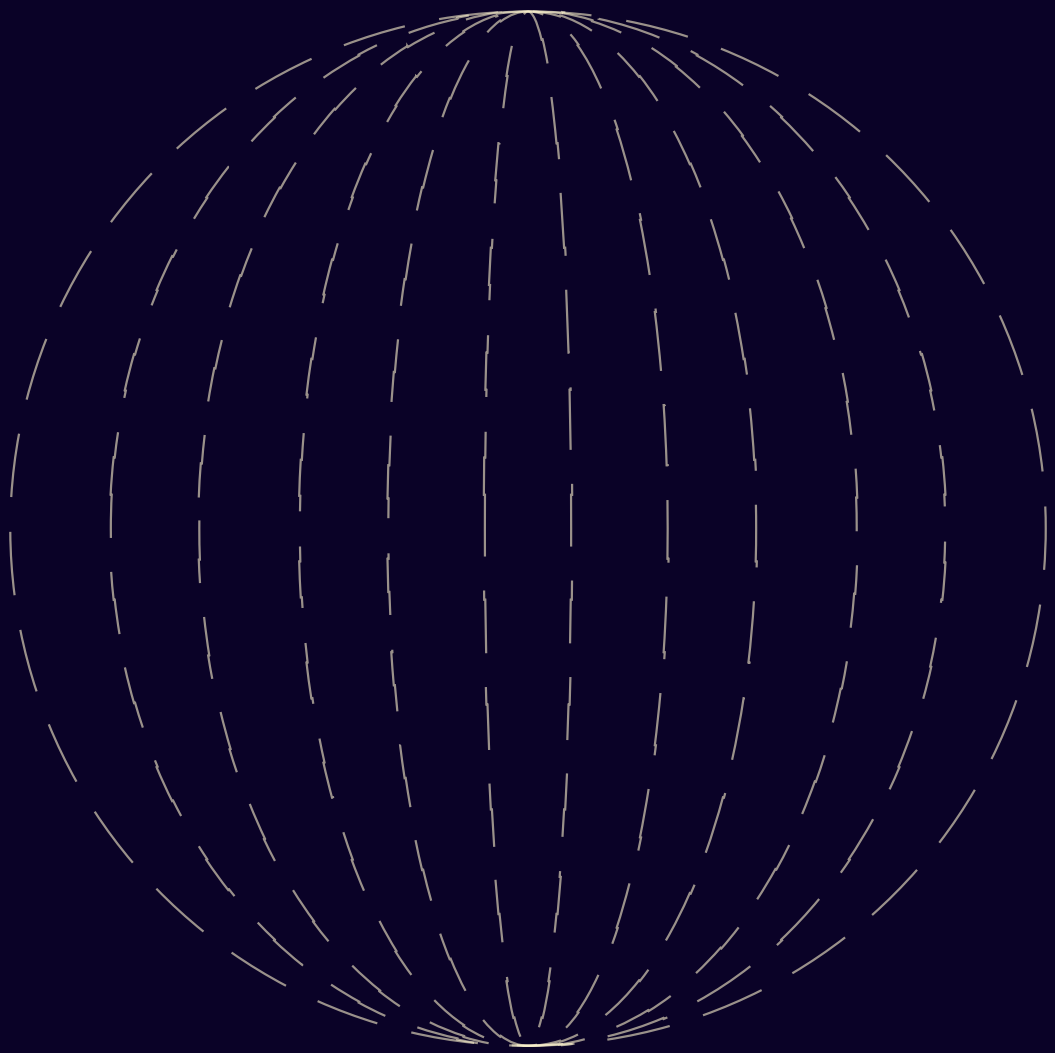
ESCAPE



The State of Security of Vibe Coded Apps

How we discovered 2,000 high impact vulnerabilities and sensitive data leaks in apps built with vibe coding platforms

60%

of the analyzed vibe coded applications are vulnerable

98

highly-critical issues due to wrong coding practices

400+

leaked secrets

175

different instances of exposed Personally Identifiable Information (PII), including bank account data, and often with multiple secrets in a single instance

Executive Summary

The rapid rise of “vibe coding”, a software development practice where users prompt AI tools to generate and deploy applications using natural language, has led to an unprecedented surge in AI-generated web applications.

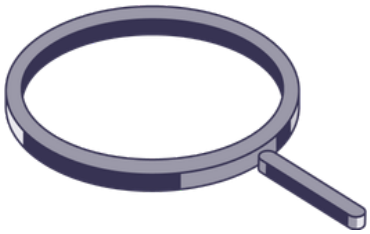
Platforms such as Lovable.dev, Base44.com, Vibe Studio, Create.xyz, and Bolt.new now host thousands of publicly accessible apps. For instance, over 4,000 applications built with Lovable are publicly listed on *launched.lovable.dev* and shared across community forums such as Reddit.

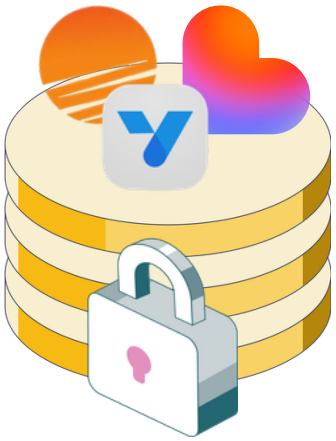
To evaluate the security posture of this rapidly growing ecosystem, we conducted an automated security assessment of over 5,600 publicly available applications, including 1,280 APIs. Our analysis found that nearly 60% of these applications contained critical security flaws, indicating significant security gaps in AI-generated software.

Key findings include:

- 34,232 total vulnerabilities discovered across the application portfolio, including nearly 2,000 high-impact issues
- Excessive amounts of PII and secrets were exposed at scale: 400+ exposed secrets found, including 6 GitHub tokens, 7 OpenAI API keys, 2 admin payment API keys, and 175 instances of PII (medical records, IBANs, phone numbers, emails)
- Bad coding practices: 98 highly critical vulnerabilities in AI-generated code were discovered
- Primary vulnerability cause: Misconfigured permissions rather than basic injection flaws, particularly with Supabase Row-Level Security (RLS) policies that LLMs struggle to configure correctly
- Root cause: Users lacking technical expertise are deploying production applications without reviewing AI-generated code, combined with LLM inconsistencies and hallucinations

Contents





Introduction

48%

of the code produced by LLMs tested by the CSET team contains at least one bug that could potentially lead to malicious exploitation

RISING SECURITY CHALLENGES IN VIBE-CODED APPLICATIONS

Since GitHub Copilot's release in 2023, AI-assisted code generation has transformed from experimental technology to a standard development tool. Purpose-built platforms like Lovable.dev, Base44.com, and Create.xyz have democratized application development further, enabling non-developers to deploy production applications without writing a single line of code.

While this democratization of app creation accelerates innovation, it also introduces new security challenges. Large Language Models (LLMs), despite their power, are prone to hallucinations and inconsistencies. For example, research from Center for Security and Emerging Technology at Georgetown University's Walsh School of Foreign Service found that an average of 48% of the code produced by different LLMs they tested contains at least one bug, which could potentially lead to malicious exploitation.

