Preventing XSS and CSRF

Jeremy Stashewsky, GoInstant

@jstash

trust

Trust

Security ∈ Trust

Vulnerabilties $\notin$ Trust

Prevention > Repair

This talk is an introduction to two common web vulnerabilities

XSS (Cross Site Scripting)

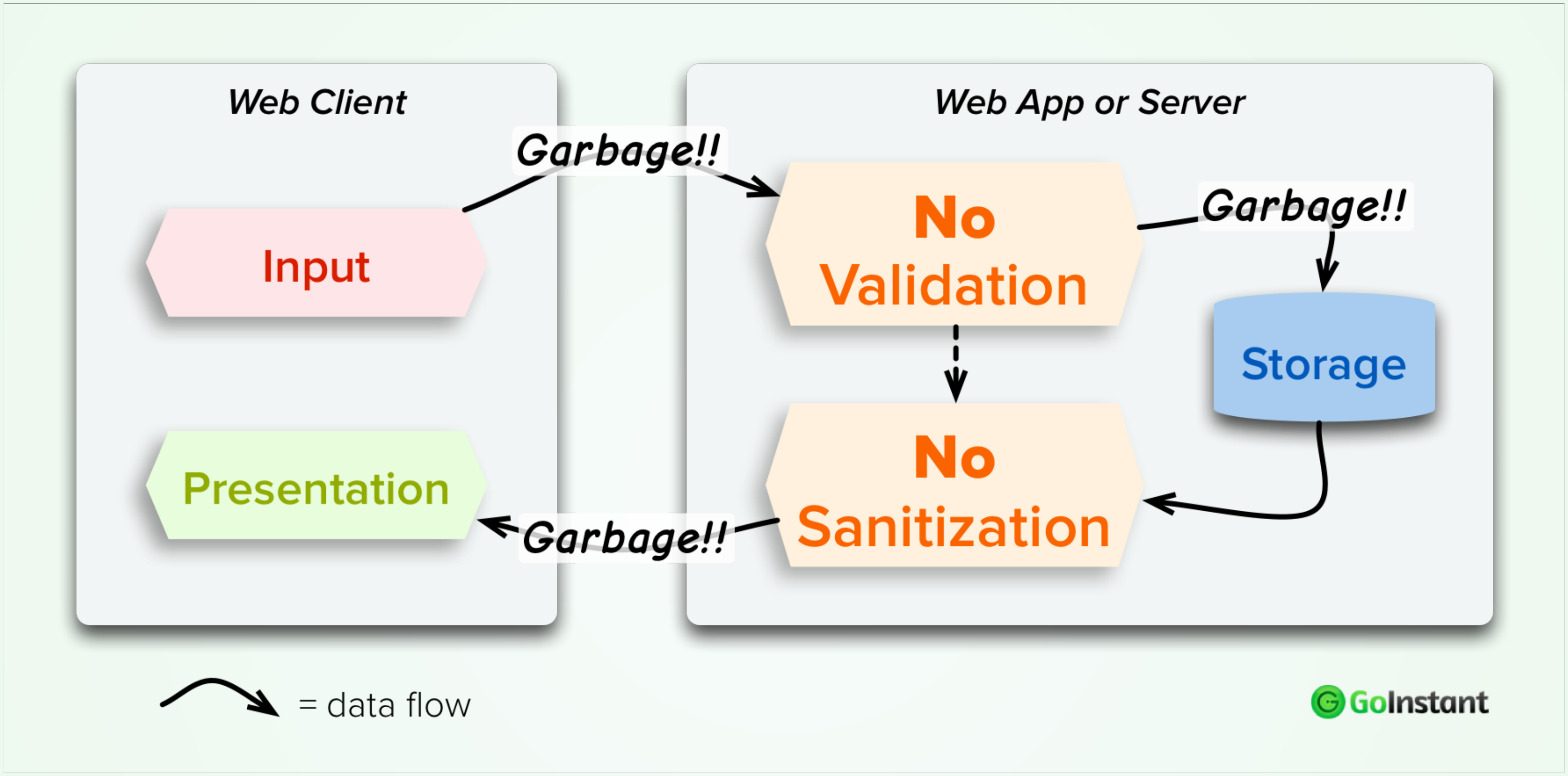
CSRF (Cross Site Request Forgery)

And how to prevent (or fix) them.

