



# Making Secure, Accessible, and Robust Web Applications

CIS I 962 (Fall 2025)  
November 24th, 2025

# Lesson Plan

5	Deployment
16	Website Security
25	Defenses and Best Practices
39	Website Monitoring & Analytics
47	Accessibility and a11y

# PollEverywhere!

We will use PollEv to take attendance and do polls during lecture. Scan this QR Code or use the link to join for attendance! (Please add your name for identification)

[PollEv.com/voravichs673](https://PollEv.com/voravichs673)



- Project Milestone 1 due tonight
- Homework 4 due tonight
- Homework 5 due next week



# Deployment

How do we make the web applications we build publicly available to the world?

# From Development to Production

So you've finally finished development, and you want to ship your web app for people to use. What are your options?



# From Development to Production

**Development** environments are where you write, test, and debug code. Often in web development, we work on a localhost server on local machines with test data to make sure our apps work.

**Production** environments are live settings where real users get to interact with the application. At this point, code should be deployed with stable infrastructure, secure backends, and optimized performance.

# Deployment

**Deployment** takes our source code used in the development and puts it in a publicly accessible server. With web applications, this often involves tying it to a domain name users then access in a browser.

There are many common deployment options for websites:

- Static Site Hosting
- Shared Hosting
- Cloud Servers
- Platform as a Service



# Deployment Workflow

1. Develop your app locally, make sure it's bug free and well-tested
2. Build your app, either locally or handled by the server host
3. Upload build files or source code repository to a server host
4. Set up app configurations, like environment variables, database credentials, domain name, and more
5. Your site goes live!

# Static Site Hosting

**Examples:** Netlify, Vercel, Github Pages

**Static Site Hosting** is designed to host pre-built files (HTML/CSS/JS and other assets) with no constantly-on backend.

Services like Netlify and Vercel allow serverless functions and API endpoints that run only in response to HTTP requests to allow a backend/database connections to exist.

**Examples:** Bluehost, SiteGround, DreamHost

**Shared Hosting** features multiple websites sharing a server. Clients may get a slice of disc space, memory, and CPU on that server on an assigned server.

You can manage files and resources on dedicated dashboards, and have support for PHP and MySQL.

**Examples:** AWS, DigitalOcean, Azure VMs, Google Cloud

**Virtual Private Servers(VPS) or Cloud Servers** are hosting options where you rent a server or provision a cloud instance and host your application there.

This option provides the most control, but requires you to setup every part of your project yourself via SSH into your server.

# Platform as a Service (PaaS)

**Examples:** Heroku, Render, Railway

**Platform as a Service (PaaS)** is a hosting option that includes a server, but the platform itself handles the server management, including network, scaling, and security.

This option keeps your apps scalable, but may become expensive and less flexible since you have less control over the servers.

# Demo: Vercel with Postgres

We'll demonstrate taking an app with some API routes in [Next.js](#) with Postgres integration to deployment in Vercel.

On Vercel, you can use their dashboard to quickly import and build projects on their platform with git.

Any pushes to the git repo will rebuild the app to reflect the changes!

- Vercel (Recommended for simple deployment cases)
- Netlify
- DreamHost
- Heroku
- Google Cloud



# Website Security

What are some common threats to websites and how do we prevent them?



# The Importance of Security

Nowadays, websites have to handle a lot of sensitive user information (personal information, payment details).

Security breaches on websites happen daily, with large data breaches causing companies legal, financial, and reputational harm.

Many vulnerabilities come from unsafe practices when writing JavaScript that malicious actors can exploit.

# Common Attacks: XSS

Cross-Site Scripting, or XSS, allows attackers to inject JavaScript into web pages in a way that is visible to other users, allow them to steal sensitive data, hijack accounts, or spread malware.

This could take the form of:

- Reflected XSS: unvalidated user input includes a runnable script

```
https://site.com/?q=<script>alert(1)</script>
```

- DOM XSS: Client-side JS allows browser to process malicious input

# Demo: Inject Some Code

```
<body>
  <script>
    document.body.innerHTML += "<div>" + decodeURIComponent(location.hash.substr(1)) + "</div>";
  </script>
</body>
```

Serve up this HTML locally, then visit:

[host]/#hello

and

[host]/#<img%20src=x%20onerror=alert('XSS')>



# Case Study: Twitter and XSS

## Article

Due to improper input sanitation with URLs in tweets, people were able to write JavaScript to be embedded into tweets themselves.

