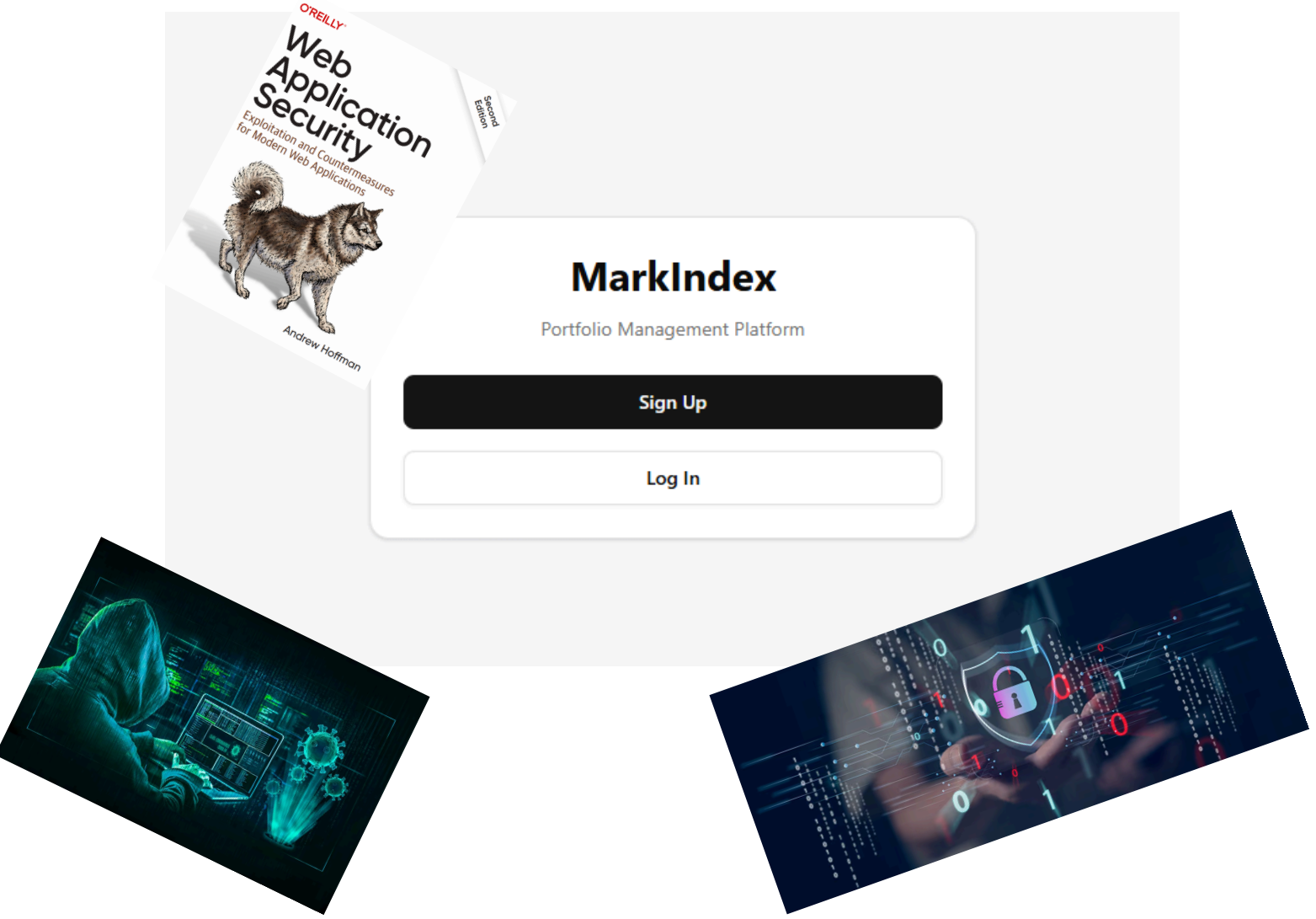
Markindex Web Security



Udarbejdet af: Steffen Grøn Andersen
Fag: Web Security
Dato: 08-12-2025

Introduction

This project serves as a comprehensive security assessment and implementation exercise for a web application built for Markindex, a marketing analytics software company. The primary purpose is to demonstrate practical understanding of web security principles by identifying common vulnerabilities and implementing appropriate countermeasures. Rather than building a feature-rich application, this project focuses on creating a security-hardened foundation that Markindex can use as a template for migrating their existing applications to a more secure architecture.