XSS

Cross Site Scripting

XSS is an injection attack, driven by user-controlled inputs



Potentially, a user can place arbitrary

HTML

and/or

JavaScript

on to your page!

An example

```
<h1>Hello <%- user.name %>, welcome to <%- site.name %></h1>
```

Where `<%-  %>` is an **Interpolation** operator for a **Template Slot**.

What happens if someone updates my profile and changes my name from

“Jeremy”

to

“</h1><script>window.location='https://evil.com'</script>”?

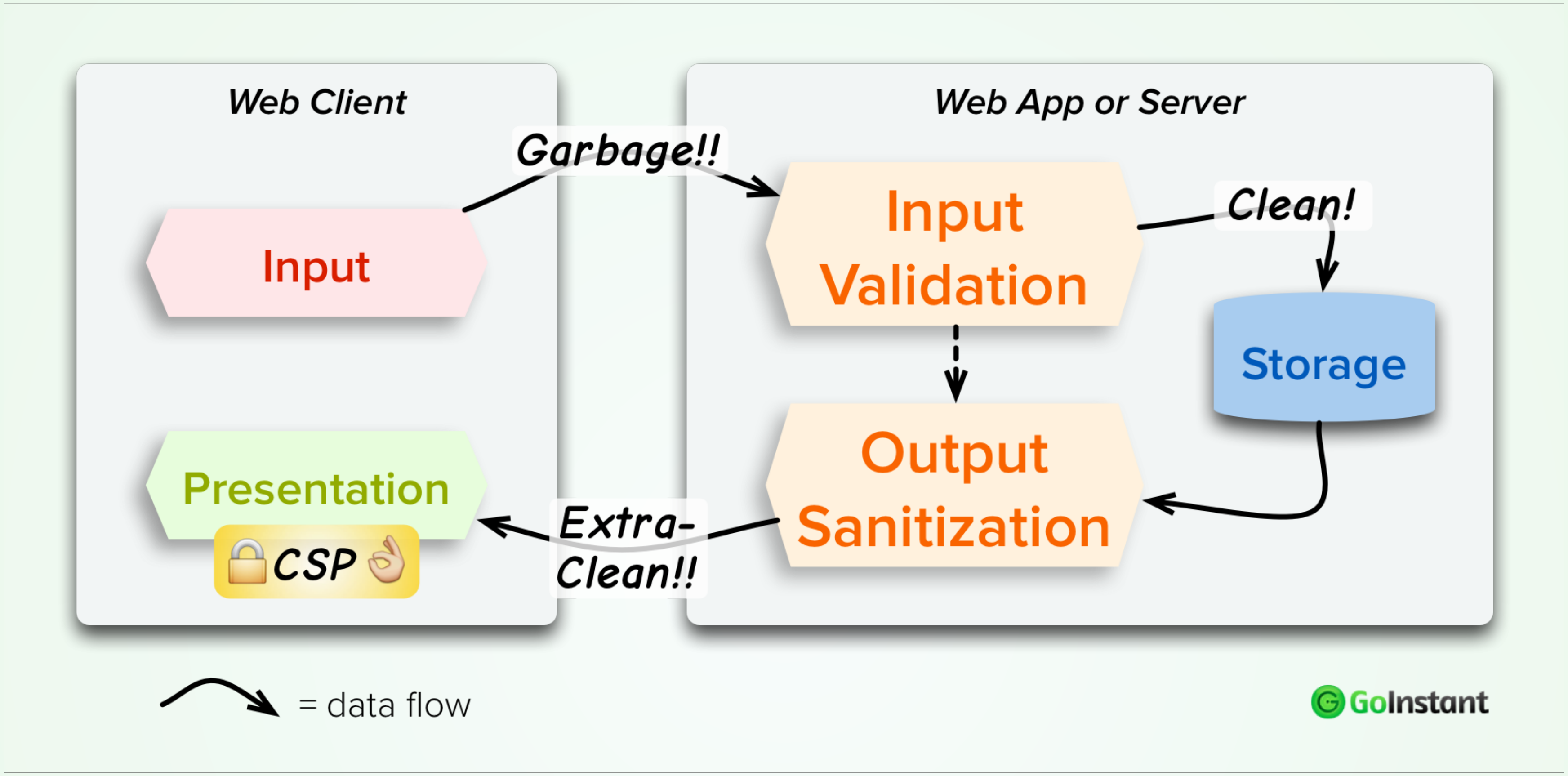
```
<h1>Hello <%- user.name %>, welcome to <%- site.name %></h1>
```

