

Research Article

Secrets That Survive Everything: Runtime Credential Exposure in Production Web Applications

Hemanth Kumar Gorijala¹

1. Independent researcher

Pre-deployment secret scanning controls operate exclusively on source code and never examine what a production application serves. We document two end-to-end exploitation chains in which Azure AD client credentials and APIM subscription keys extracted from production JavaScript bundles enabled full account takeover and mass user data exposure. An authorized security engagement covered approximately 2,000 enterprise web application assets across one organization. Of these, 113 (5.65%, 95% Clopper-Pearson exact CI: 4.7%–6.8%) contained live credentials in served content, and 63 of those 113 (55.8%) contained a complete Azure AD credential set sufficient for the full chain. To quantify the shift-right tooling gap, we constructed an independent Ground Truth (GT-194) of 194 unique secret-grade credentials extracted from production JavaScript bundles across 113 enterprise applications by Claude Opus 4.7 (Anthropic) and independently validated by GPT-5.5 (OpenAI) with Brennan-Prediger inter-rater agreement $\kappa = 0.822$. Ten scanners were evaluated against the locked benchmark. The principal finding is structural rather than tool-specific. 13.9% of GT-194 (27 of 194) is missed by every scanner evaluated, a tool-agnostic blind spot dominated by the CryptoJS encrypted-configuration construct that pattern-based detection cannot reach because the credential exists only after decryption with a co-located key. Combined coverage of all ten scanners plateaus at 86.1%. The best static scanner recovers 36.6% of GT-194 and the best runtime-aware scanner recovers 77.8% (F1 = 0.818, McNemar $p < 0.001$ against every static scanner), narrowing the gap to the LLM-oracle ceiling (85.6%, F1 = 0.851) to 7.8 percentage points. On 63 of 86 secret-exposed applications (73.3%), the full Azure AD token-mint chain (client_secret, App_ID, tenant_id, resource URI) is co-located in the same bundles, so the entire authentication chain is reachable from browser-visible code with no additional reconnaissance. We characterize five structural paths by which credentials reach production undetected, present a layered runtime detection methodology, and provide a prioritized remediation

framework. Recall figures are scoped to a single-organization Azure-heavy corpus rather than cross-industry estimates (Section 10).

Correspondence: papers@team.qeios.com — Qeios will forward to the authors

1. Introduction

Despite mature shift-left tooling for secret detection ^{[1][2]}, live credentials continue to appear in the production-served content of enterprise web applications. Static secret scanners (GitLeaks, TruffleHog, GitHub Advanced Security) and SAST tools (Semgrep, SonarQube, Checkmarx) operate on source code repositories and build configurations and do not examine what a deployed application serves to its users. Credentials that reach production through build-time substitution, CI/CD pipeline variable injection, runtime-fetched configuration, third-party script inclusion, or deliberate scanner suppression therefore enter served content without crossing any layer those tools cover. This paper characterizes the resulting exploitation surface and quantifies the corresponding shift-right tooling gap.

An authorized security engagement covered approximately 2,000 enterprise web application assets across one organization. Of these, 113 (5.65%, 95% Clopper-Pearson exact CI: 4.7%–6.8%) contained at least one live credential in served content (JavaScript bundles, HTML view-source, or JSON and XML API responses) that had bypassed every pre-deployment secret scanning control in place. Of those 113 vulnerable applications, 63 (55.8%, 95% Clopper-Pearson exact CI: 46.1%–65.1%) contained a complete Azure AD credential set (client ID, client secret, tenant ID, and resource URI co-located in the same JS bundle) sufficient to execute the full exploitation chain documented in this paper. Not every credential exposure produces an exploitable end-to-end chain. A complete chain requires both the full credential set and over-permissive service principal scopes, as documented in Sections 3.4 and 4.3. Applications were selected based on engagement scope, not screened for likelihood of exposure. Confidence intervals are Clopper-Pearson exact intervals at the 95% level.

To measure the shift-right tooling gap on the same corpus, we constructed an independent Ground Truth (GT-194) of 194 unique secret-grade credentials extracted from production JavaScript bundles across 113 enterprise applications. The benchmark is anchored to extraction by Claude Opus 4.7 (Anthropic) with no access to any of the evaluated tools and was independently validated by an LLM from a different vendor (GPT-5.5, OpenAI; Brennan-Prediger $\kappa = 0.822$). After human review and the field-deployment register of

8 May 2026, 194 unique secret-grade credentials were retained as the locked benchmark across eight credential types (Azure APIM subscription key, Azure AD client_secret, CryptoJS-AES blob, plaintext user credential, JWT, CyberArk AIM, Google API key, and Other API key/token including App Insights iKey). Ten scanners were evaluated against GT-194 at their tightened configurations, with pairwise McNemar significance testing, an UpSet-style detection-set overlap analysis, and a separately tracked set of 249 chain-completion identifiers that are public-by-design and therefore not counted in the recall denominator. Section 8.5 reports the full results.