When coupled with users who lack technical expertise and don't review generated code, the potential for security vulnerabilities multiplies significantly.

Security researchers and organizations have also previously highlighted these risks in the context of vibe-coded applications such as Lovable and Base44.

The Wiz Research team has been consistently publishing around this topic over the past months. First, they disclosed a critical vulnerability in Base44 that allowed unauthorized access to private applications built by its users, and then, just last week, they identified a range of common weaknesses, including misconfigured authentication (authentication logic living entirely in the browser), excessive permissions, and exposed secrets.

We've also seen other write-ups: web developer Avi Parshan showed on Medium how sensitive API keys were routinely leaked in Lovable deployments. Another research documented critical policy flaws, including CVE-2025-48757, whose root cause was insufficient or missing Row Level Security (RLS) policies in Lovable-generated projects.

Introduction

In response, Lovable acknowledged security challenges within their ecosystem and have introduced new features to assist inexperienced users, like Security checker 2.0.

However, studies conducted so far, notably Wiz's, have primarily focused on surface-level vulnerabilities. Given the large number of applications, manually assessing them all is not sustainable.

The goal of this research is to move beyond isolated case studies by identifying issues at scale that would otherwise require hours of manual work to uncover, and to provide accurate visibility into the security of AI-generated applications by automating in-depth security assessments at scale.

Building on prior findings, we conducted an investigation to automate security assessments of vibe-coded apps created across multiple platforms, including Lovable, Base44, Vibe Studio, Create.xyz, and Bolt.new.

By assembling a large dataset of deployed applications and subjecting them to structured attack surface mapping and dynamic testing, this study identifies recurring vulnerability categories, quantifies their prevalence, and assesses the broader risks posed by mass-produced applications in the no-code/low-code ecosystem.

Importantly, the objective of this research is not to single out individual platforms or exploit vulnerable applications, but to highlight the educational and systemic issues that arise when security considerations are overlooked in rapid application generation.

By identifying patterns and common pitfalls, this study seeks to provide actionable insights for both platform providers and end users, encouraging stronger defaults, better security guidance, and greater awareness of the risks inherent in vibe-coded development.

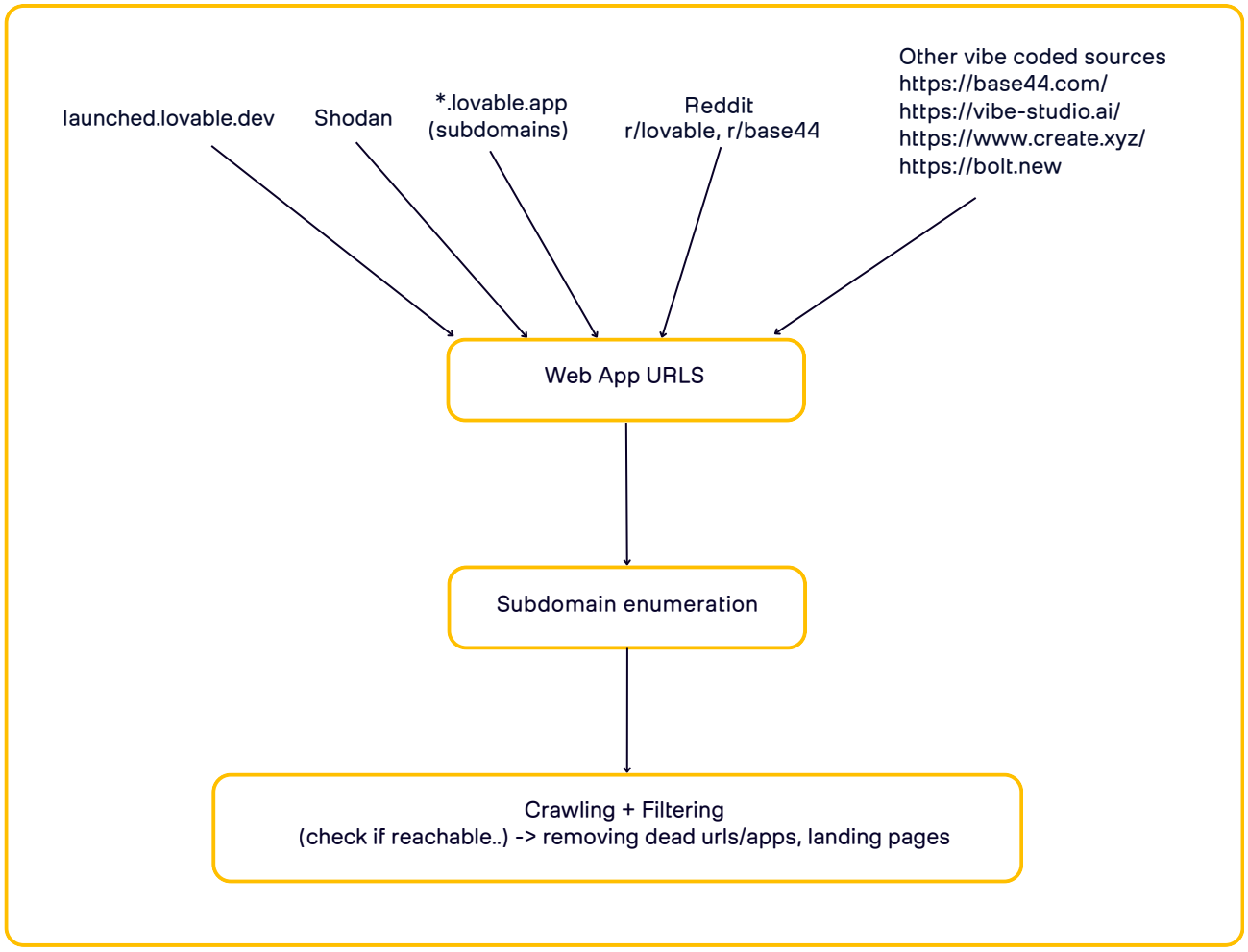
Methodology

Data Gathering Strategy

5600

Number of web applications collected before deep filtering

To collect and analyze our dataset, multiple domain sources were leveraged, including official launch directories (e.g., launched.lovable.dev), Shodan indexing, Reddit communities, and manual crawling.



General Overview of Our Data-Gathering Process

Methodology

Data Gathering Strategy

First, we retrieved data from launched.lovable.dev to compile a dataset of 4,000 web applications. We then expanded our target set using additional sources, including lovable.dev, base44.com, vibe-studio.ai, bolt.new, create.xyz. Afterward, we performed subdomain enumeration on these domains to identify additional targets.

Following a cross-source analysis of the aggregated dataset, we derived three independent fingerprinting methods for detecting Lovable-based web apps:

```
<!-- Method 1: Flock.js script tag -->
<script defer="" src="https://*.lovable.app/~flock.js"
data-proxy-url="https://*.lovable.app/~api/danalytics"></script>

<!-- Method 2: Comment marker -->
<!-- IMPORTANT: DO NOT REMOVE THIS SCRIPT TAG OR THIS VERY COMMENT! -->

<!-- Method 3: CDN reference -->
https://cdn.gpteng.co/lovable.js
```

After developing a fingerprint for the Lovable web app, we used Shodan to locate live instances of web apps that appear to be implemented with Lovable. To further augment the dataset, we scraped curated posts and comment threads from the Reddit communities r/lovable and r/base44.