Company backstory

Markindex is a marketing analytics software company with a focused mission: to provide comprehensive marketing analytics for e-commerce businesses through collaborative data sharing. Markindex introduces a secure platform for anonymized marketing data sharing. The platform enables e-commerce companies to contribute their marketing performance data and, in return, gain broader insights into industry advertising trends and benchmarks.

Markindex operates as a solo entrepreneurship, founded and run by a single developer with expertise in both web development and marketing analytics. This lean structure allows for rapid iteration and decision-making but also places the entire burden of security implementation and maintenance on one individual, making the choice of technology stack and security architecture critically important.

The fundamental problem Markindex must solve is establishing absolute trust in data confidentiality and anonymization. This creates an exceptionally high security bar. The platform must ensure that no data is leaked through vulnerabilities like SQL injection, XSS, or insecure direct object references. All publicly visible information must be thoroughly anonymized with no possibility of de-anonymization.

Technology stack

Markindex uses the following technology stack, which I will now briefly go through from a security perspective.

Knowing the tech stack is essential because each component introduces its own risks, default protections, and potential vulnerabilities. Understanding it ensures we can secure every layer of the system.

With the stack defined, we can review each technology to identify its security implications and any vulnerabilities Markindex should monitor or mitigate.

Frontend

The frontend stack is built on modern, widely adopted technologies that shape how the user interface is rendered, styled, and secured.

- NextJS App Router (vs Pages router)
- React
- Tailwind
- TypeScript
- ShadCN

Next.js serves as both the frontend framework and backend API platform for Markindex. From a security perspective, the App Router (introduced in Next.js 13) provides several built-in protections but also introduces specific security considerations that differ from the older Pages Router.

Built-in Security Features:
Automatic CSRF protection for Server Actions
Built-in support for Content Security Policy (CSP) headers
Secure defaults for HTTP headers

React handles all UI rendering and client-side interactivity in Markindex. While React provides automatic XSS protection through JSX escaping, developers can still introduce vulnerabilities through improper usage. A recently published vulnerability, CVE-2025-55182, highlighted that even mature frameworks like React can expose unexpected security risks — something I will return to in a later section.

While TypeScript doesn't prevent security vulnerabilities directly, it reduces bugs that could lead to security issues. TypeScript must always be paired with runtime validation (using Zod in Markindex's case).

Backend

The backend stack provides the core application logic, data validation, and authentication mechanisms, making it a critical layer for enforcing security and preventing malicious input from ever reaching the database or client.

- NextJS
- Argon2 password hashing
- Zod

Next.js is also used on the backend to handle server-side routes. While the frontend section covers its broader features, it is important here simply to note that Markindex relies on its server-side capabilities to separate trusted backend logic from untrusted client input.

Argon2 is used for password hashing, providing strong, slow, and salted hashes that protect user credentials even in the event of a database breach. Security depends on choosing an appropriate cost factor and never handling raw passwords outside controlled code paths.

Zod provides strict runtime schema validation, ensuring all incoming data—whether from the frontend, external APIs, or internal processes—is validated before being processed. This

significantly reduces risks such as injection attacks, malformed data, and privilege escalation caused by unintended input structures.

Database

The database layer is responsible for securely storing all persistent application data, making it essential to control access, validate queries, and prevent unauthorized exposure.

- PostgreSQL
- Drizzle ORM

PostgreSQL provides a robust and mature relational database with strong security features, including role-based permissions, row-level security, and encrypted connections. Proper configuration is crucial, as misconfigured privileges or open ports can expose sensitive data or allow unintended access.

Drizzle ORM acts as a type-safe query layer between the application and PostgreSQL. By generating structured queries rather than relying on raw SQL, it significantly reduces the risk of SQL injection. Security still depends on correct schema design and ensuring that queries enforce proper authorization rules.

Infrastructure

The infrastructure layer defines where the application runs and how it is exposed to the internet, making it a key component in preventing unauthorized access, securing network traffic, and isolating services.

- Hetzner
- Ubuntu Linux
- Cloudflare
- Caddy
- Docker

Hetzner provides the underlying hosting environment. Security depends on proper account hardening, SSH configuration, firewall rules, and ensuring that only required services are publicly accessible.