The remainder of this paper is organized as follows. Sections 3 and 4 document two end-to-end exploitation chains drawn from the corpus. Section 5 characterizes the five structural paths by which credentials reach production undetected. Sections 6 and 7 analyze why both shift-left and shift-right tooling categories systematically miss these paths. Section 8 presents a layered runtime detection methodology and the GT-194 evaluation. Sections 9 through 12 cover recommendations, limitations, future work, and responsible disclosure.

1.1. Scope and Authorization

Both exploitation chains described in this paper were identified during authorized security assessments. Organization names and application identifiers have been anonymized throughout.

1.2. Contributions

This paper makes the following contributions:

1. **Two documented exploitation chains**, end-to-end, reproducible attack paths in which Azure Active Directory client credentials and API subscription keys found in production JavaScript bundles were combined with over-permissive service principal scopes to achieve full account takeover and mass user data exposure in real enterprise applications that had passed all deployed shift-left security controls.
2. **A structural taxonomy of five paths to production** (build-time environment injection, CI/CD pipeline variable substitution, runtime configuration fetching, third-party script inclusion, and scanner suppression at the organizational level), each of which produces live credentials in deployed artifacts without those credentials appearing in the application owner’s repository at any point.
3. **A characterization of the tooling gap**. A systematic analysis of why both shift-left scanning tools (GitLeaks, TruffleHog, SAST) and shift-right tools (DAST, WAF, RASP) fail to detect this class of

vulnerability, grounded in the specific technical properties of minified production JavaScript.

4. **A detection methodology for the runtime layer.** A layered approach combining anchored vendor token patterns, Shannon entropy analysis, and key-value context scanning, designed for the constraints of production JavaScript: minification, noise suppression, and no outbound verification calls.
5. **A prioritized remediation framework,** covering immediate credential rotation, architectural remediation via the Backend for Frontend pattern, service principal scope correction, and ongoing monitoring controls for Azure AD and Azure API Management environments, with generalizations to AWS and GCP stacks.
6. **A cross-vendor LLM-validated benchmark (GT-194) and a quantified tool-agnostic blind spot.** Ten scanners are evaluated against a locked benchmark constructed by Claude Opus 4.7 and independently validated by GPT-5.5 (Brennan-Prediger $\kappa = 0.822$), with pairwise McNemar significance testing and UpSet-style detection-set overlap analysis. The principal finding is structural: 13.9% of GT-194 is missed by every scanner evaluated, predominantly the CryptoJS encrypted-configuration class. Methodological mitigations of the developer-author conflict of interest are detailed in §8.5; recall figures are scoped to this corpus rather than cross-industry estimates (Section 10).

1.3. Related Work

Secret leakage in version control. The leakage of credentials through public repositories is well-documented. Meli et al. ^[1] conducted a large-scale measurement study of public GitHub repositories, identifying thousands of exposed API keys, passwords, and tokens across millions of commits, and established that secret leakage in source code is a systemic problem providing the empirical foundation for tools such as GitLeaks and TruffleHog. Demir et al. ^[2] extended this analysis to live web content, crawling 10 million web pages and identifying 1,748 distinct credentials from 14 cloud and SaaS providers embedded in JavaScript, HTML, and JSON resources, finding that 84% of credentials appeared in JavaScript files and persisted for an average of twelve months. The present paper addresses a complementary structural problem: credentials that reach production without ever appearing in any repository, through build-time injection, pipeline variable substitution, and runtime configuration delivery.

Industry-scale measurement of credentials in served JavaScript. Intruder’s December 2025 measurement ^[4] applied a Nuclei-based JavaScript spider to ≈ 5 million applications and identified $\sim 42,000$ exposed tokens across 334 secret types, providing the first industry-scale lower-bound on credentials in served JavaScript. The present paper is methodologically complementary: a smaller per-application benchmark with cross-vendor LLM-validated ground truth, designed to quantify the structural detection gaps a regex-driven sweep cannot reach.