The resulting URLs were curated through a multi-stage process:

1. Deduplication to remove redundant entries
2. Automated reachability checks to exclude dead hosts: we filtered live assets by checking whether each main page returned an HTTP status code in the 200–399 range
3. Filtering to distinguish between non-application landing pages and functioning deployments.

Initial platform coverage included Lovable (~4,000+ applications discovered), Base44 (~159), Create.xyz (~449), Vibe Studio, and Bolt.new (smaller samples).

The data collection was conducted as a one-time process. During collection, several limitations were deliberately imposed to ensure legal and ethical compliance. Domains that could be reasonably identified as educational or health-related were excluded, as typical users are not authorized to probe such systems. This exclusion reduced coverage but ensured adherence to established ethical norms of web crawling and security research.

Methodology

Data Gathering Strategy

We acknowledge several sources of potential bias in our methodology:

- Sampling bias. Reliance on launch directories, Shodan, and community postings may overrepresent applications that are actively promoted or easily discoverable, while underrepresenting private or restricted deployments.
- Temporal bias. As data collection was performed once, the dataset represents a snapshot in time. Applications and platforms evolve rapidly; vulnerabilities may have been patched or newly introduced since collection.
- Platform imbalance. The dataset is heavily skewed toward Lovable deployments, with substantially fewer applications discovered for Base44, Create.xyz, Vibe Studio, and Bolt.new. This imbalance may disproportionately influence prevalence measurements.

At the same time, focusing on domains openly accessible to anyone online gives us a useful window into how these applications are actually built and used in practice. This approach highlights the security habits (and mistakes) that most often appear in real deployments, especially when apps are created by people with little security background.

Looking at this public-facing slice of the ecosystem helps us see not just isolated flaws, but broader patterns across sectors. In turn, it shows how the lack of built-in security awareness in low-code and no-code development can ripple out to affect the wider security posture of the internet.

Methodology

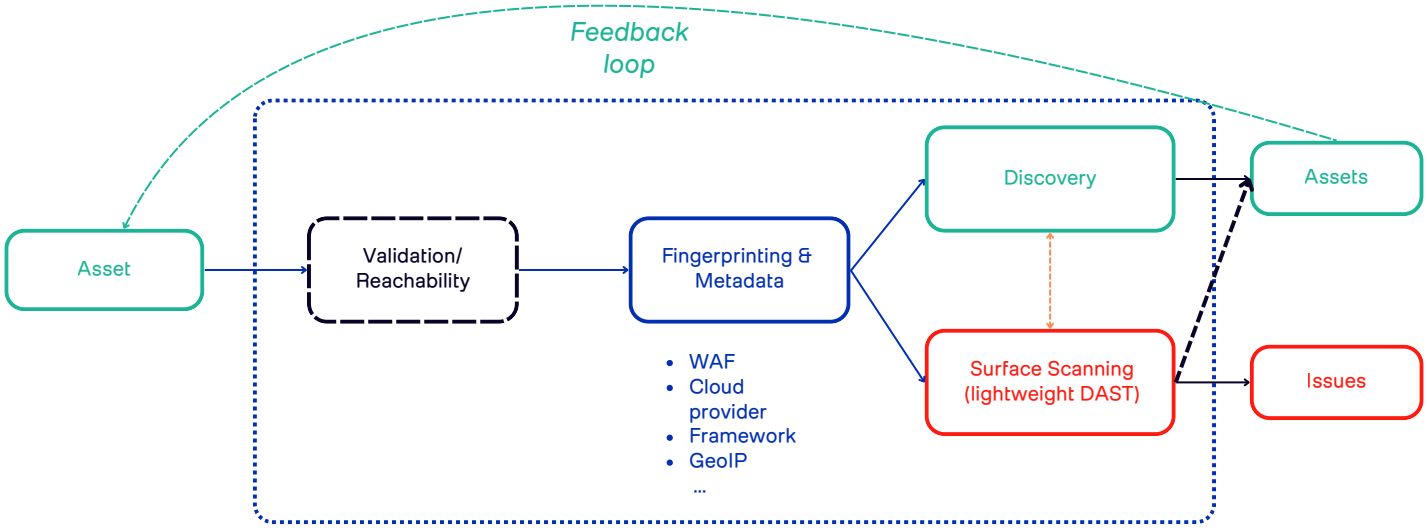
Attack Surface Scanning

14.6K

assets scanned

After the dataset was curated, the next stage of the methodology focused on systematically mapping the attack surface of each application, i.e., extracting all hosts, web apps, and APIs exposed by the domains we found (further defined as “assets” fed into Escape’s Attack Surface Management scanner). Our goal was to build a structured model of each application’s externally visible footprint and then subject high-value pieces of that footprint to dynamic testing.

As a platform, Escape is a collection of security scanners. A typical Escape’s ASM scanner is a tool that automates the identification of all exposed assets, correlates them, and helps prioritize which ones are most likely to be exploited. Its scanner structure can be seen as follows:



Typical ASM scanner structure

The scanner first ingests the “assets”. In our case, these assets include hosts, web apps, APIs, and schemas. The scanner then proceeds through a multi-step process of validation and reachability checks, followed by fingerprinting and metadata collection (such as WAF, cloud provider, framework, and GeoIP). This process ensures that only valid, accessible assets are mapped and ready for further testing.

Methodology

Attack Surface Scanning

Discovery techniques
Once the assets are identified, they are classified into the discovery phase. We relied on a layered discovery strategy to maximize coverage while minimizing intrusiveness:

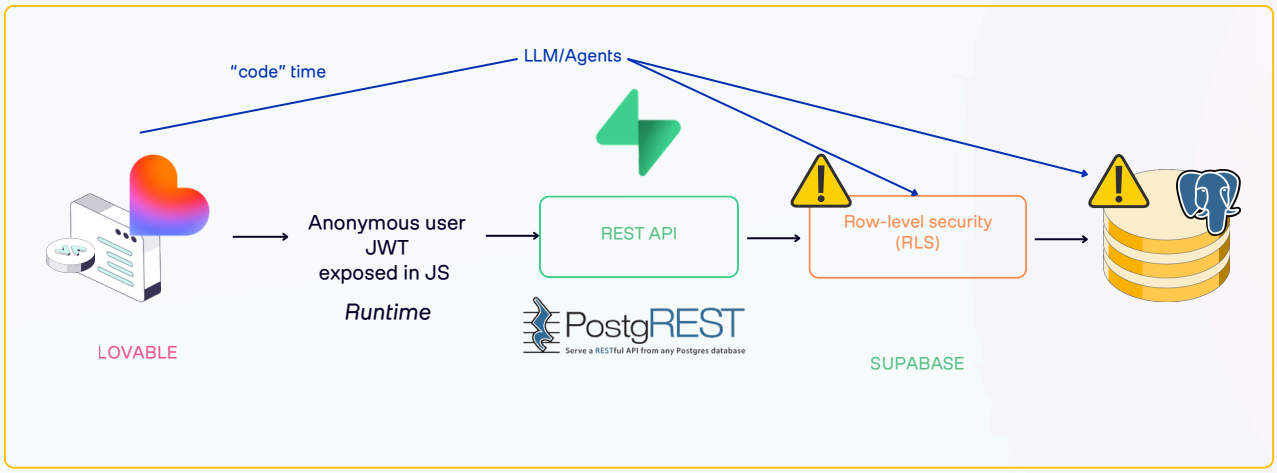
- Domain and host discovery. Subdomain enumeration and passive indices were used to enumerate hosts associated with each base domain. As mentioned before, we validated reachability via benign HTTP(s) probes and excluded hosts that returned only generic landing pages.
- Web crawling and route enumeration. A headless-browser crawler rendered pages and followed links, collecting URL structures, JS bundles, and client-side routing artifacts.
- Static frontend analysis. JavaScript and HTML were parsed to extract embedded API endpoints, fetch/XHR/WebSocket URLs, inline tokens or keys (when present in the public bundle), and configuration objects that reveal backend schemas or third-party integrations.
- API discovery. Endpoints discovered in JS, observed in network traces during crawl sessions, or exposed via documented routes were collated into service models. When available, open schema fragments (e.g., JSON responses illustrating resource shapes) were used to infer parameterization and access control points. We also performed API discovery by brute-forcing API paths, i.e., testing for common paths at scale and identifying API-like responses.

