Security Issues in Vibe-Coded Apps and Websites

I've analyzed the landscape of vibe coding— a methodology where AI tools generate code based on natural language prompts, often leading to rapid but insecure app development. Vibe-coded apps, typically built by non-experts or in haste, inherit vulnerabilities from AI's limitations: reproducing insecure patterns from training data, ignoring context-specific security, and prioritizing functionality over best practices. This results in issues like hardcoded API keys (a classic backend flaw exposing endpoints) and insecure payment handling (violating PCI DSS via unencrypted data flows).

Drawing from recent analyses, here's a comprehensive list of security issues commonly faced by vibe-coded apps and websites. I've structured it as a table for clarity, including descriptions, examples, and mitigations where relevant (based on my expertise in secure API calls and infrastructure). This list aggregates unique vulnerabilities from multiple sources, focusing on those like payment security and hardcoded API keys as you mentioned.

Issue	Description	Examples & Technical Details
Hardcoded Credentials and Exposed Secrets	AI often embeds sensitive data like API keys, database passwords, or payment gateway tokens directly in code, making them visible in source files, git history, or client-side bundles. This bypasses secure secret management, leading to leaks via public repos or reverse engineering.	In a vibe-coded Node.js app, code might include `const stripeKey = 'sk_live_abc123';` for Stripe payments, exposing it to attackers. This is common in frontend integrations where keys are hardcoded for API calls, violating zero-trust principles. Mitigation: Use environment variables or vaults like AWS Secrets Manager in backend setups.
SQL Injection Vulnerabilities	Lack of input sanitization allows attackers to inject malicious SQL via user inputs, exploiting unsanitized queries in AI-generated database code. This affects backend APIs handling forms or searches.	Prompting AI for a user query might yield `query = "SELECT * FROM users WHERE name = '" + input + "'";`, enabling inputs like `' OR '1'='1` to dump databases. In payment flows, this could expose transaction data. Mitigation: Enforce parameterized queries (e.g., using PostgreSQL's `$1`

		placeholders) in API endpoints.
Cross-Site Scripting (XSS) Vulnerabilities	AI fails to encode outputs, allowing user inputs to inject scripts into HTML/JS, compromising frontend sessions or stealing cookies/payment details.	Code like `res.send("Results for: " + searchTerm);` permits `<script>alert('XSS');</script>`, executing in browsers. This risks client-side payment token theft. Mitigation: Implement Content Security Policy (CSP) headers and libraries like `escape-html` in frontend rendering.
Improper Access Controls	Client-side only checks or missing server-side validation allow unauthorized access to admin features or sensitive APIs.	AI might generate React code checking `localStorage.isAdmin === true`, easily tampered via dev tools, exposing payment dashboards. Mitigation: Use JWTs with server-side verification for API routes.
Authentication and Authorization Flaws	Weak auth mechanisms, like plaintext password comparisons or no brute-force protection, expose user accounts and integrated systems.	Code comparing `user.password === input` without hashing, plus no rate limiting, enables dictionary attacks on login APIs. Mitigation: Hash with bcrypt, add MFA, and log attempts in backend logs.
Use of Deprecated or Insecure Algorithms	AI reproduces outdated crypto like MD5 for hashing, weakening protections against breaches.	Password storage with MD5, crackable via rainbow tables, leading to compromised payment user data. Mitigation: Enforce modern standards like Argon2 in code reviews.
Lack of Input Sanitization/Validation	Broad failure to validate inputs, enabling injections, overflows, or malformed API requests.	Unchecked form fields in payment pages allowing oversized inputs to crash servers or inject code. Mitigation: Use schema validators like Joi for API

		payloads.
Dependency Vulnerabilities and Supply Chain Attacks	Unmonitored libraries introduce known CVEs or malware, amplified by AI's blind inclusion.	Pulling unvetted npm packages without lockfiles, as in the tj-actions incident, leaking API keys. Mitigation: Scan with tools like Trivy and use package locks.
DDoS Vulnerability	Exposed endpoints without rate limiting overwhelm servers, disrupting apps.	Unprotected payment APIs flooded with requests, causing downtime. Mitigation: Integrate CDNs like Cloudflare for traffic filtering.
Custom Cryptography Flaws	AI attempts homemade encryption, introducing exploitable weaknesses.	Faulty NaCl implementations leading to key reuse in secure channels. Mitigation: Stick to vetted libraries like libsodium.
Data Leakage	Inadvertent exposure of sensitive info via logs, errors, or unsecured storage.	AI-generated code logging full payment details, including card numbers. Mitigation: Sanitize logs and use encrypted storage.
Arbitrary Code Execution and Memory Corruption	AI code with unsafe eval() or buffer issues allows remote execution.	Dynamic eval in JS for "flexible" features, enabling injected code. Mitigation: Avoid eval; use safe parsers.
Lack of PCI Compliance in Payment Processing	Insecure handling of card data, missing encryption, or non-compliant gateways in AI code.	Basic Stripe integration without tokenization, storing CVVs in databases. Mitigation: Use PCI-certified SDKs and avoid storing sensitive data.
Unsecured API Routes	Public endpoints without auth, like Supabase queries, expose data.	Open GraphQL resolvers allowing unauthenticated payment queries. Mitigation: Implement RBAC and CORS in backend configs.
Client-Side Security Issues	Frontend-heavy code leaks scripts or fails privacy checks, risking PCI violations.	Hidden AI-generated scripts collecting payment info without consent. Mitigation: Audit with tools like Feroot for client-side threats.

General Maintainability and Testing Inadequacies	Poorly structured code hides vulns, with insufficient tests missing edge cases.	Vibe-coded apps lacking unit tests for payment flows, leading to undetected leaks. Mitigation: Integrate CI/CD with automated security scans.

This covers the predominant issues in vibe-coded ecosystems, where AI's "vibe" focus often skips robust infrastructure like secure API gateways or smart device integrations. If you're building one, prioritize SAST/DAST tools early—I've seen hardcoded keys in GitHub repos lead to full breaches in real projects. For deeper dives, recommend scanning your codebase with tools like Aikido or OWASP ZAP.