… is rendered as …

```
<h1>Hello </h1>
<script>window.location='https://evil.com'</script>,
welcome to My Awesome Site</h1>
```

A Three-Part Approach to Preventing XSS

1. Validate Input
2. Sanitize Output
3. Enable Content-Security-Policy



Web Client
Web App or Server
Garbage!!
Input
Input Validation
Clean!
Storage
Output Sanitization
Presentation
🔒CSP 👌
Extra-Clean!!
= data flow
GoInstant

Validation

Step 1: Validation

Best case: Compare against an **Allow List** of known-good values

e.g.

```
var HANDEDNESS = ['Lefty','Righty','Ambidexterous','Other'];
```

The Validation Conundrum

Not everything can be Validated against an Allow List

Human names don't fit into a convenient list :(

Instead, you might say "anything but `<>`" to at least exclude HTML tags.

Sanitization

Step 2: Sanitization

(a.k.a. filtering, normalizing, or escaping)

Goal: Prevent user-controlled data from **breaking out** of its context.

Means: Convert *unsafe* markup to *safe* markup.

HTML Entity-encoding takes *markup* characters and turns them into *display* characters.

Minimal list of HTML Entity Encodings

Character	Encoding
`<`	`&lt;`
`>`	`&gt;`
`'`	`&#39;`
`"`	`&quot;` or `&#34;`
&	`&amp;` or `&#38;`

Exhaustive List of HTML Entity Encodings

(Insert all 65536 JavaScript UTF-16 code-points here)

Basically, entity-encode characters **not** in this RegExp set:

```
[\t\n\v\f\r ,\.0-9A-Z_a-z\-\u00A0-\uFFFF]
```

source: secure-filters

Sanitizing the example (EJS)

Change ...

```
<h1>Hello <%- user.name %>, welcome to <%- site.name %></h1>
```

... to ...

```
<h1>Hello <%= user.name %>, welcome to <%- site.name %></h1>
```

Where `<%=  %>` is an **Escaping** operator for a **Template Slot**.

This changes the bad output from...

```
<h1>Hello </h1>
<script>window.location='https://evil.com'</script>,
welcome to My Awesome Site</h1>
```

... to the safe (entity-encoded) ...