Secret scanning tool evaluation. Saha et al. ^[2] conducted a comparative evaluation of nine secret detection tools (including GitHub Secret Scanner, Gitleaks, SpectralOps, and TruffleHog) against a benchmark dataset of real-world credential leaks, providing precision and recall measurements and identifying the primary sources of false positives and false negatives across tool categories. Their work establishes the limits of existing scanner coverage. This paper extends that analysis to a scanning layer those tools were not designed to address: the runtime-served application surface.

Credential leakage in mobile and mini-app ecosystems. Qiu et al. ^[5] identified systematic credential exposure in mini-applications hosted within super-app platforms such as WeChat, finding 15 categories of vulnerable services in which credentials embedded in mini-app bundles enabled account hijacking and phishing. While the delivery mechanism differs from web SPA bundles, the structural cause is identical: build-time credential injection into client-delivered artifacts. The exploitation patterns and the tooling gap are the same.

JavaScript bundle security. Khodayari et al. ^[6] conducted an empirical study of JavaScript bundling practices across large-scale web deployments, analyzing bundle composition, dependency inclusion, and the security implications of bundler behavior. Their analysis of AST-level bundle reversibility is directly relevant to the detection methodology described in Section 8: understanding how bundled JavaScript can be reverse-engineered to recover credential context informs both attacker technique and defender detection strategy. Lauinger et al. ^[7] documented the widespread inclusion of outdated and vulnerable JavaScript libraries in web applications, establishing the broader pattern of client-side JavaScript as an understudied attack surface.

Static analysis tool limitations. Alqahtani et al. ^[8] evaluated JavaScript static analysis tools against a dataset of 957 real-world vulnerabilities in npm packages, finding significant gaps in detection coverage across tool categories. Their findings on the limitations of AST-based static analysis for JavaScript are

consistent with the shift-left scanning gaps documented in Section 6: static analyzers operating on pre-build source code cannot detect credentials that are introduced at build time or delivered at runtime.

Client-side JavaScript as an attack surface. Prior work on client-side JavaScript security has focused primarily on XSS vulnerabilities [9], third-party script inclusion risks, and subresource integrity enforcement. DAST tooling has evolved to cover injection vulnerabilities and authentication flaws in running applications. Neither category was designed to identify credentials embedded in served JavaScript bundles or injected into HTML at runtime. This paper maps that gap and presents a detection methodology suited to the runtime attack surface, addressing the structural limitations identified across the related work reviewed above.

1.4. Terminology

- **Shift-left:** Security controls applied before or at the point of code commit, secret scanning, SAST, pre-commit hooks
- **Shift-right:** Security controls applied after deployment, DAST, runtime monitoring, WAF
- **SPA:** Single-Page Application. A web application where the entire frontend is delivered as a JavaScript bundle loaded once by the browser
- **APIM:** Azure API Management. A gateway service that proxies and manages access to backend APIs
- **BFF:** Backend for Frontend. A server-side proxy layer owned by the frontend team that holds service credentials and proxies requests to backend APIs

1.5. Threat Model

Attacker profile and capabilities. This paper models a network-capable attacker with no prior authentication to the target application, the same capabilities available to any web visitor, bug bounty researcher, or penetration tester. The attacker can load the application in a browser, inspect served HTTP responses including JavaScript bundles and JSON API replies, and issue authenticated API calls using any credentials recovered from those responses. The attacker does not need network interception capabilities, special tooling, or insider access. The attack surface is entirely passive: credentials are read from content the application actively delivers to every visitor.

What is in scope. Credentials embedded in client-delivered artifacts (JavaScript bundles, HTML source, statically served JSON configuration files, and JSON or XML API responses) that are accessible without prior authentication. Service credentials (Azure AD client secrets, APIM subscription keys) that are

structurally reachable by this attacker model and that grant API access beyond what the current user session is authorized for. The exploitation chains in Sections 3 and 4 operate entirely within this model.

What is out of scope. Server-side credential stores (Azure Key Vault, environment variables not reflected in served content, database connection strings not returned in API responses). Attacks requiring network interception, active injection, or compromise of infrastructure components. Social engineering, phishing, and supply chain attacks. The structural gap documented in this paper is not a network-layer vulnerability. It is a consequence of credentials materializing in client-accessible artifacts through the deployment paths described in Section 5.