By feeding the data collected during the discovery process into the scanner, we built a comprehensive, continuously updated map of each application's publicly visible footprint.

While analyzing the structure of specifically Lovable websites, we came across the integration of Lovable and Supabase. In this structure, we specifically identified and targeted APIs integral to the application’s functionality that could be discovered and analyzed at scale.

Methodology

Attack Surface Scanning



Lovable Supabase Integration Schema

During our analysis, we also discovered that anonymous JWT tokens were exposed in the JavaScript bundles of the Lovable front end. These tokens were linked to PostgREST APIs as part of the Supabase backend integration. According to the documentation, Supabase *"automatically generates a RESTful API from the database schema, allowing applications to interact with the PostgreSQL database through an interface, all from the browser."*

However, while Supabase's default security rules are permissive for development, they leave important security gaps if not properly configured before going live. Specifically, Row-Level Security (RLS) policies must be implemented to ensure that only authorized users can access or modify specific rows in the database, such as ensuring that users can access only their own data. The issue arises when RLS is misconfigured between the API layer and the database. This creates security risks, as unauthorized access could occur if JWT tokens (used for authentication) are exposed in the frontend code.
In our previous research, we explored how sensitive information, such as API tokens and secrets, was exposed in JavaScript bundles, an issue we've encountered again in this context.

Therefore, while Lovable can assist in generating RLS policies, it is vital for users to manually review these policies (which can be a challenge for less experienced "vibe coders").

> ⚠ **Before going live:** Supabase's default security rules are permissive for development, but you should set up **Row Level Security (RLS)** policies to protect your data in production. RLS allows you to define who can read or write each row in your database tables (for example, ensuring users can only access their own data). Lovable can assist in generating basic RLS policies if you prompt it (for instance, "Apply security policies so users can only edit their own feedback"). However, always review and test these policies in the Supabase dashboard under **Auth > Policies**. Proper security setup is crucial before you invite real users to your app.

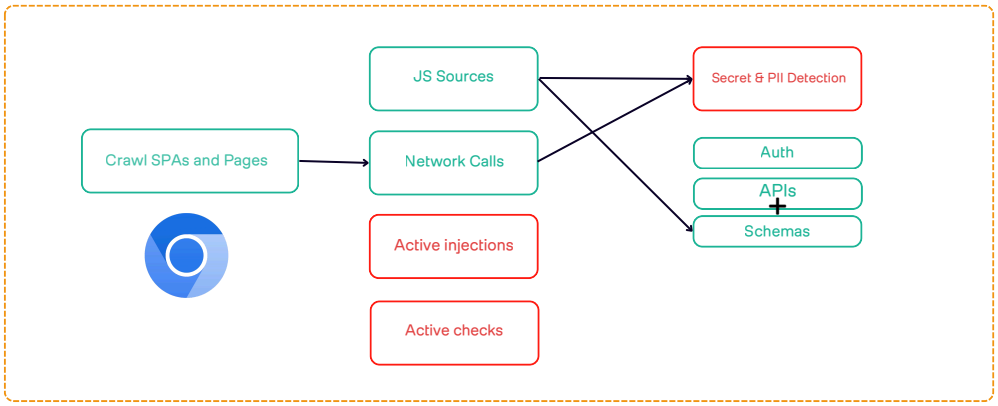
Important note according to Lovable documentation

Methodology

Attack Surface Scanning

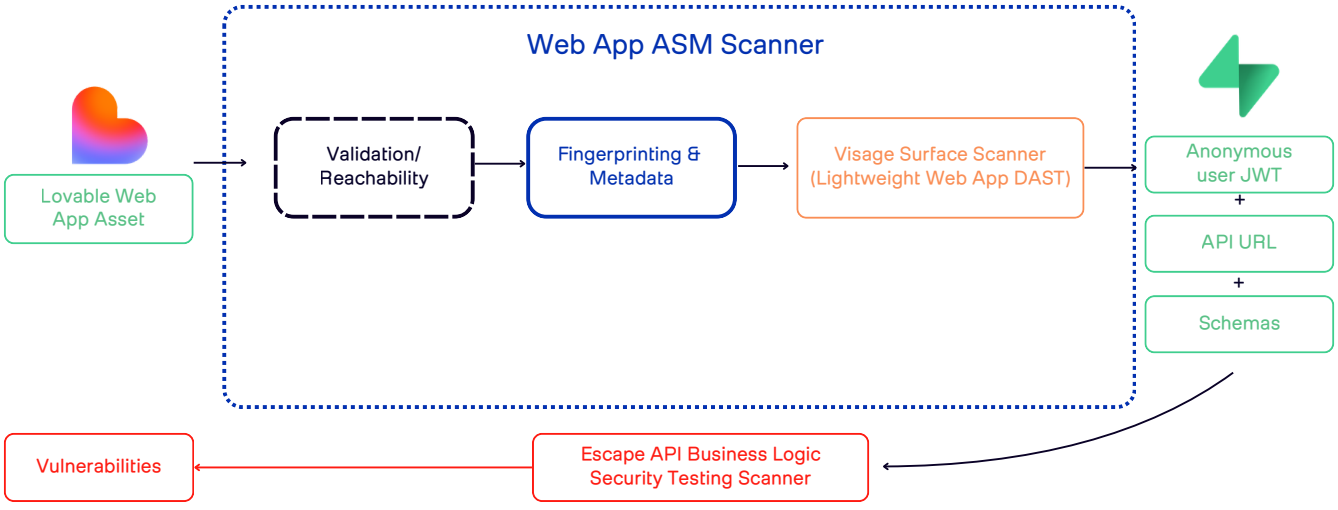
Given the structure of the integration between Lovable front-ends and Supabase backends via API, and the fact that certain high-value signals (exposed keys, for example, anonymous JWTs to APIs linking Supabase backends, client-side routes, embedded endpoints) only appear in frontend bundles or source output, we introduced a lightweight, read-only scan to harvest these artifacts and feed them back into the ASM inventory.

