Developer's Guide to Cross Site Scripting

OWASP APPSEC AU 2017

Felix Shi (@comradepara)

Disclaimer

This is a primer to building web apps that are resilient to Cross Site Scripting (XSS) for devs and QAs.

Please consult your local security team or physician if you think you are suffering from XSS.

whoami

Felix Shi (@comradepara)

- A security guy at Xero
  - Lives across the ditch

- Infosec
- Running
- Cartography

Something something my own opinions does not reflect those of my employer.

Presentation Overview

1. Background

- Fundamentals
  - HTML/Javascript
  - Same Origin Policy
- What is XSS
- Why should you care
- Why is it still an issue
- Exploitation theory

2. Demo / Defense

- Exploitation practice
- Prevention theory
  - Output Encoding
- Prevention practice
  - Backend
  - Frontend
- Defense in Depth
  - Input validation
  - Content Security Policy
  - Cookie Flags

Background

What's in a **modern web application**?

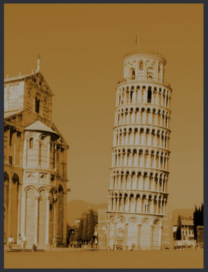
- Stuff the browser uses
  - HTML, Javascript, images, CSS etc.
- Stuff the server uses
  - Ruby, Java, C#, Python etc.
- Persistent server side storage
  - Databases, file systems,
    AWS S3 / Azure Blob, etc.

HTML

- Has been around since 1989
  - Invented by Sir Tim Berners-Lee
- The building block of the web
- Elements on the page are described using tags

HTML Tags

- <b> Hello I'm bold </b>
  - **Hello I'm bold**
- <u> and I'm underlined </u>
  - *And I'm underlined*
- <img src='tower.jpg' />



Common ways to include Javascript on a page

- Inline snippets
  - `<script>console.log("Hello");</script>`
  - `<img src='hi.jpg' onload='alert(1)' />`
- External file
  - `<script src="https://hostname/test.js" />`

What can you do with Javascript?

- Alter the look and functionality of the page
- Access user data associated with the site
- Perform actions on the user's behalf

What can't you do with Javascript?

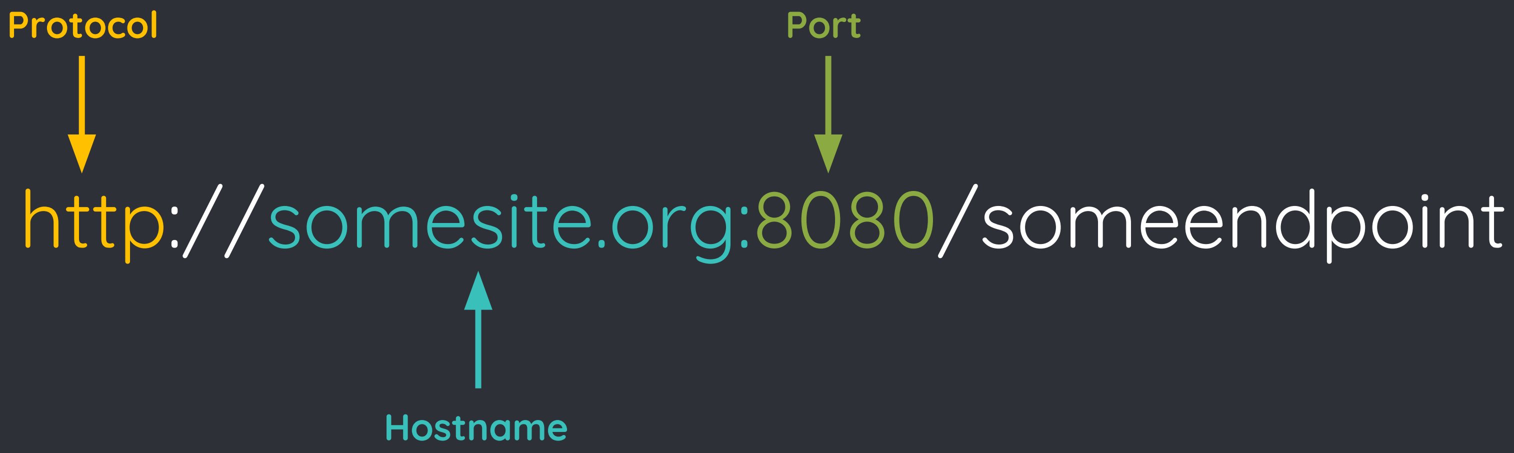
- Unlock the meaning of existence
- End world hunger
- Tell a browser to load data used by youronlinebank.org from evil.org
  - (Unless explicit permissions were given by the former using *CORS*)

What can't you do with Javascript?