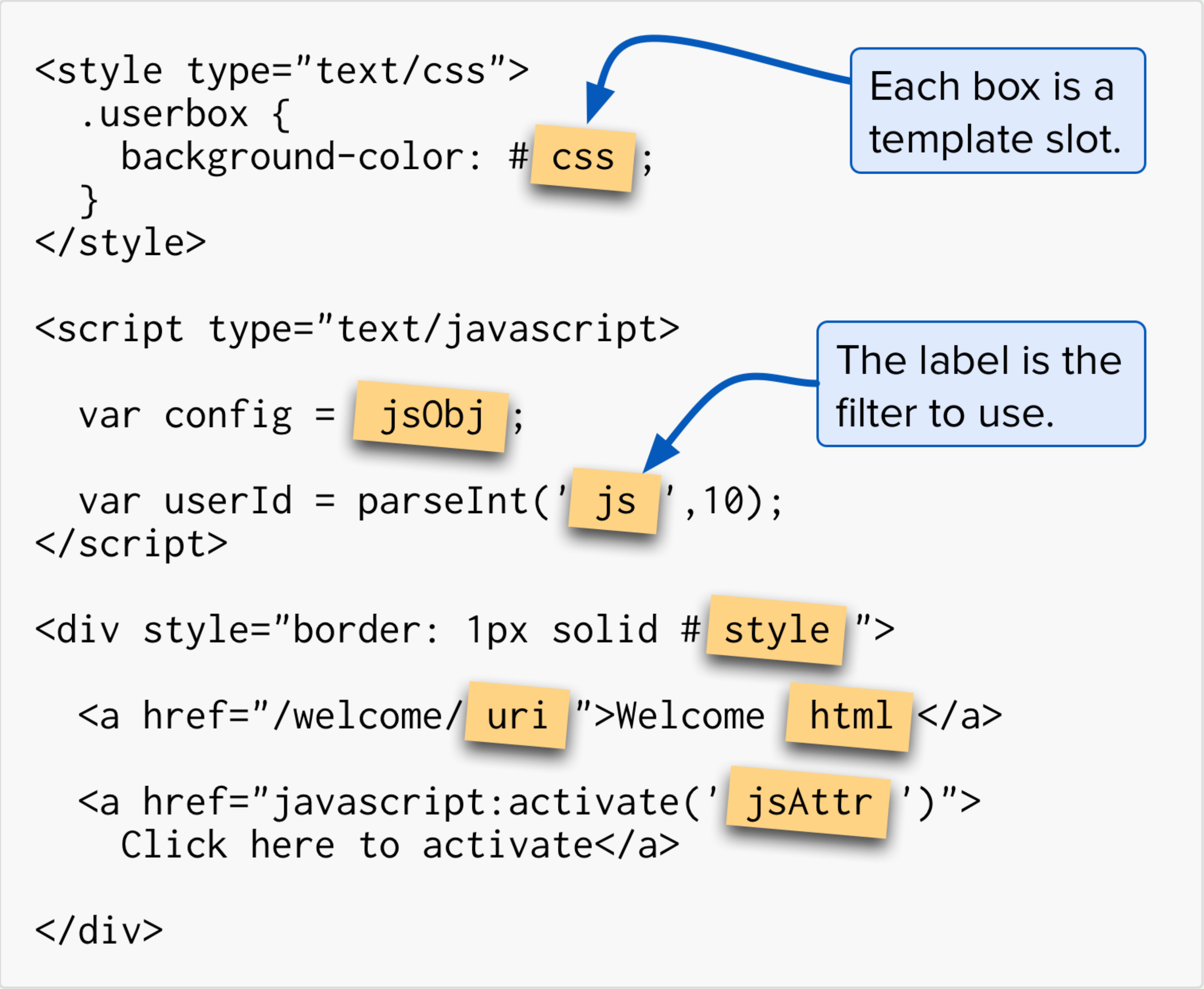
```
<h1>Hello &lt;/h1&gt;
&lt;script&gt;window.location=&#39;https://evil.com&#39;&lt;/script&gt;,
welcome to My Awesome Site</h1>
```

So… I just have to worry about escaping HTML?

No

There's more to it than HTML entity-encoding!

Contextual Filtering



```
<style type="text/css">
  .userbox {
    background-color: # css ;
  }
</style>

<script type="text/javascript>

  var config = jsObj ;

  var userId = parseInt(' js ',10);
</script>

<div style="border: 1px solid # style ">

  <a href="/welcome/ uri ">Welcome html </a>

  <a href="javascript:activate(' jsAttr ')">
    Click here to activate</a>

</div>
```

Each box is a template slot.

The label is the filter to use.

https://github.com/goinstant/secure-filters

JavaScript Variable Attack

```
<script>
  var foo = <%- someJSON %>;
</script>
```

+

```
{ someJSON: JSON.stringify("</script><script>alert('boom');//") }
```

=

```
<script>
  var foo = "</script><script>alert('boom');//";
</script>
```

Sanitizing JavaScript Literals

In strings, things like `<` become `\x3C`, etc.

```
<script>
  var foo = "</script><script>alert('boom');//";
</script>
```

… becomes …

```
<script>
  var foo = "\x3C/script\x3E\x3Cscript\x3Ealert('boom');//";
</script>
```

JavaScript sanitization doesn't save you from `innerHTML`

```
<script>
  var userName = "Jeremy\x3Cscript\x3Ealert('boom')\x3C/script\x3E";
  element.innerHTML = "<span>"+userName+"</span>";
</script>
```

Query Param Attack

```
<a href="/show?user=<%= userId %>">...</a>;
```

+

```
{ userId: "42&user=666" }
```

=

```
<a href="/show?user=42&amp;user=666">...</a>;
```

The server sees `https://example.com/show?user=42&user=666`, so maybe shows user 666 now?