Ubuntu serves as the server operating system. Keeping it secure requires regular updates, minimal installed packages, strict user permissions, and hardening measures such as disabling password logins and restricting sudo access.

Cloudflare provides an additional security perimeter through features like DDoS protection, TLS termination, rate limiting, and WAF rules. Correct configuration is essential to prevent bypasses or misrouted traffic.

Caddy is used as the reverse proxy and handles automatic HTTPS. Its security advantages rely on proper routing rules, header configuration, and ensuring that backend services are not exposed directly to the public internet.

Docker containerizes the application, isolating services and simplifying deployments. Security depends on limiting container privileges, avoiding root containers, keeping images up to date, and restricting network communication between containers unless explicitly required.

CI/CD

The CI/CD pipeline automates building, testing, and deploying the application, making it a critical point for maintaining code quality and security throughout the development lifecycle.

- Github Actions

GitHub Actions manages automated workflows for Markindex. Security depends on properly handling secrets, restricting workflow permissions, and ensuring that only trusted code can trigger deployments, preventing supply chain or injection risks.

Analytics

While analytics tools are not part of core application functionality, any dependent infrastructure can introduce security risks and must be carefully managed.

- PostHog

PostHog is used for analytics and user behavior tracking in Markindex. Although normally not a focus in security reviews, it is important to recognize that third-party services can be compromised, as has happened with PostHog in the past—a topic I will return to in a later section.

Application features

The application was developed as a secure-by-design Next.js scaffold intended to demonstrate how common web-application features can be implemented while consciously mitigating the most prevalent security risks.

Rather than building functionality in isolation, each feature was guided by a user story to ensure clarity of purpose and a consistent link between user needs and the technical decisions made during development.

Signup, Login and Session Management

US001
As a visitor I want to create a new account so that I can access the platform and be assigned to businesses.

US002
As a registered user I want to log in with my credentials so that I can access my assigned businesses and portfolios

US003
As a logged-in user I want to stay logged in across browser sessions so that I don't have to re-authenticate constantly.

Technical summary

Data transfer from the frontend to the backend is handled using Next.js server actions and form submissions, ensuring that all critical logic runs on the server. Input is validated with a Zod schema before any processing occurs.

The authentication logic follows the copenhagenbook methodology, where passwords are hashed using Argon2 and a session token is created upon successful login. The session token is stored in a cookie, lives for thirty days unless extended, and is deleted when the user logs out.

All database interactions are made through Drizzle ORM. An optimistic check for a session token is performed in the proxy file on every request, and the session token is fully validated both when rendering pages and when handling form submissions.

All error messages returned to the user remain intentionally generic to avoid revealing internal logic.

Security rationale

Using server actions ensures that user input never bypasses the backend’s trusted execution environment. Validating all input with Zod on the server prevents malformed or malicious data from entering the system.

The copenhagenbook methodology provides a secure foundation for managing identity and authentication, while Argon2 hashing ensures passwords cannot be recovered if the database is compromised. Storing the session token in a cookie keeps session handling simple and consistent, and the controlled thirty-day lifespan limits long-term exposure. Validating the session token on every request prevents unauthorized access and reduces the risk of stale or tampered sessions.

Drizzle ORM inherently protects against SQL injection by using parameterized queries, and the optimistic cookie check in the proxy layer allows quick rejection of invalid users before the request reaches the application's core logic.

Finally, returning only generic error messages prevents attackers from learning whether validation failed, whether an account exists, or how the system internally handles authentication steps, reducing the risk of user enumeration and targeted attacks.

Business Identity and Access Management

US004: Create Business
As a logged-in user I want to create a new business so that I can manage portfolios for that business

US011: View My Businesses
As a logged-in user I want to see a list of all businesses I have access to so that I can navigate to the business and view its portfolios

US005: Assign Users to Business
As a business admin I want to assign users to my business by their email address so that they can view portfolios and collaborate

Technical summary

The platform allows authenticated users to create businesses, view the businesses they belong to, and assign additional users to those businesses. Business listing is derived from the user's session token, which is resolved to a userId and used to fetch associated businesses. Creating a business is handled entirely on the server via the createBusiness action, which authenticates the caller and assigns them as the initial admin. Business navigation uses a UUID rather than a serial ID, and each business page performs both authentication and authorization checks before rendering. A user may only view a business if they are assigned to it, and assignment of users is performed by specifying their email address. All errors returned from these flows remain generic.