- Unlock the meaning of existence
- End world hunger
- Tell a browser to load data used by youronlinebank.org from evil.org
  - (Unless explicit permissions were given by the former using *CORS*)

Same Origin Policy

The browser only allows scripts in one page to access data from another page only if the protocol, hostname and port are exactly the same.



Same Origin Policy

https://myonlinebank.org/statements

↕ Protocol Matches ↕ Hostname does not match

https://evil.org/index.html

Therefore evil.org can't load statements from the online bank site via Javascript

I trust the webapps I use!

Let's talk about...

Cross Site Scripting!

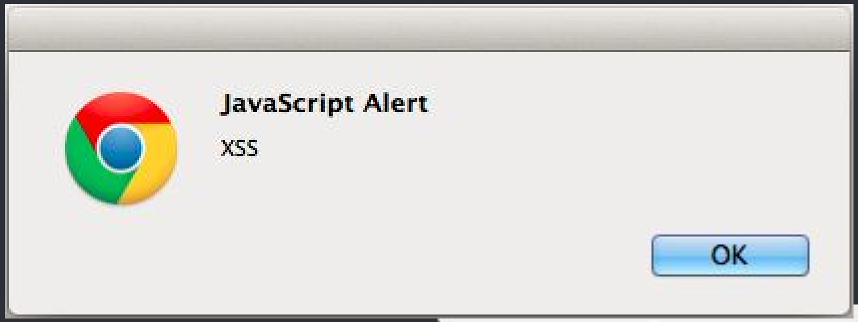
What is Cross Site Scripting (XSS)?

What is Cross Site Scripting (XSS)?

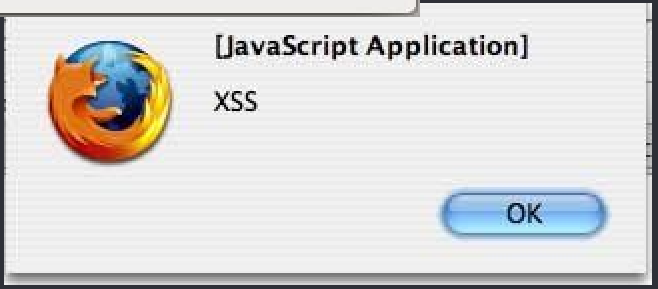
Someone can get their own Javascript to run in the context of your site

What is XSS?

Misconception Annoying dialog boxes start popping up on your site.


JavaScript Alert

XSS


[JavaScript Application]

XSS

Why should I care?

¯_(ツ)_/¯

Who does it affect?

How could it affect the user?

- The user’s browser executes script…

- **Alter** the look and functionality of the page
  - Phishing vector?

- **Access** private user data associated with the site
  - Session Hijacking?

- **Perform actions** on the user’s behalf

¯_(ツ)_/¯

What's the business impact though?

How could it affect your company?

- Loss of **trust**
  - Bad PR
- Fixing technical debt is expensive
  - Which leads to angry product owners
  - Anger leads to hate, something… dark side
- Regulation & Compliance issues
  - Some certs require a clean pentest report

Why is it still an issue?

Why is it still an issue?

Because handling user defined data is **hard**

Exploitation Time!!!

XSS Exploitation Theory

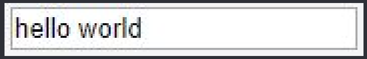
- Identify the entry points of user defined data.
- Identify how the above data gets used in the web app.
- The goal of XSS is to get the browser to execute user defined scripts.

XSS Exploitation Theory

- **Identify the entry points of user defined data.**
- Identify how the above data gets used in the web app.
- The goal of XSS is to get the browser to execute user defined scripts.