2. Background, Azure AD, APIM, and Why These Credentials Matter

2.1. The Azure AD + APIM Architecture

Azure API Management acts as a reverse proxy gateway sitting in front of backend APIs. It handles authentication and authorization by validating tokens and subscription keys, rate limiting, request and response transformation, and analytics. The typical flow in an Azure-native SPA application is:

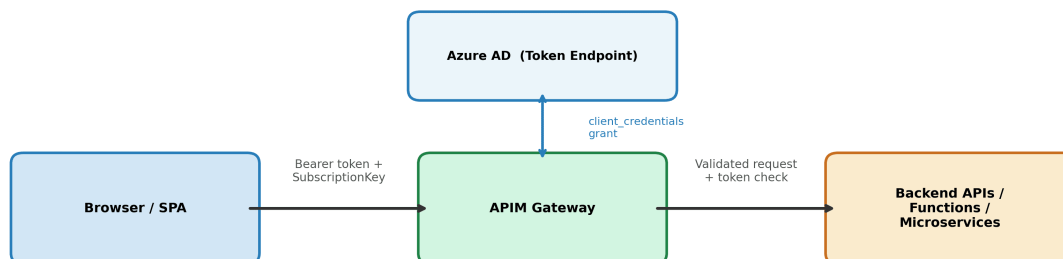


Figure 1. Azure AD + APIM request flow. The browser SPA authenticates via Azure AD using client credentials and calls backend APIs through the APIM gateway using both a Bearer token and a subscription key.

APIM enforces access through two separate credential types: Azure AD tokens (JWTs issued by the identity provider) and subscription keys (gateway-specific pass keys scoped to API products) ^[10]. Both are required to reach protected endpoints.

2.2. The Four Credential Values

In the applications examined, four values were present in the client-side JavaScript bundle:

Credential	Purpose	Should it be in the browser?
AppID (Client ID)	OAuth2 client identifier registered in Azure AD. Tells the identity provider which application is authenticating	Yes, public by OAuth2 spec for public clients
AppKey (Client Secret)	Password paired with AppID. Used in confidential client flows (server-to-server)	Never
Resource	API resource URI or scope used during token acquisition	Harmless alone, combined with AppKey enables full token generation
SubscriptionKey	APIM gateway pass key (Ocp-Apim-Subscription-Key header) required to reach APIM-protected APIs	Never

Table 1.

The AppID is legitimately public in OAuth2 public client flows. The AppKey is not. It is the client secret for a confidential client flow, designed for server-to-server authentication where the application code is not visible to end users ^{[11][12]}. Client secrets are architecturally valid for server-side web applications. The problem is not that the secret exists. It is that it appeared in browser-accessible JavaScript, where it is visible to every visitor who opens developer tools. OAuth2 grant types operate independently on a single application registration. A client secret present on the registration activates the client credentials grant unconditionally, the authorization server applies no constraint between grant types, so the intended authorization flow does not restrict which grants the server will accept ^[11].

2.3. Why All Four Values Together Is Critical

The AppKey alone is insufficient (it requires the AppID and resource URI to generate a token. The SubscriptionKey alone is insufficient) API endpoints require a valid Bearer token. Individually, each value

has limited reach. Together, they represent a complete authentication package: the ability to authenticate as the application itself and call any API the application is authorized to access.

The tenant ID (required to construct the token endpoint URL) was visible in each application's login redirect URL, hardcoded alongside the other values in the same bundle.

3. Exploitation Chain 1, Azure AD + APIM

3.1. Discovery

The client-side JavaScript bundle of a production application (Azure AD authenticated, sitting behind an API Management gateway, serving thousands of users) contained all four credential values described in Section 2.2. The bundle was served over HTTPS to every visitor of the application with no authentication required to retrieve it.

The tenant ID was present in the application's login redirect URL, which was also hardcoded in the bundle.

3.2. Token Generation

The Azure AD token endpoint was called using the extracted credentials:

```
POST https://login.microsoftonline.com/{tenant\_id}/oauth2/token  
grant_type=client_credentials  
client_id={AppID}  
client_secret={AppKey}  
resource={Resource}
```

The endpoint returned a fully authenticated Bearer access token. The grant type used (`client_credentials`) is a confidential client flow designed for server-to-server communication where the client secret is stored securely on the server ^[11]. (The v1 endpoint format is shown, the v2 endpoint uses `/oauth2/v2.0/token` with a scope parameter in place of resource. Both remain in active use across enterprise Azure environments.) The application was using it in a public client context where the secret was visible to every user.

3.3. API Endpoint Reconstruction

Rather than scanning the bundle for additional secrets, it was read as a map of how the application was built. Buried in the minified code were API endpoint definitions, GET and POST endpoints with complete schema structures showing exactly which parameters each endpoint expected. The frontend had documented its own backend.