Security rationale

All data transfers from frontend to backend occur through server-only actions to avoid exposing internal business logic to the client. The user's session token is validated on every request, ensuring that business information is only accessible to authenticated users. Authorization is enforced per business by verifying that the requesting user is associated with the business before returning any data; users without access receive a neutral 404 response to avoid leaking the existence of the resource. Using a UUID instead of a numeric ID prevents predictability and enumeration of business identifiers. Assigning users by email requires no privilege beyond business admin rights, and all errors—such as nonexistent users or invalid operations—are intentionally generic to avoid revealing sensitive details about the system or its users.

Portfolio lifecycle

US007: Create Portfolio
As a business admin I want to create a portfolio for my business so that I can showcase work/projects to business members

US008: Toggle Portfolio Visibility
As a business admin I want to set portfolios as visible or hidden so that I can control which portfolios business members can see

US009: View All Portfolios (Admin)
As a business admin I want to see all portfolios for my business so that I can manage all content regardless of visibility status

US010: View Visible Portfolios (Business Member)
As a user assigned to a business I want to see all visible portfolios for that business so that I can view the work/projects I have access to

Technical summary

Business admins can create portfolios and manage their visibility for each business. Portfolio creation and visibility toggling are handled exclusively on the server through dedicated form actions that check the user's authorization. Admins have full access to all portfolios, including hidden ones, while members can only view portfolios marked as visible. Visibility controls and management actions are never rendered to non-admin users, ensuring that interface and functionality are only exposed to those with the appropriate role.

Security rationale

Server-side form actions enforce authorization for all portfolio management operations, preventing non-admin users from creating or altering portfolios. By hiding administrative controls in the frontend and verifying authorization on every server request, the system prevents privilege escalation through client-side manipulation. Portfolio visibility enforcement ensures that sensitive or incomplete work is not exposed to business members prematurely, while admins retain complete oversight. Using role-based access checks and returning generic error messages further reduces the risk of leaking information about portfolio existence or access permissions.

Business Assets (Media Uploads)

US006: Upload Business Logo
As a business admin I want to upload a logo for my business so that the business has visual branding

Technical summary

Business admins can upload a logo for their business, which is used for branding purposes. On the client side, optimistic checks are performed to ensure the file size and MIME type appear valid, providing immediate feedback in the UI. On the server, the file is fully validated for maximum size and allowed MIME types, and the image signature is checked to prevent type confusion attacks. Once validated, the image is converted to a Base64 string and stored in the database.

Security rationale

All file validation occurs on the server to prevent malicious files from bypassing client-side checks. Checking the image signature ensures that files are correctly typed and mitigates attacks that exploit MIME type spoofing.

Restricting size and allowed formats reduces the risk of resource exhaustion and unexpected behavior. Storing the image in Base64 within the database avoids file system access issues and limits exposure from path traversal attacks. Returning only generic error messages on failure prevents attackers from learning whether a particular file caused a rejection or why.

Collaboration and feedback

US012: Add Comment to Portfolio
As a user assigned to a business I want to add comments to visible portfolios so that I can provide feedback and collaborate with other team members

US013: View Portfolio Comments
As a user assigned to a business I want to see all comments on a portfolio so that I can read feedback and discussions

Technical summary

Users assigned to a business can add comments to portfolios and view existing comments. Admins have unrestricted access, allowing them to view and comment on all portfolios regardless of visibility. Business members can only see and add comments on portfolios that are marked as visible. All comment-related actions, including creation and retrieval, are handled by server-side form actions that authenticate the user, verify their role, and enforce portfolio visibility rules. Comment content is validated and sanitized before storage to ensure integrity and proper display.

Security rationale

Server-side handling ensures that users cannot bypass visibility restrictions or submit comments to portfolios they do not have access to. Authentication and authorization checks prevent unauthorized users from reading or writing comments. Sanitizing comment content

mitigates the risk of cross-site scripting (XSS) attacks. Role-based access control ensures that admins maintain full oversight while members are restricted to appropriate content, and generic error messages prevent leakage of information about portfolio existence or access permissions.