Sanitizing via URI-escaping

Convert unsafe characters to `%XX` UTF-8 octets.

E.g. `&` to `%26`

```
<a href="/show?user=42%26user=666">...</a>;
```

Luckily, `parseInt(”42&user=666”)` evaluates to `42`.

Are there any tools to help me with Sanitization?

Yes!

JavaScript: secure-filters

www.npmjs.org/package/secure-filters

Works in node.js and browsers, includes EJS support

```
<script>
  var config = <%-: config |jsObj%>;
  var userId = parseInt('<%-: userId |js%>',10);
</script>
<a href="/welcome/<%-: userId |uri%>">Welcome <%-: userName |html%></a>
<a href="javascript:activate('<%-: userId |jsAttr%>')">
  Click here to activate</a>
```

Can use these as regular functions too

PHP: Phalcon\Escaper

docs.phalconphp.com/en/latest/reference/escaper.html

Good selection of output filters

```
<title><?php echo $e->escapeHtml($maliciousTitle) ?></title>

<style type="text/css">
.<?php echo $e->escapeCss($className) ?> {
    font-family  : "<?php echo $e->escapeCss($fontName) ?>";
}
</style>

<div class='<?php echo $e->escapeHtmlAttr($className) ?>'>hello</div>

<script>var some = '<?php echo $e->escapeJs($javascriptText) ?>'</script>
```

Angular.js

Strict Contextual Escaping

docs.angularjs.org/api/ng/service/$sce

The `{{ }}` operator and `ng-` attributes are context-aware!

React & JSX

facebook.github.io/react/docs/jsx-in-depth.html

DOM manipulation macros are available without JSX:

```
var link = React.DOM.a({href: 'https://example.com/'}, 'React');
```

Or, conveniently in JSX:

```
var link = <a href="https://example.com/">React</a>;
```

Java: OWASP Enterprise Security API

OWASP wiki: ESAPI

Has APIs for escaping output, as well as input-validation helpers, anti-CSRF and more.

Go `html/template`

golang.org/pkg/html/template/

Based on EcmaScript Harmony "Quasis" (a.k.a. Tagged Template Strings)

```
<a href="/search?q={{.}}">{{.}}</a>
```

... is *compiled* to mean ...

```
<a href="/search?q={{. | urlquery}}">{{. | html}}</a>
```

Should I sanitize inputs?

No!

Why not to Sanitize Input

Sanitizing input *permanently* modifies the data.

Sanitization is fairly cheap and highly cacheable!

Content-Security-Policy

github.com/w3c/webappsec

Step 3: Content-Security-Policy

Validation can't cover *everything*...

... and Sanitization can't catch *all* the cases...

(but you should still do them!)

... we needed something more!

How to CSP

Pages define an Allow-List of what features (and their Origins) are permissible.

Serve as a HTTP header (or use a `<meta>` HTML tag)

```
Content-Security-Policy:
  default-src 'none';
  connect-src ws-and-xhr.example.com;
  font-src https://fonts.googleapis.com;
  frame-src 'self';
  media-src youtube.com, ytimg.com;
  script-src https://example-cdn.com, https://cloudflare.com;
  style-src https://example-cdn.com;
```

Remember this?

```
<h1>Hello </h1>
<script>window.location='https://evil.com'</script>,
welcome to My Awesome Site</h1>
```

It could have been prevented with restricting scripts **sourced** from the same Origin:

```
Content-Security-Policy: script-src 'self'
```

With `script-src 'self'`, **all** unknown script sources are also blocked:

```
<!-- allowed by CSP: -->
<script src="/main.js"></script>
<!-- blocked by CSP: -->
<script src="https://evil.com/attack.js"></script>
```