Using the Bearer token and APIM subscription key together:

```
Authorization: Bearer {token}
```

```
Ocp-Apim-Subscription-Key: {SubscriptionKey}
```

The reconstructed endpoints were called. User profile data returned immediately. A password reset endpoint was identified using the same schema reconstruction approach, same token, same subscription key, same reconstructed schema. The finding was reported at this point and the assessment did not proceed further.

3.4. The Scope Condition

The exposed credentials alone constitute a significant finding regardless of what they unlock. A client secret in a public JavaScript file means any visitor can authenticate as the application, the blast radius depends entirely on what permissions that application has been granted. In this case, the amplifying factor was that the service principal had been granted over-permissive API scopes. Application permissions (acting as the application itself, not as a specific user) had been granted where delegated permissions (acting as the signed-in user) would have been appropriate and sufficient ^{[13][12]}.

The result: the application's client identity had been given the ability to perform user-level operations across every account in the system. Without this misconfiguration, the Bearer token would have had access only to application-level resources. With it, the token was effectively an administrative credential for every user account.

3.5. Subscription Key Blast Radius

One APIM subscription key typically maps to a product containing multiple APIs:

```
SubscriptionKey → APIM Product "Internal APIs"
```

```
+-- /users/*
```

```
+-- /payments/*
```

```
+-- /admin/*  
+-- /reports/*
```

One exposed subscription key grants access to every API in that product. The scope of exposure is determined by how the APIM product is configured. A detail that is not visible from the client side and must be assessed in the APIM portal.

Result: Full account takeover from four values in a JavaScript file served to every visitor of the application.

This was not a sophisticated attack. It required no exploit framework, no vulnerability scanner, and no special tooling. It required reading the JavaScript file the application was already serving to everyone and understanding what the credentials unlocked.

4. Exploitation Chain 2, Client-Side Encryption Bypass

4.1. Discovery

A different application, a different codebase, and a different obfuscation approach, but the same Azure AD credential pattern waiting at the end.

The credentials reached production via Path 1 (Section 5.1): Azure AD client credentials were stored in an environment file and baked into the JavaScript bundle at build time. Rather than leaving them as plaintext, the development team had encrypted the configuration object using CryptoJS before bundling. A pattern intended to obscure the credentials from casual inspection. The developer intent is that the configuration is protected because it is encrypted.

The problem: the decryption key was hardcoded in the same JavaScript file, three lines away from the encrypted string ^{[14][15]}.

4.2. Decryption

The hardcoded key was used to decrypt the environment string using the same CryptoJS call visible in the source:

```
CryptoJS.AES.decrypt(encryptedConfig, hardcodedKey).toString(CryptoJS.enc.Utf8)
```

The output was a complete configuration object. Every Azure credential, every service key, every internal endpoint the application needed to function. The encryption had provided exactly zero protection. The

key to unlock everything was sitting next to the lock.

The decrypted object contained the same pattern: AppID, AppKey, Resource, Azure AD client credentials embedded in what the development team believed was a secured configuration.

4.3. Exploitation

As with Chain 1, the service principal had been granted over-permissive scopes, Application permissions allowing an application-level token to perform user-level data access operations.

From the decrypted credentials:

- A Bearer token was generated using the extracted Azure AD client credentials
- API endpoints were located hardcoded elsewhere in the bundle
- Endpoint schemas were reconstructed from the minified bundle structure
- Authenticated GET endpoints were called with the Bearer token
- Full user profile data (names, email addresses, account identifiers) was returned for any user in the application

Result: Personal data for thousands of users was accessible to anyone who opened the JavaScript file and understood what the encrypted string was hiding.

4.4. Why CryptoJS Obfuscation Is Particularly Dangerous

The CryptoJS pattern creates a false sense of security that is difficult to identify in a standard code review. Developers implement it believing the configuration is protected. Security reviewers see encryption and move on. Static secret scanners see an encrypted string and find nothing to flag.

The construct is meaningless only when you read the entire file rather than scan it for plaintext secrets, which is exactly what static scanners do not do.

