Preventing XS
and CSRF

Jeremy Stashewsky, GoInsta

@jstash

salesforce GoInstant

trust

Trust

Security ∈ Trust

Vulnerabilties $\notin$ Trust

Prevention > Repair

This talk is an introduction to t common web vulnerabilitie

XSS (Cross Site Scripting)

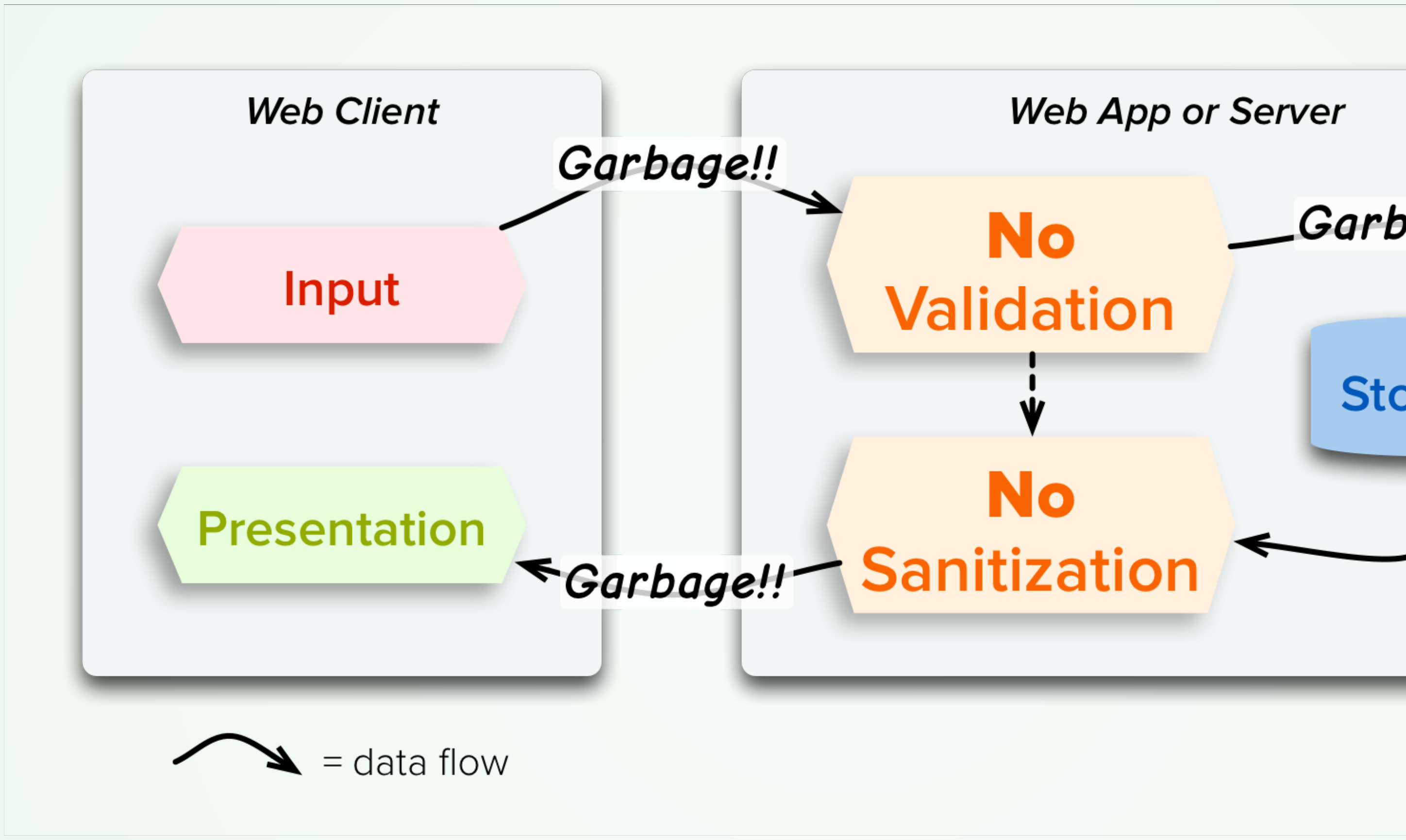
CSRF (Cross Site Request Forgery)

And how to prevent (or fix) them.