Common entry points for user provided data

- Form Input Fields
  - <Input type = "text" Id = "name"… value="hello world">



hello world

- URL Parameters
  - https://example/test_page?name=Felix
  - https://example/test_page?name=<img src=# onerror=alert(1)>

Other entry points for user provided data

- Cookie values
  - User_name = alert('hello');
- User Agents
  - Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like </script><script>alert('')</script>) Chrome/41.0.2228.0 Safari/537.36
- Pretty much anything on the client side can be messed with by the user

For Example...

(interactive demo)

Types of Cross Site Scripting - **Reflected**

Example URL

```
http://trustedsite/search.php?q=<script>alert(1);</script>
```

Page source returned to the victim

```
<html>...<div>
    <script>alert(1);</script>
</div>...</html>
```

Exploitation Vector:

Social Engineering, an attacker crafts a URL and gets people to click on it.

Script Entry Point

- Various places, all ending up in persistent storage.
  - For example: Entries in a **guestbook**

Exploitation Vector

- User just needs to visit page that renders the stored script.

- More dangerous than reflected XSS.
  - Can be prepared in advance
  - Can affect multiple users

Types of Cross Site Scripting - **DOM Based**

Example user data

```
http://trustedsite/search.php?q=<script>alert(1);</script>
```

Page source excerpt

```
...<script>
        document.write(document.URL.indexOf("q=")+2);
</script>..
```

Note that the XSS script **does not** appear in the source code.

Demo Time!

:D

Defence

Prevention Theory

- XSS issues are introduced when user supplied Javascript snippets are executed by the browser
- Faulty handling of user provided data

Defence

- Multiple user defined strings were rendered on the page:
  - The title URL parameter
  - Username field
  - Message field

Defence

URL:

```
http://url/entries?title=<script>alert(1);</script>
```

HTML Output:

```
<h1>
        Thank you for signing my
        <script>alert(1);</script>
</h1>
```

Defence

- ~~Don't allow user input~~
  - Not possible IRL :(

- Ensure that user provided data is validated when appropriate

- Ensure that user provided data is properly encoded/escaped on output

What is Encoding

Defence

HTML Encoding is a technique that converts potentially unsafe characters into their encoded form.

Character	HTML Encoded
<	&lt;
>	&gt;
&	&amp;

Defence - Encoding

Input:

```
<script>
    alert(1);
</script>
```

HTML Encoded Output:

```
&lt;script&gt;
    alert(1);
&lt;/script&gt;
```

Defence - Encoding

Input:

```
<script>
    alert(1);
</script>
```

HTML Encoded Output:

```
&lt;script&gt;
    alert(1);
&lt;/script&gt;
```

User sees:

```
<script>alert(1);</script>
```

Defence - Encoding

Input:

```
<script>
    alert(1);
</script>
```

HTML Encoded Output:

```
&lt;script&gt;
    alert(1);
&lt;/script&gt;
```

NO SCRIPT EXECUTION FOR YOU!!1

User sees:

```
<script>alert(1);</script>
```

HTML Encoding for Developers

Templates: Django, Flask, Rails v. > 3.0, Mustache for Node.JS

- Secure by default
  - Automatically HTML encodes user data

- Opting out on a case by case basis
  - Opting out of HTML Encoding in Flask:

```
{{username | safe}}
```

Defence - Encoding (Frontend)