This pattern is not isolated to a single application or development team. Across the 113-application corpus assessed in this study, the CryptoJS obfuscation construct (encrypted configuration object with a co-located decryption key) recurred across multiple unrelated codebases, with 16 applications (14.2% of vulnerable apps) containing CryptoJS-encrypted configurations and 37 distinct CryptoJS-AES encrypted-configuration blobs in the GT-194 benchmark. The recurrence across unrelated codebases suggests the pattern originates from a shared internal library, a shared development template, or a common architectural recommendation propagated across teams. A single flawed pattern adopted at the

architecture or template level can introduce the same cryptographic misuse across an entire application portfolio simultaneously.

5. How Secrets Reach Production

Credentials reach production undetected through structural paths in modern build and deployment pipelines, regardless of organizational security maturity. Shift-left tools do not fail in these scenarios. The path from source code to production contains branches that no pre-deployment scanner covers. Five such paths are characterized in Sections 5.1 through 5.5.

5.1. Path 1, Build-Time Environment Injection

React applications using `REACT_APP*` environment variables, and Angular applications using `environment.prod.ts`, pass credentials to the build tool at compile time. webpack’s `DefinePlugin` and the Angular CLI substitute these values directly into the output bundle ^[16].

The credential never exists in the repository. It is read from the build environment at compile time and written into the output artifact. Every secret scanner that ran before the build completed saw nothing, because there was nothing to see.

The Angular CLI’s production build pattern deserves particular attention. The 113 credential-bearing applications break down by framework as follows:

Framework	Apps	Share
Angular (CLI)	75	66%
React	25	22%
REST API / JSON config	5	4%
Legacy / jQuery	4	4%
ASP.NET WebForms	2	2%
React (CRA)	2	2%

Table 2.

Angular’s dominance (66%) is not coincidental. The Angular CLI treats `environment.prod.ts` as the documented and recommended approach for environment-specific configuration, credential injection is the framework’s intended pattern. webpack’s `DefinePlugin` structurally ensures that any value placed in the environment file is compiled directly into the output bundle. The prevalence of credential exposure across this portfolio is a direct consequence of framework architecture, not isolated developer error. Where a framework’s recommended pattern produces credentials in client-delivered artifacts by design, the risk is systemic rather than individual.

5.2. Path 2, CI/CD Pipeline Variable Substitution

This is a frequently observed path. A placeholder value lives in the repository while the real credential is stored as a pipeline variable in Azure DevOps, GitHub Actions, or a similar CI/CD platform. The placeholder is what every repository scanner, pre-commit hook, and SAST tool sees. The credential materializes only in the build artifact, after every scanner has already completed. It never exists in git at any point.

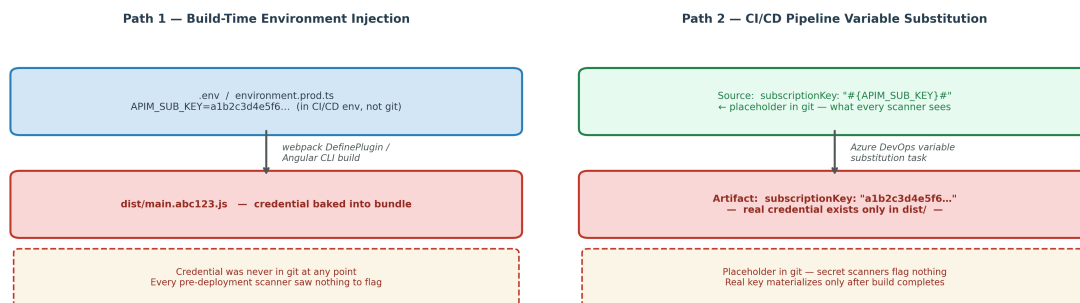


Figure 2. Two structural paths by which credentials reach production without repository exposure. Left, build-time environment injection: the credential is injected by webpack or the Angular CLI at compile time and never exists in git. Right, CI/CD pipeline variable substitution: a placeholder lives in source while the real credential is written into the artifact only after every scanner has completed.

5.3. Path 3, Runtime Configuration Injection

Some applications fetch their configuration after the browser loads the initial bundle:

- A `/assets/config.json` file served statically at runtime alongside the application
- A `window.__APP_CONFIG__` object injected into `index.html` by the server at request time
- SSR state blobs (the `__NEXT_DATA__` object injected by Next.js or `window.__INITIAL_STATE__` injected by Nuxt) which regularly contain tokens, API endpoints, and internal service configuration

These configurations are loaded after deployment, sometimes from CDN nodes, sometimes from origin servers. They do not exist as files in the repository. They are assembled and served at runtime, invisible to any scanner that ran before deployment.