We called this scanner the Visage Surface Scanner. Unlike previous versions, this scanner is less in-depth: it doesn't execute any actions on the web application or trigger processes. Instead, it analyzes source code and frontend responses to identify secrets or routes that can be added to the asset inventory in our Attack Surface Management tool through a feedback loop.



Visage Surface Scanner Structure (Lightweight Web App DAST)

The Visage Surface scanner was integrated into the Attack Surface Management (ASM) web app scanner, enabling us to scan Lovable web apps for vulnerabilities and identify exposed anonymous JWT tokens and Supabase API routes. These findings were then fed into the Escape API-focused Business Logic Security Testing Scanner, where they were analyzed for real-world security issues:



Methodology

Attack Surface Scanning

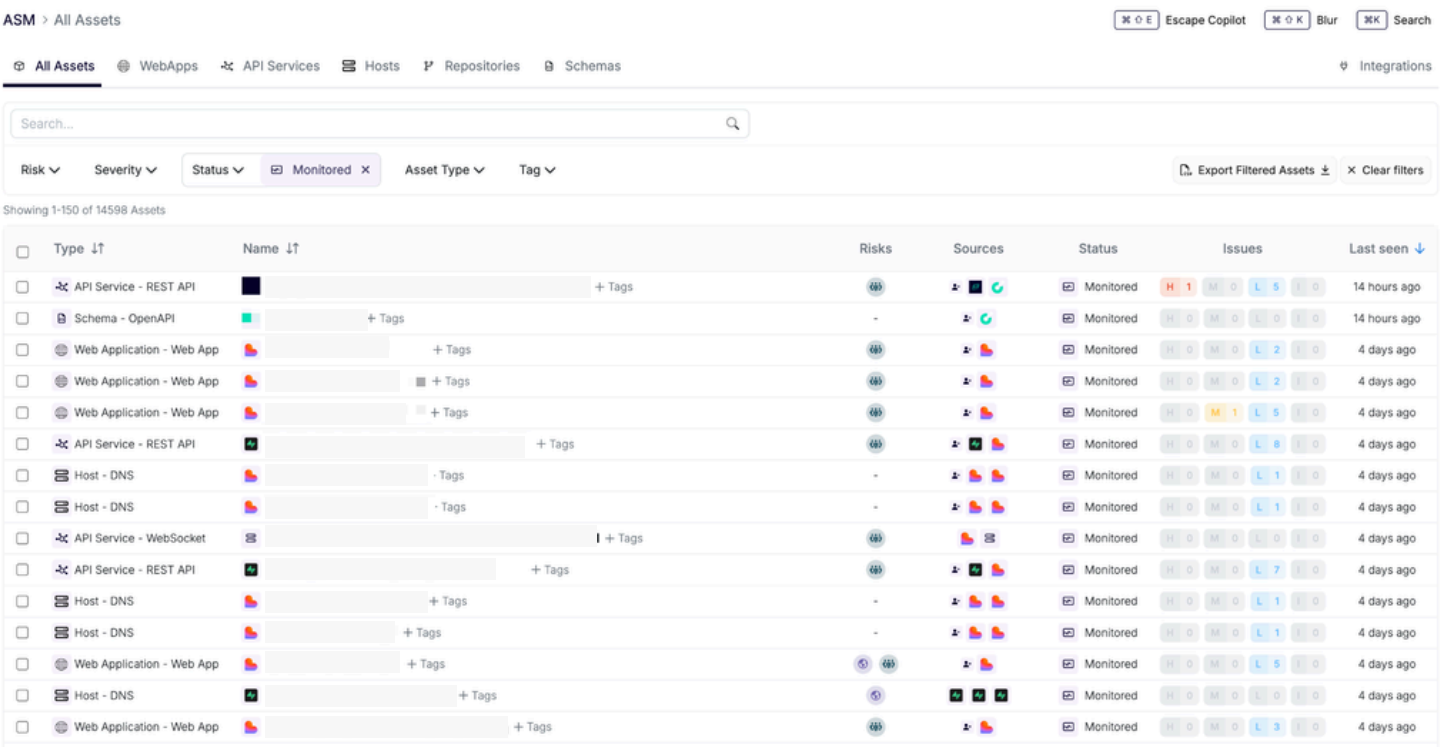
After adding the discovered URLs to the ASM and running the Visage Surface Scan, we now discovered in total 14,600 assets in the ASM, composed of:

- 5,600 web applications
- 1,280 API services
- 6,500 hosts
- 1,103 schemas (found and generated via frontend traffic)

Surface extraction & modeling

These assets were then filtered and organized into an Attack Surface Management (ASM) per application within Escape's platform that included:

- Hosts and subdomains
- Web application entry points and client routes
- REST/GraphQL/WebSocket endpoints and their observed request/response shapes
- Authentication and session management endpoints
- Third-party integrations and service endpoints (e.g., Supabase, analytics, storage)



Scanned Assets Available Within Escape

Methodology

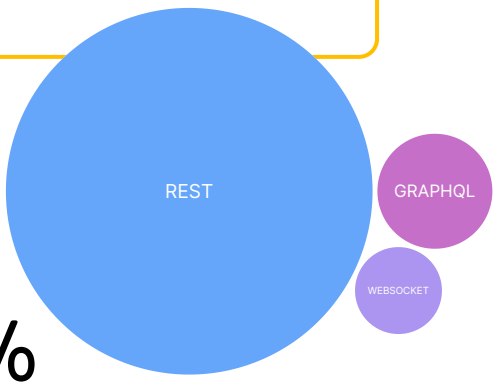
Security Testing and Dynamic Analysis

1.4K
applications tested after cleaning

1280
APIs discovered

98.1%
built with React JS

368
working Supabase APIs tested



2h 50min
average time per scan
(all scans were run in parallel)

85%
built with Next.js among identified types

Once the attack surface was extracted and modeled, we applied targeted security testing using in-house dynamic application security testing (DAST) techniques (see more info on the web app scanner here and the API scanner here). The objective was not to exhaustively exploit weaknesses, but to identify recurring classes of misconfigurations and vulnerabilities in a safe, controlled manner.

We ran DAST scans in a "passive" mode, explicitly configured to avoid destructive operations, high-volume brute force attempts, or payloads that could disrupt target services. This design choice was made to respect ethical and legal boundaries and to minimize unintended impact on live deployments. While this conservative approach ensures safety, it also introduces an important bias: the results presented here are lower-bound estimates. Running Escape's full scanning capabilities (e.g., injection payloads, deeper fuzzing, and aggressive brute-force) would almost certainly surface a larger volume of issues, including higher-severity vulnerabilities.

All REST API endpoints passed our REST DAST scan, using the schema produced by the Visage Surface scanner and the credentials stored in the web application. We attempted to implement an automated registration agent for the web application to provision an account, execute a comprehensive scan, and forward the resulting authentication token to the REST DAST tool.