XSS

Cross Site Scripting

XSS is an injection attack, driven by user-controlle



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 **Ter**

What happens if someone updates my profile and c
name from

“Jeremy”

to

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

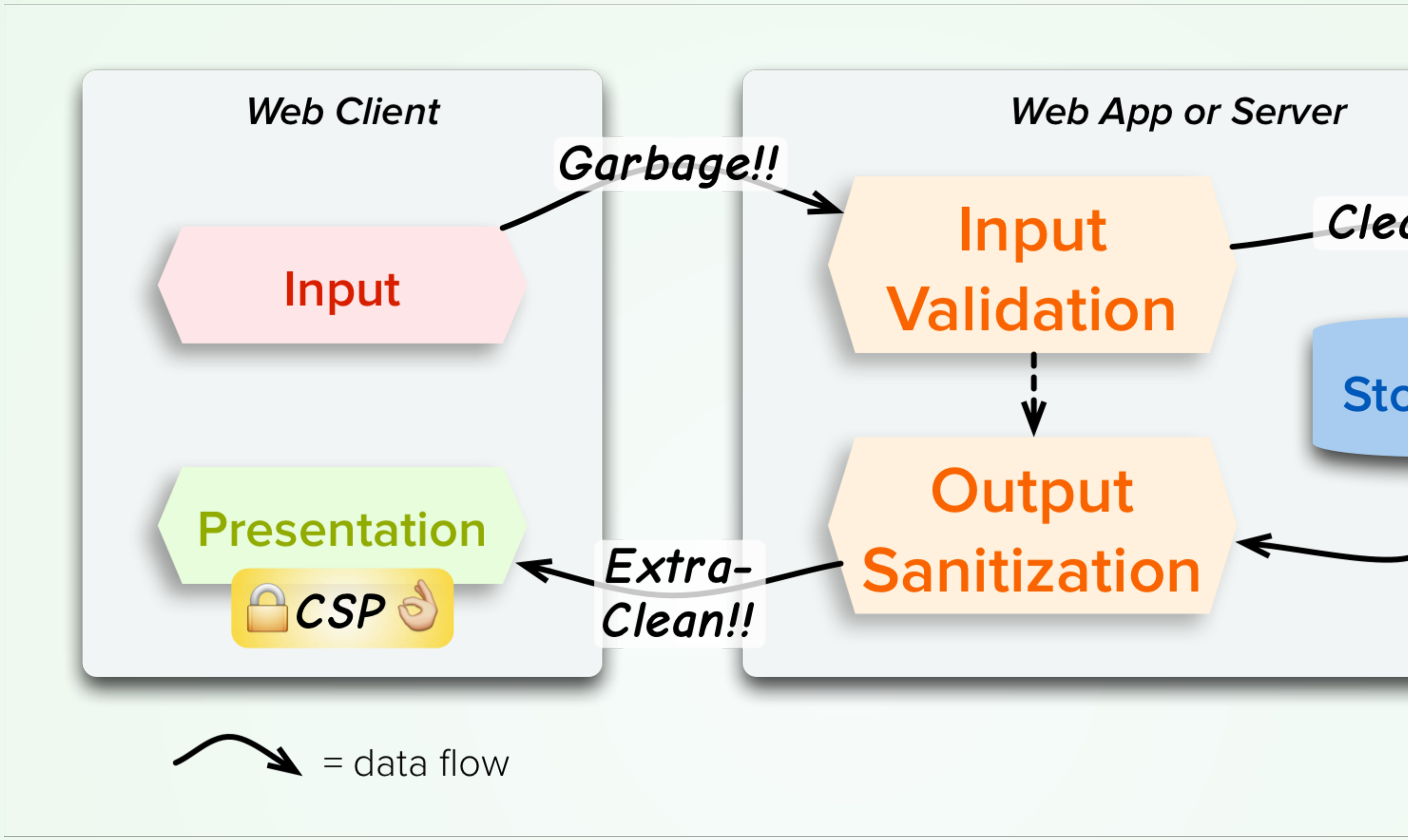
```
<h1>Hello <%- user.name %>, welcome to <%- site.name %></h1>
```