In the applications assessed for this study, two targets were REST API configuration endpoints (not JavaScript bundles) returning credential-bearing JSON responses directly to the browser. In both cases the endpoint was called by the SPA on load to retrieve runtime configuration, and the response contained Azure AD client credentials in plaintext JSON fields. These endpoints were publicly accessible without authentication. This confirms that the runtime attack surface extends beyond JavaScript files to include configuration APIs that serve secrets to the browser on demand, and that no JavaScript-only scanner would detect credentials delivered through this path.

5.3.1. This Pattern Extends Beyond Azure

The exploitation chains documented in this paper involve Azure AD and Azure API Management because that is the stack present in the applications assessed. The same structural paths appear across other providers. The remediation guidance in Section 9 is Azure-specific and should be adapted to the relevant platform.

The same paths appear across every stack:

- **AWS:** Cognito user pool client IDs and client secrets embedded in React bundles via `REACT_APP_*` variables. API Gateway API keys substituted into JavaScript at build time. S3 presigned URL generation credentials hardcoded in SPA configuration.
- **GCP:** Firebase API keys and project configuration objects (`apiKey`, `authDomain`, `projectId`) served in `firebase-init.js` to every visitor. These are legitimately public for Firebase Authentication but are frequently accompanied by service account credentials that are not.

- **Generic SaaS:** Twilio account SIDs and auth tokens, SendGrid API keys, Stripe secret keys, and Mailgun API keys substituted into build artifacts from CI/CD pipeline variables. The credential format changes. The path to production does not.

The exploitation technique is identical across all of these: read what the application serves, identify the credential, authenticate with it. The shift-left tooling gap is identical: none of these credentials touched a repository at any point in their path to the browser.

5.3.2. Credentials Persist Across All Environments

Because credential injection occurs at build time, every environment that runs a build receives the same secrets baked into its artifact. Development, SIT, UAT, pre-production, and production environments all receive credentials through the same pipeline, and all produce the same exposure.

Across the applications assessed in this study, the same credential set was confirmed present in production, UAT, SIT, and pre-production environments of the same application in multiple cases. Lower environments typically have weaker access controls, broader team access, and no security monitoring, making them at least as attractive a target as production for an attacker who has identified the pattern.

This has a direct consequence for remediation: rotating credentials in production without simultaneously fixing the build pipeline leaves every lower environment still exposed. An attacker who has already extracted credentials from a UAT environment retains access regardless of what happens in production. Remediation is only complete when the credential is removed from the build pipeline and rotated across all environments simultaneously.

5.4. Path 4, Third-Party Script Inclusion (Supply-Chain Credentials)

Modern web applications routinely embed third-party scripts. Analytics platforms, tag managers, customer-support widgets, error monitoring SDKs, advertising libraries, and CDN-hosted dependencies all execute in the application's origin and run with full access to its DOM, cookies, and storage. These scripts also carry their own credentials, including API keys, project identifiers, instrumentation tokens, and customer-tenant identifiers that are public-by-design from the third party's perspective but materialize in the served bundle of every site that integrates them.

Two structural problems follow. First, the credentials embedded by third parties are outside the application owner's repository, so no scanner the owner runs can flag them. They enter the browser through a script tag that the owner did not author. Second, the security posture of those third-party

endpoints, including rate limiting, scope enforcement, and audit logging, is determined by the third party rather than the application owner. Demir et al. [3] reported that 16% of verified credential exposures originated from third-party inclusions, with the highest third-party rates observed for OpenAI (24%), Twilio (24%), Mailchimp (23%), Alibaba (23%), Stripe (22%), and SendGrid (22%) credentials served via embedded SDKs that the integrating application did not control. In the corpus assessed for this study, the Google API keys (6 instances in GT-194) and a subset of the Other API key/token class are consistent with this path, integrated through third-party SDKs whose credential material enters the served bundle outside any first-party scanning workflow. The exposure pattern is structurally indistinguishable from build-time or runtime injection. A credential reaches a deployed artifact, is served to every visitor, and has no automated detection on either side of the integration.

5.5. Path 5, Scanner Suppression at the Organizational Level

A fifth contributing factor operates at the organizational level and compounds the four structural paths above. Many SPA architectures structurally require credentials in the browser, including Google Maps API keys, Firebase configuration, Stripe publishable keys, and Twilio client tokens. When shift-left scanners flag these, the operational path of least resistance is to suppress the alert through ignore-listing files, dismissing alerts as false positives, adding inline suppression comments, or allowlisting specific patterns in the SAST ruleset. The suppression is deliberate. The developer knows the credential is real but believes it is an accepted architectural risk or lacks the authority or time to implement the server-side proxy pattern (Section 9.1) that would eliminate the in-browser requirement entirely. The credential is live, required, and invisible to the security tooling stack. The build process did not bypass the scanner. The scanner was deliberately told to look away. The outcome is identical to the structural paths. A live credential reaches a production artifact with no automated monitoring.