People wrote worms (self-replicating malware) that caused tweets to be sent automatically as people viewed infected tweets!

### Realtime results for Onmouseover

33,053 more tweets since you started searching.



**JeanneDoo** <http://a.no>  
/@"onmouseover=",\$('textarea:first').val(this.innerHTML);\$('status-update-form').submit();"class="modal-overlay"/  
less than 20 seconds ago via web



**RCA Promo** RT @miguelarga:  
[www.t.co/@miguelarga](http://www.t.co/@miguelarga) /@"onmouseover="document.getElementById('status').Miguelarga;\$('status-update-form').submit();"class="modal-overlay"/  
less than 20 seconds ago via web



**Marv Carbonado** <http://a.no>  
/@"onmouseover=",\$('textarea:first').val(this.innerHTML);\$('status-update-form').submit();"class="modal-overlay"/  
half a minute ago via web



**RobinwoodChurch** <http://a.no>  
/@"onmouseover=",\$('textarea:first').val(this.innerHTML);\$('status-update-form').submit();"class="modal-overlay"/  
half a minute ago via web

# Common Attacks: Session Hijacking

**Session Hijacking**, or **sidejacking**, is an attack that exploits unsecure session handling. Unprotected session cookies and authentication tokens can be stolen to impersonate users on websites.

This attack can be prevented by using an HTTPS (HTTP Secure) protocol on all pages of a website, not just login. Most website hosting platforms like Heroku and Vercel already provide HTTPS.

**Padlock means your are secure/on HTTPS!**



# Case Study: Firesheep

## Article



Firesheep was a Firefox browser extension that was able to capture session cookies transmitted over unsecured Wi-Fi networks (like public Wi-Fi) and allow hijackers to impersonate users on websites with their login credentials.

Threats like these are why you are warned about signing into public Wi-Fi networks or are encouraged to use a VPN!

# Common Attacks: CSRF

**Cross-Site Request Forgery (CSRF)** allows attackers to send requests to real websites (if credentials are available and a session cookie is still active) from false, malicious websites.

Many modern sites protect against CSRF by blocking cookies on cross-site POST requests (cookie attribute: SameSite=Lax/Strict), while frameworks like Express also have CSRF tokens to secure cookie-based authentication.

# Case Study: TikTok CSRF

## The Bug Report

A certain TikTok API endpoint allowed someone to change the password on users who signed up with third party apps via CSRF. Combined with a reflected XSS bug in a URL parameter, this essentially allowed a “one-click account takeover.”





# Defenses and Best Practices

How do you best defend against threats to web applications?

# Defenses and Best Practices

- Input Validation and Sanitization
- Content Security Policy
- Secure Use of Cookies
- Use JavaScript Frameworks' built-in security features
- Keep your dependencies up to date, detect vulnerable dependencies, and fix/remove/replace them

# Validation & Sanitization

Ensuring that any user input cannot be interpreted as code will prevent XSS from occurring.

**Validation:** Ensure that inputs match the types and format you want, for instance emails match a specific regex.

**Sanitization:** Remove potentially dangerous characters or markup, like HTML tags

- **Escaping:** turning special characters that could be interpreted as HTML (<, >, “) into text forms (&lt, &gt).
- Frameworks (like React) may help by auto-escaping characters!

# Validation Example



```
document.getElementById('greeting').innerHTML = "Hello, " + userInput + "!";
```

**Can XSS if userInput is: `<script>alert("XSS")</script>`**



```
document.getElementById('greeting').textContent = "Hello, " + userInput + "!";
```

**No XSS if .textContent is used**



```
return <div>Hello, {userInput}!</div>;
```

**React auto-escapes**

# Content Security Policy

A **content security policy (CSP)** is a browser feature that restricts what resources a page can load, like scripts.

This helps catch XSS and other injection attacks by disallowing unsafe scripts.