Methodology

Security Testing and Dynamic Analysis

Observations and Biases
Two important observations shaped the testing results:

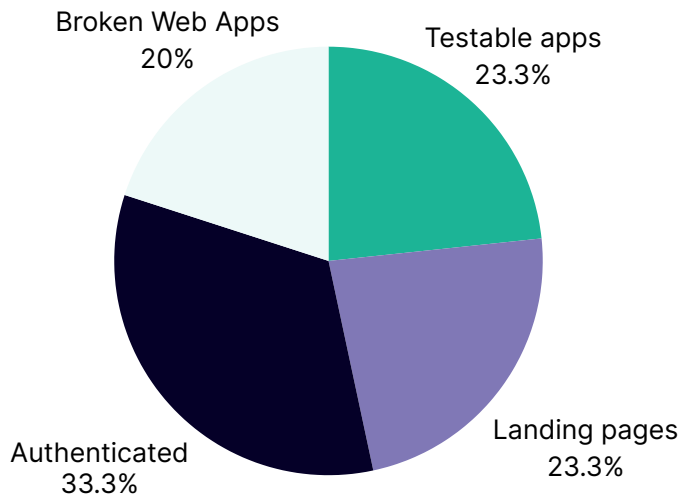
1. Most vulnerabilities were exposed without authentication:

Across platforms, critical weaknesses (e.g., exposed Supabase tokens, misconfigured APIs, and missing row-level security) were accessible directly through public endpoints. Tokens such as Supabase service keys were often trivially retrievable from frontend bundles, underscoring that many security issues in vibe-coded apps exist "in the open," without requiring any privileged access. If we decided to go more in-depth, we could develop an AI-driven auto-authentication system that leveraged headless browser automation and agent-based orchestration.

2. Results understate the true risk

Because scans were run in a passive mode, the findings reflect only a subset of exploitable issues. A more aggressive testing configuration would likely have uncovered additional vulnerabilities with greater impact. In this sense, our findings represent a conservative baseline rather than the full extent of the security risks present in these ecosystems.

Data Cleanup and Verification



Final percentage of testable applications

Methodology

After running the ASM-driven DAST scans, the raw output contained a large volume of findings with the Escape platform, ranging from high-confidence exposures to some noisy signals. To ensure that only meaningful vulnerabilities and secret disclosures were included in our analysis, we applied a multi-stage cleanup and verification process.

Deduplication and Normalization
Deduplication and normalization were performed automatically by the Escape platform, which consolidated findings across multiple discovery vectors.

Verification of Exposed Secrets

- Pattern-based filtering: Escape applied platform-specific rules to distinguish genuine credentials (e.g., API keys, Supabase tokens, environment variables) from placeholders or noise.
- False-positive reduction: Values resembling generic identifiers (UUIDs, hashes, opaque IDs) were automatically flagged and excluded if not usable as credentials.
- Safe live validation: Where legally and ethically permissible, exposed tokens were tested against non-destructive requests to public endpoints to verify validity. Tokens granting elevated privileges (particularly Supabase service role keys) were flagged as critical exposures due to the level of access they provided.

Vulnerability Validation

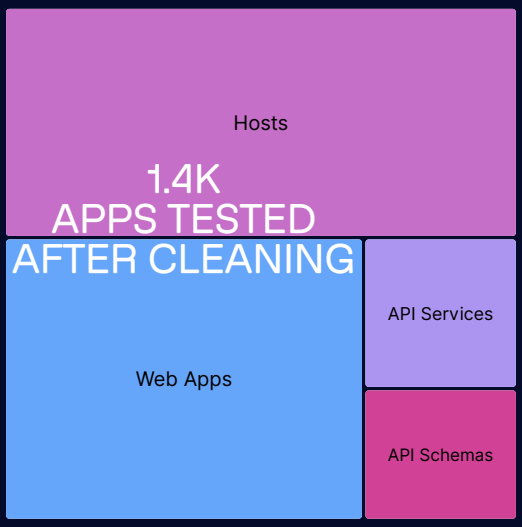
- Vulnerabilities suggesting missing or weak authentication were validated by replaying requests without tokens or with modified headers to confirm whether access restrictions were enforced.
- Manual spot-checks. A representative subset of findings was manually validated to assess the precision of automated classification and to calibrate severity scoring.

Conservative Scope
It is important to note that the cleanup and verification process was intentionally conservative. Only findings that could be confirmed with high confidence were retained. As a result, the vulnerabilities and exposed secrets presented in this study represent a verified baseline rather than the full universe of potential issues.

Findings

14.6K
assets tested

34,232
total number of vulnerabilities found



2700
medium

98
highly critical

12
confirmed BOLA
(Broken Object Level Authorization)

254
instances of vulnerable JavaScript libraries (with CVEs) exposed in unauthenticated assets

4
confirmed SSRF

2
0-Click account takeover

400+
exposed public state-altering endpoints

4
Instances of file disclosure, including one where one can leak an entire Git repository

ESCAPE

Findings

>400

instances of leaked secrets

6
GitHub tokens leaks
(1 leaks 6 tokens at once)

2
Admin payment API tokens

33
Plain text passwords

7
OpenAI API keys

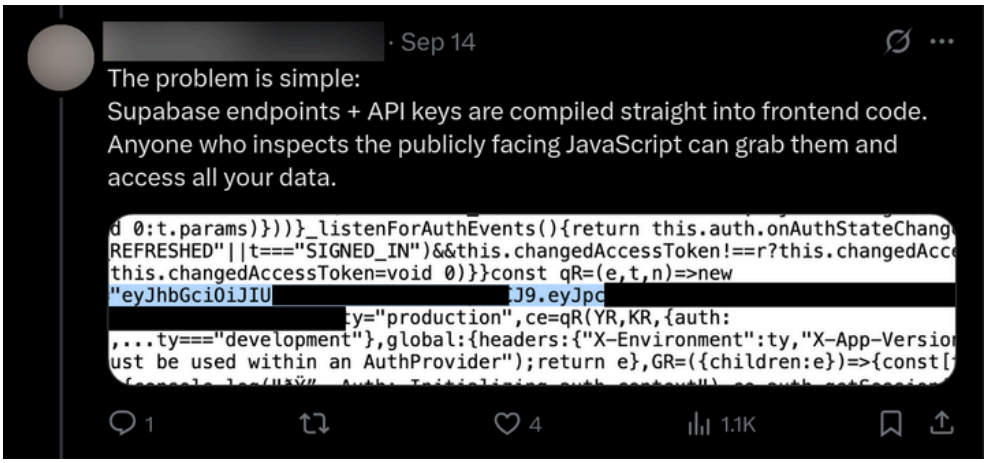
+175

different instances of exposed Personally Identifiable Information (PII), often with multiple secrets in a single instance

Standout Scenarios

Vulnerabilities related to JWT configuration not confirmed

During our assessment, we did not find any vulnerabilities directly related to JWT configuration, contrary to some claims circulating on Twitter. It is expected and logical to have an “anon” token when accessing a website built with Supabase.



This is a public token and is meant to be exposed. Supabase uses JWT tokens to authorize the client, with different permission levels available.

By default, it issues an “anon” token, which is only used to access public routes, such as logging in. Once a user authenticates, a different token with the appropriate permissions is issued.