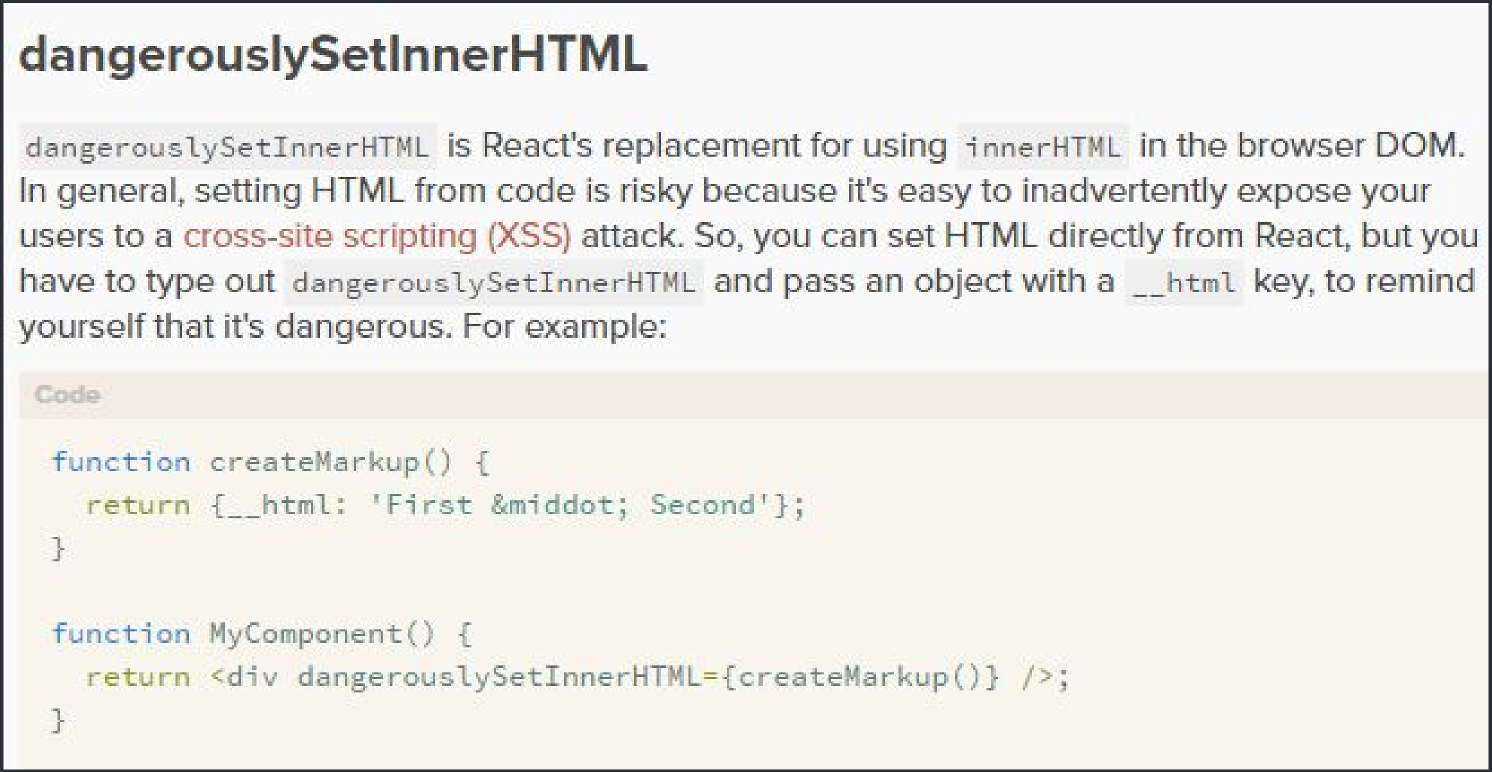
HTML Encoding for Developers

- Most modern front-end Javascript frameworks also HTML encode their output by default.
  - For example: Angular.js, React.js

Opting out of HTML Encoding in React.js…

Defence - Encoding (Frontend)