# CSP Example

The CSP for this slide deck (on Google Slides) is as follows:

```
require-trusted-types-for 'script';  
report-uri https://csp.withgoogle.com/csp/docs-tt;  
base-uri 'self';  
object-src 'self' blob;;  
report-uri https://docs.google.com/presentation/cspreport;  
script-src 'nonce-jxHw5dBcS6GrK_YnPYLpsw' 'unsafe-inline' 'strict-dynamic'  
https: http: 'unsafe-eval';  
worker-src 'self' blob;;
```

This blocks common unsafe code like innerHTML, eval, (Trusted Types), whitelists certain scripts (script src), and reports CSP violations to Google.

# Same-Origin Policy & CORS

**Website origin:** Combination of a websites protocol (HTTP/HTTPS), hostname, and port.

`https://example.com:443`

**!=**

`https://example.com`

`http://example.com`

**!=**

`https://example.com`

By default, browsers have a **Same-Origin policy**, which blocks cross-origin requests. You have to enable Cross-Origin Resource Sharing (CORS) in order to allow your server's resources to be shared with others.

# Browser Cookies

**Browser Cookies** are used to session-based data on the web.

Safe cookie usage often uses certain security flags (or have safe defaults:

- `HttpOnly`: JS can't access this cookie, protects from theft during XSS
- `Secure`: Cookie is sent through HTTPS(encrypted connections)
- `SameSite`: Controls whether browser sends cookies on cross-site requests, either `Lax` (on top-level navigation) or `Strict` (never). Prevents CSRF



# Dangerous Code/APIs

`eval()` is evil!!!!

This function lets you run code from a string, which is extremely vulnerable to XSS attacks.

Additionally, APIs like `localStorage` and `sessionStorage` are easily accessible with JS with little protection, meaning that you should avoid keeping sensitive data in those structures.

# Security Tools

Dependencies in your package.json may contain vulnerabilities and security flaws that attackers scan for as targets. It's important to keep your dependencies updated, or be aware of these vulnerabilities!

`npm audit` is a command that will detect and fix vulnerabilities.

Platforms like <https://snyk.io/> can help scan for vulnerabilities in your GitHub repos with a wide database of vulnerabilities.

# Demo: npm audit and snyk

```
npm i react-syntax-highlighter@15.6.1
```

This dependency is an older version (Current 16.1.0) that includes a vulnerability. `npm audit` or `snyk test` will catch this.

<b>moderate</b>	PrismJS DOM Clobbering vulnerability
Package	prismjs
Vulnerable versions	<1.30.0
Patched versions	>=1.30.0
Paths	.>react-syntax-highlighter>refractor>prismjs
More info	<a href="https://github.com/advisories/GHSA-x7hr-w5r2-h6wg">https://github.com/advisories/GHSA-x7hr-w5r2-h6wg</a>

```
Tested 29 dependencies for known issues, found 1 issue, 1 vulnerable path.
```

```
Issues to fix by upgrading:
```

```
Upgrade react-syntax-highlighter@15.6.1 to react-syntax-highlighter@16.0.0 to fix  
X Arbitrary Code Injection [Low Severity][https://security.snyk.io/vuln/SNYK-JS-PRISMJS-9055448] in prismjs@1.27.0  
introduced by react-syntax-highlighter@15.6.1 > refractor@3.6.0 > prismjs@1.27.0
```

# Resources

- [OWASP 2025 Top 10 \(Common Security Threats\)](#)
- [MDN Web Security Docs](#)
- [Web Security Academy](#)
- [Google Web Fundamentals- Security](#)
- <https://snyk.io/>



# 5-Minute Break!

---



# Website Monitoring

What's the importance of monitoring and in what ways  
can we monitor user activity in apps?

# Monitoring

Web app development doesn't end at deployment. It is important to monitor user actions and app performance, and also detect suspicious activity that could lead to attacks.

- Front-end Logging
- Monitoring Services
- Performance metrics for optimization
- User activity data for UI/UX research

# Logging in the Frontend

You should already be familiar with being able to `console.log()` on the frontend of your app. You may also send your logs to a third-party service or a backend.

For monitoring purposes, logging helps with:

- Logging app errors
- Logging user actions (button click, navigation)
- Performance metrics (page load times)



# Monitoring Services

Monitoring services connected to your app provide a clean dashboard for errors and logs. These may also help automate your monitoring, see trends, and output alerts to services like Slack or email.



**DATADOG**



**SENTRY**



**LogRocket**

# Performance Metrics

A deployed app may differ wildly when used by real users from a development environment.