Security Implementation

SQL Injections Prevention

Drizzle ORM is used consistently across the entire codebase, and it automatically applies parameterized queries. This provides a strong and effective defense against SQL injection.

```
const [existingUser] = await db
    .select()
    .from(user)
    .where(eq(user.email, email))  // Parameterized
    .limit(1);
```

Cross-Site Scripting (XSS) Prevention

React automatically escapes all user-provided content rendered in JSX, ensuring that every location where user input is displayed is protected.

```
  <p className="text-sm font-medium text-zinc-900 dark:text-zinc-100">
    {comment.userEmail}  {/* Auto-escaped by React */}
  </p>
```

When httpOnly is enabled, JavaScript running in the browser cannot read or modify the cookie. This means even if an attacker injects malicious JavaScript into your page (XSS), they cannot steal your session token using document.cookie.

```
  cookieStore.set(SESSION_COOKIE_NAME, token, {
    httpOnly: true, // Prevent JavaScript access (XSS protection)
    secure: process.env.NODE_ENV === "production",
    sameSite: "lax",
    expires: expiresAt,
    path: "/",
  });
```

Cross-Site Request Forgery (CSRF) Prevention

CSRF prevention is implemented through SameSite cookies, which serve as the primary defense. In lib/auth/session.ts (lines 108 and 122), the session cookie is set with SameSite=Lax, providing CSRF protection while still allowing standard navigation.

```
cookieStore.set(SESSION_COOKIE_NAME, token, {
   httpOnly: true,
   secure: process.env.NODE_ENV === "production",
   sameSite: "lax", // CSRF protection while allowing navigation
   expires: expiresAt,
   path: "/",
 });
```

Next.js Server Actions provide several built-in protections, including Origin header validation by comparing the Origin with the Host or X-Forwarded-Host, a POST-only request method, and default same-origin enforcement that allows actions to be invoked only from the same host.

Client-side manipulation Prevention and Server-side validation

Client-side manipulation is prevented through server-side input validation using Zod schemas, serving as the primary defense. Every user input is validated on the server, regardless of client-side checks.

```
 const loginSchema = z.object({
   email: z.string().email("Invalid email address"),
   password: z.string().min(1, "Password is required"),
 });

 const result = loginSchema.safeParse(rawData);
 if (!result.success) {
   return { error: "Something went wrong" };
 }
```

This ensures that attempts to bypass validation via browser DevTools, send malformed data through tools like Postman or curl, or remove required HTML attributes are all rejected, and only valid input is accepted.

Use of Transport Layer Security (TLS)

The application is TLS-ready via environment-aware secure cookie flags, while TLS/HTTPS is correctly handled at the infrastructure or deployment level.

```
  export async function setSessionCookie(
    token: string,
    expiresAt: Date
  ): Promise<void> {
    const cookieStore = await cookies();
    cookieStore.set(SESSION_COOKIE_NAME, token, {
      httpOnly: true,
      secure: process.env.NODE_ENV === "production", // HTTPS only in
production
      sameSite: "lax",
      expires: expiresAt,
      path: "/",
    });
  }
```

Caddy automatically obtains a TLS certificate from Let's Encrypt for markindex.io, handles automatic renewal before expiration without manual intervention, redirects all HTTP traffic to HTTPS, serves the site exclusively over HTTPS, applies a modern TLS configuration with TLS 1.2+ and strong cipher suites, and enables OCSP stapling for improved certificate validation performance.

```
# CaddyFile
markindex.io {
        reverse_proxy localhost:3000
}

www.markindex.io {
        redir https://markindex.io{uri} permanent
}

markindexapp.com {
        redir https://markindex.io{uri} permanent
}

www.markindexapp.com {
        redir https://markindex.io{uri} permanent
}
```