dangerouslySetInnerHTML



dangerouslySetInnerHTML

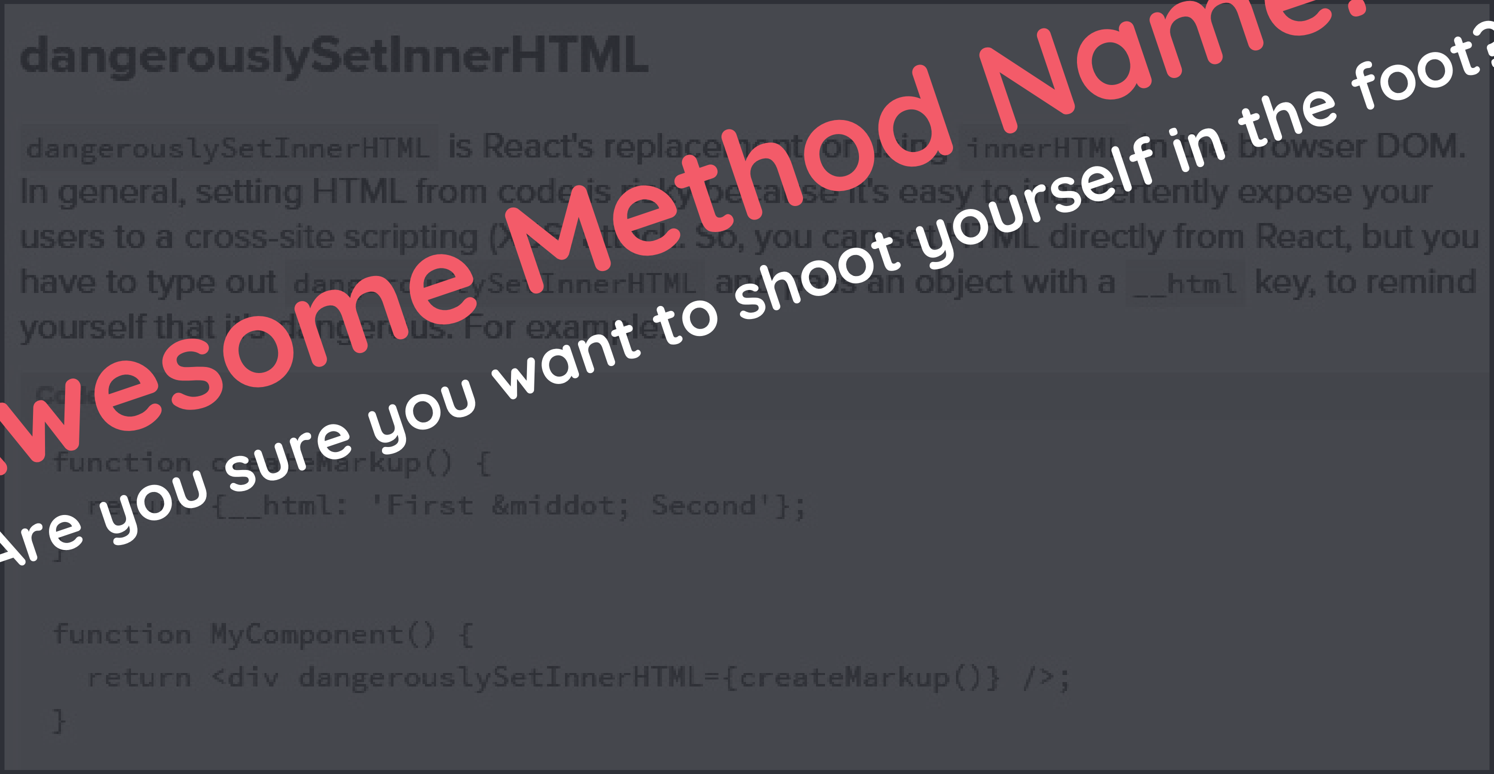
`dangerouslySetInnerHTML` is React's replacement for using `innerHTML` in the browser DOM. In general, setting HTML from code is risky because it's easy to inadvertently expose your users to a cross-site scripting (XSS) attack. So, you can set HTML directly from React, but you have to type out `dangerouslySetInnerHTML` and pass an object with a `__html` key, to remind yourself that it's dangerous. For example:

Code

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

Defence - Encoding (Frontend)

dangerouslySetInnerHTML



dangerouslySetInnerHTML

`dangerouslySetInnerHTML` is React's replacement for using `innerHTML` in the browser DOM. In general, setting HTML from code is risky because it's easy to inadvertently expose your users to a cross-site scripting (XSS) attack. So, you can set HTML directly from React, but you have to type out `dangerouslySetInnerHTML` and pass an object with a `__html` key, to remind yourself that it's dangerous. For example:

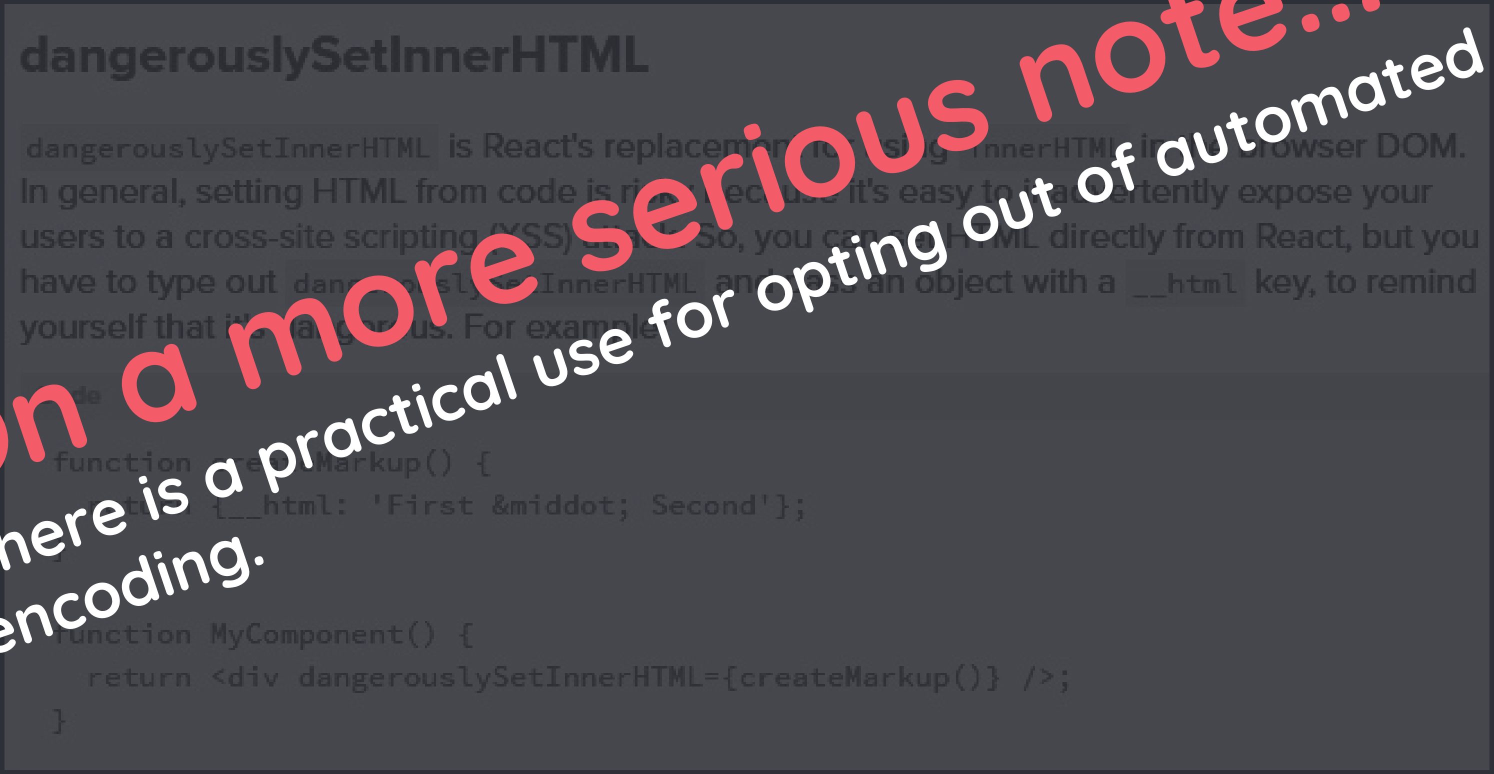
```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

Awesome Method Name!

"Are you sure you want to shoot yourself in the foot?"

Defence - Encoding (Frontend)

dangerouslySetInnerHTML



dangerouslySetInnerHTML

`dangerouslySetInnerHTML` is React's replacement for using `innerHTML` in the browser DOM. In general, setting HTML from code is risky because it's easy to inadvertently expose your users to a cross-site scripting (XSS) attack. So, you can set HTML directly from React, but you have to type out `dangerouslySetInnerHTML` and pass an object with a `__html` key, to remind yourself that it's dangerous. For example:

Code

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

On a more serious note...

There is a practical use for opting out of automated encoding.

Defence - Encoding (Back-end)

HTML Encoding for Developers

Still want to do encoding on the server-side manually?

- Use an established library!
  - .NET (If you are not using Razor)
    - System.Web.HttpUtility.HtmlEncode
  - Java
    - StringEscapeUtils.esapeHTML

Don't write your own encoding library

What would happen if we HTML Encoded Everything?

It is

Demo Time

Again :D

OH NOES! :(

Defence

- Another user defined data was found used the page:
  - Alternate text for the user's avatar

```
<img src='auto generated url'
alt='Username'/>
```

Defence

Username:

```
<script>alert(1);</script>
```

With HTML Encoding:

```
<img src = 'generated_url'
alt = '&lt;script&gt;alert(1);&lt;/script&gt;' />
```

Defence

Username:

```
' onload=alert(1) v='
```

With HTML Encoding:

```
<img src = 'generated_url'
alt = '' onload=alert(1) v='' />
```

Note: Not all HTML Encoder encodes the apostrophe character.

Defence

Username:

```
' onload=alert(1) v='
```

With HTML Encoding:

```
<img src = 'generated_url'
alt = '' onload=alert(1) v='' />
```

Note: Not all HTML Encoder encodes the apostrophe character.

Let's talk about Encoding

(Again)

Encoding Again

This time the user defined data was used inside a HTML **attribute**.

Other examples of user data in attributes:

```
<input type="text" value="user data" />
<img src="user data">
```

Another Encoding mechanism must be used in this scenario.

Attribute Encoding

Character	Attribute Encoded
'	&#39;
"	&quot;

Defence

Username:

```
' onload=alert(1) v='
```

With Attribute Encoding:

```
<img src = 'some auto generated url'
alt = '&#39; onload=alert(1) v=&#39;' />
```

Attribute Encoding for the Developers

If you are using templates

Make sure you wrap user input in quotes!

```
<img src="blegh" alt="{{user_input}}">
```

Defence

Attribute Encoding for the Developers

Use the appropriate attribute encoding method in your framework.

- Use an established library!
  - .NET
    - System.Web.HttpUtility.Html**Attribute**Encode
  - Java (OWASP Encoder)
    - org.owasp.encoder.Encode.forHTML**Attribute**

Knowing when to use which encoding is important!

Context

HTML

```
<div>user input</div>
```

HTML Attribute

```
<input value="user input">
```

URL

```
http://mysite/index?title=user input
```

Context

Javascript Escaping

```
<script>var title = user input;</script>
```

Style / Cascading Style Sheet

```
background-image: user input;
```

And some others...

Context

Sometimes you need to use multiple encodings!

```
<script>
var title = ' ';alert(123); </script>
<script>alert(1);//';
</script>
```

Context

Sometimes you need to use multiple encodings!

```
<script>
    var title = ' ';alert(123);
</script>
<script>
    alert(1);//';
</script>
```

Context

Sometimes you need to use multiple encodings!

```
<script>
    var title = ' ';alert(123);
</script>
<script>
    alert(1);//';
</script>
```

Let's talk about other ways to mitigate XSS

Let's talk about

Defence In Depth

What is Defence in Depth?

"A concept in which multiple layers of security controls (defense) are placed throughout an information technology (IT) system. Its intent is to provide redundancy in the event a security control fails or a vulnerability is exploited that can cover aspects of personnel, procedural, technical and physical security for the duration of the system's life cycle."

- *Smart fellow(s) from Wikipedia*

Defence in Depth

What is Defence in Depth?

Tl;dr *Multiple layers of security mechanisms will make the attacker's life more difficult.*

Image source: https://upload.wikimedia.org/wikipedia/commons/6/6f/Caerlaverock_Castle_from_the_air_1.jpeg

Defence in Depth for Cross Site Scripting

- Perform Input Validation
- Implement Content Security Policy
- Tag sensitive cookies with security flags

Input Validation

- Should you allow special characters such as < and > in some fields?
- A **whitelist** approach is always preferred over blacklist
- Reject fields that have failed validation
- Ensure that input validation is used consistently across all points of input

Input Validation

Special mention for user defined URLs!

```
<a href='user input'>My site</a>
```

Javascript can be embedded by prefixing the link with **javascript:**

For example:

```
<a href='javascript:alert(1);'>My Website</a>
```

Input Validation

Special mention for user defined URLs!

```
<a href='user input'>My site</a>
```

Validation Strategy:

- Fail the validation if it starts with Javascript:
- Validate that the user data is a valid URL
- (Optional) Check if URL is on a blacklist

Content Security Policy (CSP)

A mechanism for deterring the following attacks:

- Cross Site Scripting
- Clickjacking
- Other Misc. attacks

Defence in Depth - Content Security Policy

Content Security Policy (CSP)

A mechanism for deterring the following attacks:

- **Cross Site Scripting** ⟵ **We'll only talk about this one today**
- Clickjacking
- Other Misc. attacks

What is Content Security Policy?

A series of directives/instructions in:

- HTTP response header

  -
    ```
    HTTP/1.1 200 OK
    …
    Content-Security-Policy: …policy definition here…
    ```
- Meta tag on the page

  -
    ```
    <meta
            Http-equiv = "Content-Security-Policy"
            Content = "…policy definition here…">
    ```

Defence in Depth - Content Security Policy

How does CSP prevent Cross-Site-Scripting?

Let's have a look at an example policy and see how it works...