... is rendered as ...

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

A Three-Part Approach Preventing XSS

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


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

Validation

Step 1: Validation

Best case: Compare against an **Allow List** of know
values

e.g.

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

The Validation Conundrum

Not everything can be Validated against an Allo

Human names don't fit into a convenient lis

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

Sanitizatio

Step 2: Sanitization

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

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

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

HTML Entity-encoding takes *markup* characters and
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 Reg

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

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 **Tem**

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;/scr
welcome to My Awesome Site</h1>
```

So... I just have to worry about escaping HTML?

No

There's more to it than HTML entity-encodi

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 i
template s

The label is
filter to use.

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 Literal

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 sa
from `innerHTML`

```
<script>
  var userName = "Jeremy\x3Cscript\x3Ealert('boom')\x3C/script\x
  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/sho`
`user=42&user=666`, so maybe shows user 666

Sanitizing via URI-escaping

Convert unsafe characters to `%XX` UTF-8 oct

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

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

Luckily, `parseInt("42&user=666")` evaluates t

Are there any tools to help me with Sanitization?

Yes!

JavaScript: secure-filte

www.npmjs.org/package/secure-filters

Works in node.js and browsers, includes EJS su

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

Can use these as regular functions too

PHP: Phalcon\Escape

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<

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

Angular.js

Strict Contextual Escaping

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

The `{{ }}` operator and `ng-` attributes are conte

React & JSX

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

DOM manipulation macros are available withou

```
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 Enterpri
Security API

OWASP wiki: ESAPI

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

Go `html/template`

golang.org/pkg/html/template/

Based on EcmaScript Harmony "Quasis" (a.k.a. T
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 da

Sanitization is fairly cheap and highly cachea

Content-Security-Polic

github.com/w3c/webappsec

Step 3: Content-Security-Poli

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 the
are permissible.

Serve as a HTTP header (or use a `<meta>` HTM

```
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 script
from the same Origin:

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

With `script-src 'self'`, **all** unknown script sourc
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, inste

```
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 mo*

```
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 he
a valid CSP header.

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

XSS Prevention In Summar

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 som

For example,

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

Say `evil.com` has the following HTML:

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

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

you're *still authenticated* with `example.co`

How do we fix this?

In Human terms:

Assert that the user **intended** to do this acti

E.g.

The user was on my website ...

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

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

In Technical terms:

"The user was on my website …"

=

Put into any Forms a unique, *secret* Anti-CSRF toker

to their login-cookie.

"… then they clicked submit on a form …"

=

Actions that change **application state** shoul

`POST` / `PUT` / `PATCH` / `DELETE`

(consistent with the REST Architectural Style)

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

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

=

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

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

Fixing example.com

Assume it's running a simple Express 3.x node.js serv
templates.

Express Routes

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

... change this to ...

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

Connect Anti-CSRF middlewa

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 t

```
<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 idea

Idea: ask the user for confirmation (just make su
confirmation isn't CSRF-attackable and is server-co

Idea: For *really* sensitive operations, re-promptir
password is good, especially for long-lived ses

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. inste
3. **Use Anti-CSRF tokens**: ties together presence on
   and intent.

In Conclusio

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 t
  make Real-time, Backend-as-a-Service web APIs, a
  serious about Security.