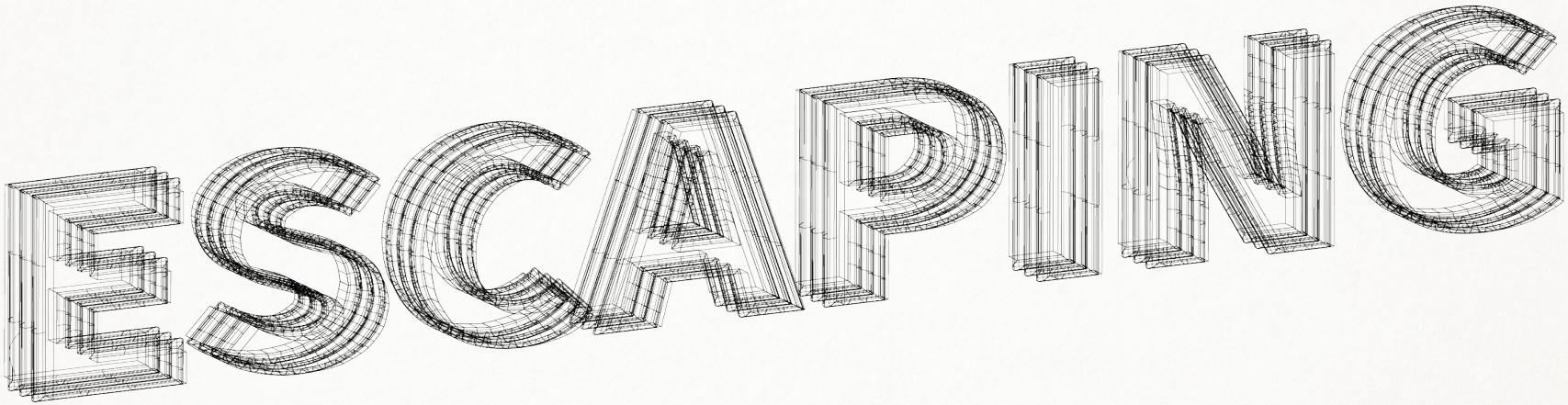

FIG. 6

Don’t roll your own

```
  // Regular Expressions for parsing tags and attributes
  var SURROGATE_PAIR_REGEXP = /[\uD800-\uDBFF][\uDC00-\uDFFF]/g,
    // Match everything outside of normal chars and " (quote character)
    NON_ALPHANUMERIC_REGEXP = /([^#-~ |!])/g;


  // Good source of info about elements and attributes
  // http://dev.w3.org/html5/spec/Overview.html#semantics
  // http://simon.html5.org/html-elements

  // Safe Void Elements - HTML5
  // http://dev.w3.org/html5/spec/Overview.html#void-elements
  var voidElements = toMap('area,br,col,hr,img,wbr');

  // Elements that you can, intentionally, leave open (and which close themselves)
  // http://dev.w3.org/html5/spec/Overview.html#optional-tags
  var optionalEndTagBlockElements = toMap('colgroup,dd,dt,li,p,tbody,td,tfoot,th,thead,tr'),
      optionalEndTagInlineElements = toMap('rp,rt'),
      optionalEndTagElements = extend({},
                                              optionalEndTagInlineElements,
                                              optionalEndTagBlockElements);

  // Safe Block Elements - HTML5
  var blockElements = extend({}, optionalEndTagBlockElements, toMap('address,article,' +

'aside,blockquote,caption,center,del,dir,div,dl,figure,figcaption,footer,h1,h2,h3,h4,h5,' +
```

Use your framework

For example, React escapes anything safely in a normal template `{variable}`.

But if you're using `dangerouslySetInnerHTML={}` then **the clue is in the name** that you need to pay more attention!

Make sure you're escaped for the right context

Escaped for HTML `!=` escaped for a HTML tag attribute

https://goo.gl/Zd1HQV

FIG. 7

_blank tabnapping

Do you link off to an external site, opening it in a new tab by setting `target="_blank"` ?

https://goo.gl/nu30RK

Some (not all) mitigations

```
<a
 href="http://site.com"
 target="_blank"
 rel="noopener noreferrer"
>
  External link
</a>
```

Code obfuscation

Don’t bother – serious attackers can overcome this easily

Avoid troublesome stuff

iframes

flash

java applets

Stay informed!

Keep reading, keep learning

Work with backend devs, ops people and testers to create applications that are secure across the whole attack surface

Thanks

@jenofdoom