Below are some important metrics you can test:

- First Contentful Paint (FCP)
- Largest Contentful Paint (LCP)
- Cumulative Layout Shift (CLS)
- Interaction to Next Paint (INP)

Google's web-vitals package covers all these!

# Demo: Google web-vitals

npm i web-vitals

```
import { useEffect } from "react";
import { onFCP, onLCP, onINP } from "web-vitals";

function sendToAnalytics(name: string, value: number) {
  console.log('Web Vitals Metric:', name, value);
}

export default function WebVitalsMonitor() {
  useEffect(() => {
    onFCP(metric => sendToAnalytics('FCP', metric.value));
    onLCP(metric => sendToAnalytics('LCP', metric.value));
    onINP(metric => sendToAnalytics('INP', metric.value))
  }, []);
  return null;
}
```

# User Analytics

Monitoring the behavior of users, such as where they click and how long they stay on your website, helps developers see patterns and prioritize chances to improve user experiences.

Things you may monitor include:

- Navigation, Clicks, and Scroll Depth
- Session length and bounce rates (how many people immediately leave instead of clicking more links)
- Interactions like video playing, image carousel use, etc.

# Demo: A/B Testing

A/B Testing is common in UI/UX research to see what layout works best. Let's run a simple test to show off Sentry for monitoring user activity.

<https://1962-25fa-ab-test-demo.vercel.app/>

Go to this link and click the button!

Let's get some A/B test data!

# Google Lighthouse Audit

Google Lighthouse is a service that provides quick and easy analysis of performance metrics and various other metrics, like accessibility and SEO.

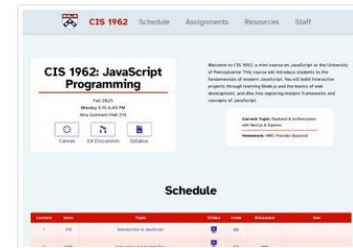
## Audit for our Class Website



Performance

Values are estimated and may vary. The [performance score](#) is calculated directly from these metrics. [See calculator.](#)

▲ 0-49    ■ 50-89    ● 90-100





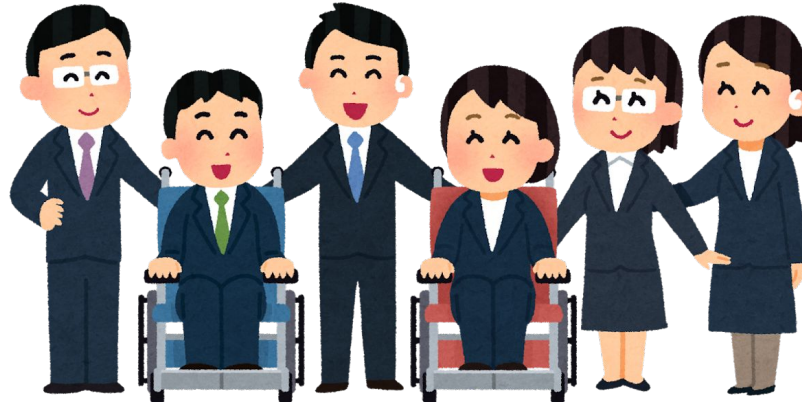
# Accessibility

How do we design our apps to be accessible for users with disabilities and different circumstances?

# Why Accessibility Matters

1 in 4 adults have disabilities.

Designing websites to be accessible helps reach wider audiences and comply with legal standards!