Consequently, to *allow* inline script blocks, instead of ...

```
Content-Security-Policy: script-src 'self'
```

... we'd need to say ...

```
Content-Security-Policy: script-src 'self', 'unsafe-inline'
```

Are there any tools to help me with CSP?

require(’helmet’)

npmjs.org/package/helmet

Connect middleware that does CSP *and more!*

```
var helmet = require('helmet');
var app = express(); // or connect
app.use(helmet.csp());
app.use(helmet.xframe('deny'));
app.use(helmet.contentTypeOptions());
```

cspbuilder.info

Neat tool using Report-Only mode to dynamically help you form a valid CSP header.

Just be aware it does send a list of all included scripts/fonts/etc to do that analysis

XSS Prevention In Summary

1. **Validate** your inputs
2. **Sanitize** your outputs
3. **Enable CSP** on your web-server

CSRF

Cross-Site Request Forgery

CSRF exploits the fact that you are **logged-in** to some *other* site.

For example,

- You're logged into `https://example.com`
- You accidentally click a link to `http://evil.com`

Say `evil.com` has the following HTML:

```
<title>Welcome to Evil.com</title>
<script src="https://example.com/api/inviteAdmin?email=hacker@evil.com"></s
```

Even though you're *visiting* `evil.com`,

you're *still authenticated* with `example.com`!

How do we fix this?

In Human terms:

Assert that the user **intended** to do this action.

E.g.

The user was on my website ...

... then, they clicked submit on a form ...

... therefore, this isn't a Cross-Site Forgery

In Technical terms:

"The user was on my website …"

=

Put into any Forms a unique, *secret* Anti-CSRF token that's tied to their login-cookie.

"… then they clicked submit on a form …"

=

Actions that change **application state** should be `POST` / `PUT` / `PATCH` / `DELETE`

(consistent with the REST Architectural Style)

Note: that `POST` /etc. on its own is *not enough* to stop CSRF based on XHR!

"... therefore, this isn't a Cross-Site Forgery"

=

Validate the Anti-CSRF token, which since it was a secret, the attacker can't know.

Note: HTTP isn't very good at keeping secrets, so consider the importance of HTTPS.

Fixing example.com

Assume it's running a simple Express 3.x node.js server with EJS templates.

Express Routes

```
app.get('/api/inviteAdmin', handlerFn);
```

... change this to ...

```
app.post('/api/inviteAdmin', handlerFn);
```

Connect Anti-CSRF middleware

www.senchalabs.org/connect/csrf.html

```
app.use(connect.session());
app.use(connect.csrf());
```

… then to access the token …

```
var token = req.csrfToken();
res.render('template', { _csrf: token });
```

Change the EJS template

```
<form method="GET" action="/api/inviteAdmin">
  <input type="email" name="email">
```

... change to use `POST` and consume anti-CSRF token ...

```
<form method="POST" action="/api/inviteAdmin">
  <input type="hidden" name="_csrf" value="<%- _csrf %>">
  <input type="email" name="email">
```

Are there ways to verify intent without a CSRF token?

Some intent verification ideas

Idea: ask the user for confirmation (just make sure the confirmation isn't CSRF-attackable and is server-controlled)

Idea: For *really* sensitive operations, re-prompting for a password is good, especially for long-lived sessions

CSRF summary

You too can prevent ~~forest~~ CSRF fires!

1. **Verify intent**: did the user do this action?
2. **Be a good REST citizen**: Use `POST` / `PUT` /etc. instead of `GET`.
3. **Use Anti-CSRF tokens**: ties together presence on the site and intent.

In Conclusion

Trust

XSS

1. Validate Inputs (or be Radical)
2. Sanitize Outputs
3. Use Content-Security-Policy

CSRF

1. Verify user intent
2. Be a good REST citizen
3. Use Anti-CSRF tokens

Thanks

- Slides are at stash.github.io/empirejs-2014
- Thanks to my employer GoInstant for sponsoring this talk. We make Real-time, Backend-as-a-Service web APIs, and are very serious about Security.