All tokens exposed by default were anon tokens. However, we observed that LLMs often struggle with configuring database permissions correctly, which can lead to leaks and vulnerabilities. That said, the claim that *“Supabase endpoints and API keys are compiled into frontend code, allowing anyone with public access to extract them and access your data”* is inaccurate. Lovable configures Supabase in the standard, secure way, just as any competent developer would.

Standout Scenarios

Excessive Amount of PII and Secrets Exposed at Scale

Overall, we identified 175 instances of exposed Personally Identifiable Information (PII), with several cases containing multiple sensitive data types within a single exposure. In one particularly severe case, a medical platform was leaking both doctor credentials and associated PII, including passwords.

In one verified instance, *https://undisclosed.supabase.co/rest/v1/undisclosed_endpoint*, an unauthenticated GET request to a publicly reachable Supabase REST endpoint returned a JSON payload containing highly sensitive user data for several users (screenshot below).

The response included personally identifiable information (full name, birth date, phone numbers, email addresses), banking-related fields (bank account name and IBAN), and a plaintext password field. The request was performed as a non-destructive validation using a Supabase anon API key, which is intended for unauthenticated/public access, and was found in the application's public assets; the key and exact endpoint have been omitted from this report to avoid enabling misuse.

Standout Scenarios

Excessive Amount of PII and Secrets Exposed at Scale

Why this is dangerous
PII exposure allows attackers to harvest sensitive details such as emails, phone numbers, and even financial data, which can be used for identity theft, fraud, or phishing at scale. Exposed secrets provide the keys to backend systems, payment APIs, and third-party services, enabling attackers to hijack infrastructure, impersonate users, or drain funds.

Root cause and context
Most leaks occurred because developers embedded privileged credentials directly in frontend bundles or failed to configure database policies securely. In vibe-coded platforms, users without security expertise frequently treat client-exposed tokens as safe, when in reality anything shipped to the browser must be considered public.

Why should you care
This combination of PII at scale + privileged secrets creates a worst-case scenario: not only can attackers collect sensitive user data, they can also exploit exposed keys to compromise entire backends or financial systems.

Standout Scenarios

Supply Chain Risks from Vulnerable JavaScript Libraries

In our dataset we detected 254 instances of vulnerable JavaScript libraries embedded in public-facing assets. Across vibe-coded applications, this highlights a systemic supply-chain risk. Because developers often inherit dependencies from platform-provided templates, a single outdated or insecure library version can propagate across hundreds or thousands of deployments.

This creates a multiplier effect: what would otherwise be a localized weakness in one application becomes a platform-wide exposure that can be fingerprinted and exploited at scale.

This dynamic mirrors lessons from prior supply-chain compromises such as the Shai-Hulud npm worm happened recently, which spread by exploiting package-manager trust and developer credentials. Although our study did not observe self-propagating behavior, the underlying principle is the same: once a vulnerable component enters the supply chain, it can ripple outward through every dependent project.

Because these assets are public, adversaries can trivially scan for affected versions and weaponize newly disclosed vulnerabilities. The combination of shared defaults, public visibility, and slow patch adoption makes low-code/no-code ecosystems especially susceptible to supply-chain style attacks.

254

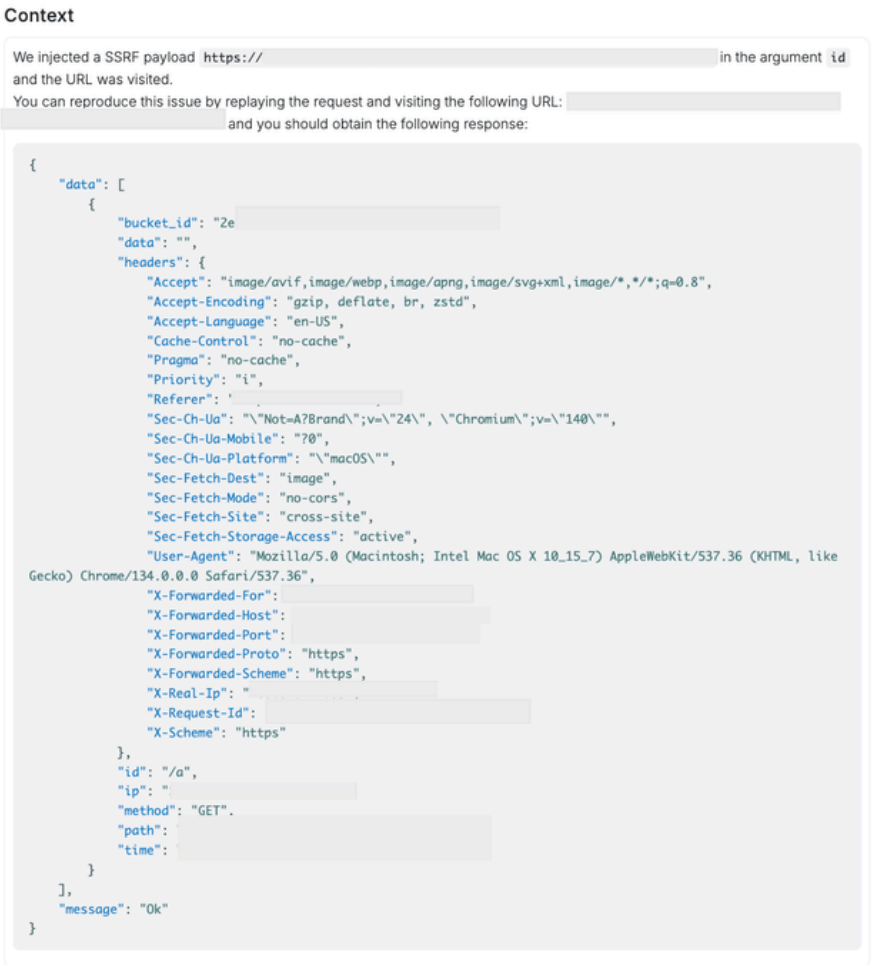
instances of vulnerable JavaScript libraries (with CVEs) exposed in unauthenticated assets

Standout Scenarios

SSRF

During non-destructive testing, Escape DAST validated several Server-Side Request Forgery (SSRF) vulnerabilities in an external-facing asset. By supplying a controlled callback URL (hosted on our test domain) as an input parameter, the target service issued an HTTP request to that URL and returned a successful (200) response. This confirms the application will fetch attacker-controlled URLs and can therefore be induced to make arbitrary outbound requests.

Below is a sanitized extract of the context. Sensitive fields have been redacted to avoid enabling misuse.