```
Content-security-policy:
  default-src 'none';
  script-src 'self' cdn.mysite.com;
  style-src 'self' cdn.mysite.com;
```

Example Policy

```
Content-security-policy:
  default-src 'none';  ←  Don't load any external javascript, images, css etc.
  script-src 'self' cdn.mysite.com;
  style-src 'self' cdn.mysite.com;
```

Example Policy

```
Content-security-policy:
  default-src 'none'; ← Don't load any external javascript, images, css etc.
  script-src 'self' cdn.mysite.com; ← Load .js from same origin or the CDN
  style-src 'self' cdn.mysite.com;
```

Example Policy

```
Content-security-policy:
  default-src 'none'; ← Don't load any external javascript, images, css etc.
  script-src 'self' cdn.mysite.com; ← Load .js from same origin or the CDN
  style-src 'self' cdn.mysite.com; ← Same applies to .css files
```

Example Policy

```
Content-security-policy:
  default-src 'none'; ← Don't load any external javascript, images, css etc.
  script-src 'self' cdn.mysite.com; ← Load .js from same origin or the CDN
  style-src 'self' cdn.mysite.com; ← Same applies to .css files
```

Note

Inline Javascripts cannot be executed if CSP is in use, unless the unsafe-inline directive is being used.

Demo with the following CSP directives enabled

```
default-src 'none';
```
Don't load any external javascript, images, css etc.

```
script-src 'self' cdnjs.cloudflare.com;
```
Only allow the loading of .js from same origin or Cloudflare CDN

```
style-src 'self' cdnjs.cloudflare.com;
```
Only allow the loading of .css from same origin or Cloudflare CDN

```
img-src www.gravatar.com;
```
Only allow images from gravatar.com to be loaded

```
report-uri /csp-report;
```
Report all violations to the above endpoint

Challenges with CSP

- Inconsistent support across different browsers
- Implementing and fine tuning CSP policy is hard
  - Regression challenges
  - (You can always turn it on in report-only mode)
- May not play well with legacy libraries and frameworks

Cookie Security Flags

- Prevent your precious session cookies from being stolen by evil Javascript with the two following flags.
- **HttpOnly**: Cookie is not accessible via Javascript
- **Secure**: Cookie can only be sent via HTTPS

Now
For the
Takeaway
Message

(You don't have to put up with me for much longer)

Takeaway

Developers Developers Developers

- Know where user data's used on the page
- Know the frameworks you are using
- Encode / Escape user data properly
- Validate input when appropriate
- Set cookie security flags
- Use Content Security Policy

Takeaway

Testers Testers Testers

- Take note of pages that contain user data
- Test by inserting script and see if they executed
- Look for XSS as a part of your quality assurance process
- Use a proxy:
    - ZAP, Burp, Charles, Fiddle
- Ask your security team for guidance
- Automate whenever possible

Misc.

Useful Links

More info on XSS

https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

https://www.owasp.org/index.php/Testing_for_Cross_site_scripting

https://www.google.com/about/appsecurity/learning/xss/

https://excess-xss.com/

Test Strings for the QAs

http://ha.ckers.org/xss.html

http://htmlpurifier.org/live/smoketests/xssAttacks.php

Content Security Policy (CSP)

https://developers.google.com/web/fundamentals/security/csp/

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

Misc.

Useful Links

Proxies:

Burp (free edition): http://portswigger.net/burp/

ZAP: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

Fiddler: http://www.telerik.com/fiddler

Charles: https://www.charlesproxy.com/

Exercises:

The XSS Game: https://xss-game.appspot.com/

Google Gruyere: https://google-gruyere.appspot.com/

XSS/SQLi Lab VM Image: https://pentesterlab.com/exercises/xss_and_mysql_file

BeEF when you really want to mess around with XSS:

Browser Exploitation Framework (BeEF): https://github.com/beefproject/beef

Slide theme from slidescarnival.com

Cheers

- Cheers

and have an awesome afternoon:D