Use of encryption / hashing

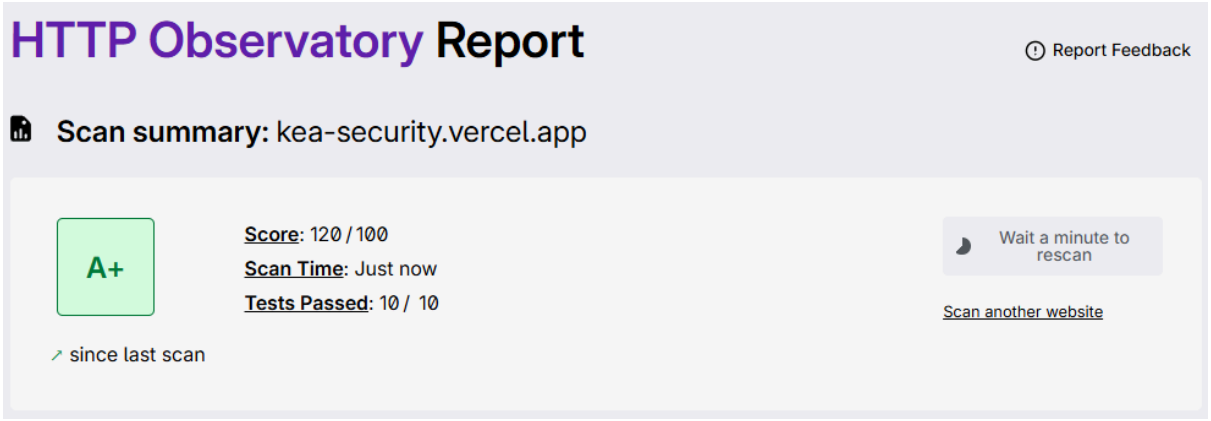
Password hashes use Argon2id, which is non-reversible and remains secure even if the database is compromised. Rainbow table attacks are blocked through unique salts for each password, brute-force cracking would take approximately 89 years for an 8-character password, and the memory-hard nature of Argon2id makes GPU or ASIC attacks prohibitively expensive.

```
import { hash, verify } from "@node-rs/argon2";

const HASH_OPTIONS = {
  memoryCost: 19456,
  timeCost: 2,
  outputLen: 32,
  parallelism: 1,
};

export async function hashPassword(password: string): Promise<string> {
  return await hash(password, HASH_OPTIONS);
}
```

Mozilla Observatory



CVE

In the following sections I would like to go through some recent CVE incidents that have affected this project and how I should tackle them.

React CVE

React is used as the core library for building and rendering the user interface in Markindex, handling both client-side interactivity and server-side rendering in conjunction with Next.js.

As it is deeply integrated into the application, it is important to monitor React for security updates and vulnerabilities, since any flaw could potentially impact the entire system.

On 3 December 2025, React disclosed a security flaw that could allow unauthenticated remote code execution by taking advantage of an issue in the way React processes payloads sent to React Server Function endpoints.

> *The vulnerable RSC protocol allowed untrusted inputs to influence server-side execution behavior. Under specific conditions, an attacker could craft requests that trigger unintended server execution paths. This can result in remote code execution in unpatched environments.*[1]

An attacker without authentication could send a specially crafted HTTP request to a Server Function endpoint, which React would deserialize in a way that allows remote code execution on the server.

The severity of this vulnerability is critical, as it allows unauthenticated attackers to execute arbitrary code on the server, potentially compromising sensitive data, server integrity, and any connected services.

To address this issue, all affected packages should be updated immediately to the fixed versions. Developers should verify that their projects are not using vulnerable React Server Component packages, clear any local caches to remove old versions, and review environment credentials to ensure they have not been exposed.

PostHog CVE

PostHog is a tool used for analytics and user behavior tracking in Markindex, helping me understand application usage and performance.

When using a tool such as PostHog, it is important to monitor both the tool itself and its dependent infrastructure, as vulnerabilities in these services can have cascading effects on your own systems.

[1] https://nextjs.org/blog/CVE-2025-66478

On 24 November 2025, PostHog announced that they had been a victim of the Shai-Hulud attack, which affected over 300 packages.

> *We've identified that several of our packages contain compromised versions. We've unpublished the affected versions for our main repo, and are published new, safe versions of the packages.*[2]

Upon execution, the malware downloads and runs TruffleHog to scan the local machine, stealing sensitive information such as NPM Tokens, AWS/GCP/Azure credentials, and environment variables.

The severity of this vulnerability is high, as it directly targets developer environments and could lead to exposure of critical secrets.

To address this issue, you should first remove any compromised package versions from your project and ensure that your local development environment no longer contains them. Clearing package caches helps prevent old, vulnerable versions from being reinstalled. After that, update to the latest, verified versions of the affected packages. Finally, review any sensitive credentials or environment variables to confirm they have not been exposed, and rotate them if there is any doubt about their integrity.

[2] https://status.posthog.com/incidents/kv3nj636f59c