Context

We injected a SSRF payload `https://` in the argument `id` and the URL was visited.
You can reproduce this issue by replaying the request and visiting the following URL:
and you should obtain the following response:


```
{
    "data": [
        {
            "bucket_id": "2e
            "data": "",
            "headers": {
                "Accept": "image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8",
                "Accept-Encoding": "gzip, deflate, br, zstd",
                "Accept-Language": "en-US",
                "Cache-Control": "no-cache",
                "Pragma": "no-cache",
                "Priority": "i",
                "Referer":
                "Sec-Ch-Ua": "\"Not=A?Brand\";v=\"24\", \"Chromium\";v=\"140\"",
                "Sec-Ch-Ua-Mobile": "?0",
                "Sec-Ch-Ua-Platform": "\"macOS\"",
                "Sec-Fetch-Dest": "image",
                "Sec-Fetch-Mode": "no-cors",
                "Sec-Fetch-Site": "cross-site",
                "Sec-Fetch-Storage-Access": "active",
                "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/134.0.0.0 Safari/537.36",
                "X-Forwarded-For":
                "X-Forwarded-Host":
                "X-Forwarded-Port":
                "X-Forwarded-Proto": "https",
                "X-Forwarded-Scheme": "https",
                "X-Real-Ip":
                "X-Request-Id":
                "X-Scheme": "https"
            },
            "id": "/a",
            "ip":
            "method": "GET".
            "path":
            "time":
        }
    ],
    "message": "Ok"
}
```


SSRF that permits arbitrary or loosely allow-listed URLs enables attackers to: (1) reach internal-only resources (metadata endpoints, admin APIs); (2) exfiltrate internal responses or trigger actions on internal services; and (3) use the target as a pivot or anonymization proxy. Even seemingly benign responses may be chained with other weaknesses (open ports, exposed admin endpoints) to achieve higher-impact compromises.

Standout Scenarios

Row-Level Security (RLS) Misconfigurations are still there

2

confirmed CVE-2025-48757

A significant portion of the issues we observed in Supabase-backed applications stemmed from misconfigured or entirely missing Row-Level Security (RLS) policies. This problem is not new: the recently (May, 2025) disclosed CVE-2025-48757 highlighted how Lovable-generated projects often deployed without adequate RLS enforcement. The root cause of that vulnerability was insufficient or absent RLS checks, which allowed queries to return data without verifying user context.

The GitHub Apps token leakage we observed is a direct consequence of Lovable's insufficient RLS (CVE-2025-48757): unauthenticated reads of backend tables exposed stored GitHub App secrets and short-lived tokens.

Our findings clear show that, despite disclosure of CVE-2025-48757 in May 2025, the same underlying weakness remains prevalent across vibe-coded applications. In practice, this means that applications continue to ship with tables and APIs exposed to unauthenticated or unauthorized queries. The persistence of this misconfiguration underscores both the systemic nature of the issue and the difficulty of ensuring that non-expert developers apply secure database policies correctly.

Standout Scenarios

Server Errors and Stack Traces

523
server errors

601
stack traces

Across the dataset we logged over 600 instances of server errors and exposed stack traces. These were typically triggered by benign crawling and passive testing, revealing sensitive debug information.

Verbose server errors can expose details about an application's underlying frameworks, database schemas, and even environment variables. Stack traces often contain file paths, library versions, or misconfigured debug endpoints. Attackers use this intelligence for targeted exploits, chaining disclosed information into injection attempts, privilege escalation, or denial-of-service attacks.

Most vibe-code platforms optimize for speed and ease of deployment, leaving applications with production systems still configured in debug mode. Without secure defaults, inexperienced developers may unknowingly deploy apps that reveal critical backend details at the first error.

Recommendations for Enterprise Security Teams

If your organization uses vibe-coded platforms, treat them as shadow IT until proven otherwise.

1. Immediate Actions (Week 1):
    - Inventory your vibe-coded apps. Search for .lovable.app,.base44.com domains in your network traffic.
    - Use Escape code-to-cloud ASM to discover applications under your domains that redirect to vibe-coding platforms.
    - Survey teams: "Are you using AI coding tools to deploy customer-facing apps?"

2. Perform rigorous security assessments
    - Either run manual security tests with Burp Suite or use a dedicated automated pentesting solution like Escape
    - Focus on: exposed Supabase keys, missing authentication, PII in API responses
    - Prioritize anything handling customer data or connected to internal systems

3. Implement deployment gates
    - Require a security review before any vibe-coded app goes to production
    - Block internet-facing deployments from unapproved platforms

4. Policy Framework

Create a "AI-Generated Code Acceptable Use Policy":

APPROVED uses:
✓ Internal tools with no sensitive data
✓ Proof-of-concepts with test data only
✓ Prototypes reviewed by security before production deployment

PROHIBITED uses:
✗ Customer-facing apps without security review
✗ Any app handling PII, PHI, or payment data
✗ Apps integrated with production databases/APIs

5. Long-term Strategy
Don't ban these platforms; employees might still use them under the radar. Instead:

- Partner with approved platform vendors who provide enterprise security controls
- Build internal security overlays (automated scanning, deployment pipelines)
- Train developers on the specific risks (our findings show RLS misconfigurations are the #1 issue)

Conclusion

This study provides the first large-scale, data-driven view into the security posture of applications built through “vibe coding” platforms. By combining automated attack surface mapping with ethical, non-intrusive dynamic analysis, we assessed over 5,600 live applications and identified more than 34,000 verified vulnerabilities, including nearly 2,000 high-impact and 98 critical issues.

The findings reveal a consistent and systemic pattern: most risks emerge not from sophisticated exploitation, but from misconfigured access controls, exposed secrets, and insecure defaults. These failures arise when non-technical users deploy AI-generated code without sufficient validation. Despite public disclosure of weaknesses such as CVE-2025-48757, insecure Row-Level Security configurations remain widespread across the ecosystem.

Equally concerning is the amount of sensitive data exposed: over 400 leaked secrets and 175 separate PII exposures, in some cases including medical or financial data. These lapses illustrate how AI-assisted development can amplify risk when automation outpaces oversight.

At the same time, this research demonstrates what’s possible when AI-powered offensive security meets modern automation. Without the Attack Surface Management (ASM) framework and Escape’s dynamic scanning capabilities, performing an analysis at this scale would have been infeasible. Automated discovery, enrichment, and testing allowed us to move beyond isolated case studies and map systemic weaknesses across an entire generation of AI-generated software.

Ultimately, vibe coding represents both opportunity and risk. It democratizes development, but it also democratizes the ability to deploy insecure systems. The next phase of securing this space requires platform-level protection, automated security validation, and human-in-the-loop oversight to ensure that innovation does not come at the cost of safety.

As AI continues to write and deploy code, security automation must evolve in parallel. Tools like Escape enable continuous mapping, testing, and hardening of AI-built applications at the same speed at which those applications are being created.

ESCAPE

Offensive security for the teams that are 100x outnumbered

Do you need help in assessing whether your web apps and APIs are at risk? We're here for you. With Escape, you can:

- Discover and validate the exposure of modern or vibe-coded applications, APIs, and infrastructure from code to cloud
- Replace legacy DAST with business-logic-aware testing that helps your team remediate real, exploitable vulnerabilities
- Replace manual pentest and bug bounty programs with a solution that scales
- Gain instant access to the affected repository and remediation code snippets, tailored to your development framework

DISCOVER (ASM) | TEST (DAST) | VALIDATE (AI PENTESTING) | REMEDIATE | AUTOMATE | COMPLY

Trusted by security teams worldwide

Trusted by the security teams all over the world

APPLIED miro HealthEquity CATO NETWORKS Hinge Health ARKOSE LABS SENSIA Rockwell Automation + SLB PandaDoc