# Why Accessibility Matters

```
...  
<div onclick="submitForm()" class="fancy-button">  
  Apply Now  
</div>
```

Select a role

Apply

# Why Accessibility Matters

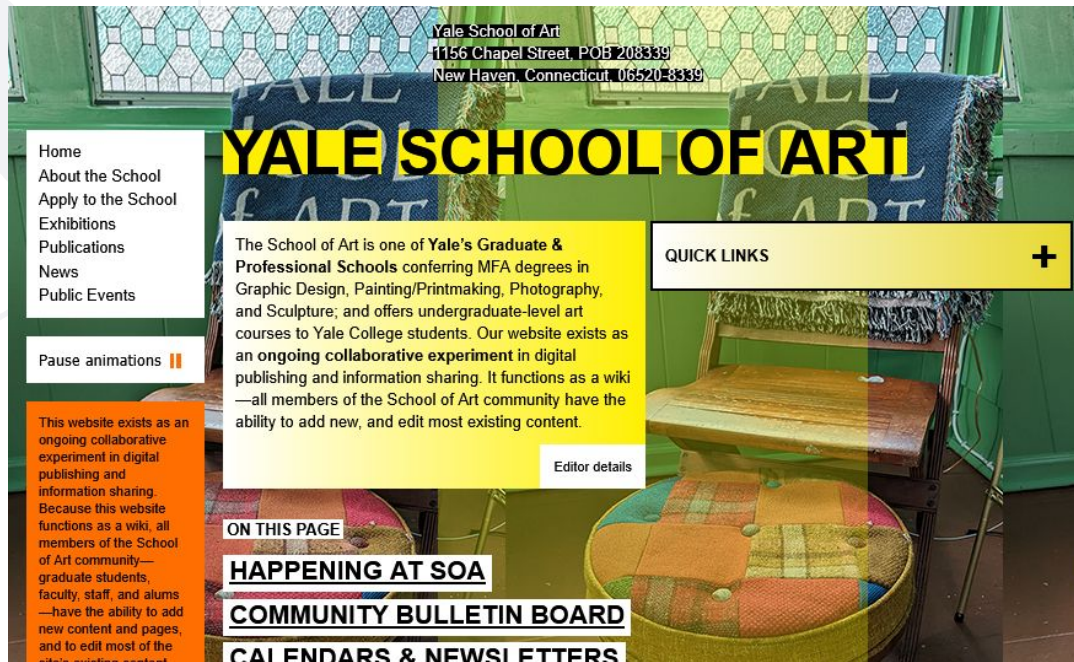
```
<button  
  type="submit"  
  class="fancy-apply-btn"  
  aria-label="Apply for selected job"  
>  
  Apply  
</button>
```

Select a role

Apply

# Bad Accessibility is Noticeable

<https://www.art.yale.edu/>



# A11y and WCAG

**A11y** is an abbreviation for **accessibility**, and represents the initiative to design digital spaces that everyone can use. This means considering not just disabilities, but also slower internet connections and mobile devices.

WCAG (Web Content Accessibility Guidelines) is a set of standards are recommendations for developing accessible web content.

[WCAG2](#)

# Content should be Perceivable

Information and user interface components must be presentable to users in ways they can perceive.

Examples:

- Text alternatives for images (alt text)
- Captions for videos
- Good color contrast (avoid bad color contrast)
- Use semantic HTML tags to structure a page and provide purposes and relationships between elements

# Content should be Operable

Interface components and navigation must be usable by all, including with a keyboard.

Examples:

- All operations, interactions, and navigation can be performed with keyboard only
- Define focus pseudo-classes when using a keyboard
- Avoid content flashing quickly (avoiding seizures)
- Provide time for users to read and use content

# Content should be Understandable

Information and operation of the UI must be clear and predictable.

Examples:

- Provide clear and simple language, and provide definitions for abbreviations, vernacular, and jargon
- Navigation is consistent across the entire interface
- Changes of context (like updates and navigation) should be initiated by the user, rather than sudden and automatic
- Help users avoid and navigate through errors (like clearing form errors, providing error feedback explicitly)

# Content should be Robust

Content must be robust enough to work reliably with current and future technologies, including assistive tech.

Examples:

- Use Semantic HTML!
- Maximize compatibility between different browsers and devices such as mobile and tablet
- Use ARIA to provide roles and assistance where HTML is not enough



# ARIA

**Accessible Rich Internet Applications (ARIA)** are a set of roles and attributes you can add to HTML to add meaning to pages for assistive technology like screen readers.

While many semantic HTML elements are already accessible (`<button>`, `<nav>`), it's important to use ARIA to provide extra context for custom elements or plain `<div>` elements.

```
<button aria-pressed="true">
```

```
<div aria-live="polite">
```

# Testing a11y

Can you navigate through your whole app with only the Tab, Space, and Enter keys?

Test page content with screen reader apps like NVDA

Use automated tools like:

- Lighthouse
- Axe DevTools

# Resources

- [a11y Project](#)
- [WCAG2](#)
- [MDN ARIA Docs](#)