6. Why Shift-Left Scanners Miss This

The five paths in Section 5 produce credentials in deployed artifacts through different mechanisms but converge at the same boundary. Shift-left scanners run before or at the point of commit, before the build executes, before the pipeline substitutes variables, before any third-party script is fetched, and before the artifact is assembled. The credential does not exist when the scanner runs. By the time it exists in the served artifact, every shift-left scanner has already completed.

Table 3 maps this timing mismatch precisely.

Pipeline Stage	Credential present?	Shift-left scans here?	Shift-right covers here?
Source code / git repository	No, placeholder or absent	Yes	No
CI/CD pipeline variable store	No, stored server-side	No	No
Build artifact (dist/)	Yes, injected at compile time	No	No
Third-party SDK delivered at runtime	Yes, embedded by integration	No	No
Deployed to CDN / Blob Storage	Yes, served to every visitor	No	No
Runtime-fetched config (/assets/config.json)	Yes, fetched by browser	No	No

Table 3.

The gap is not a configuration failure or a tool weakness. It is a structural property of when each scanning layer operates relative to when the credential materializes. No tuning of existing tools closes it. The scanning layer itself must change.

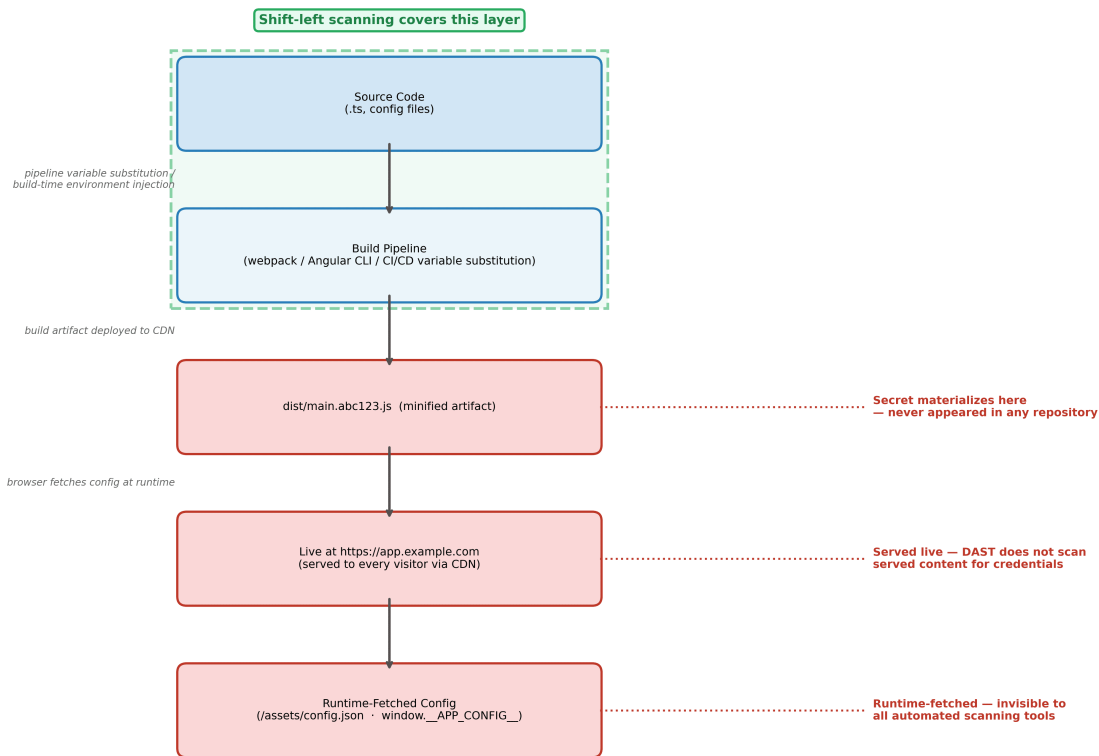


Figure 3. Where shift-left scanning stops versus where secrets materialize. Shift-left tooling covers only the source code layer. Secrets introduced through build-time injection, pipeline variable substitution, runtime configuration fetching, or third-party script inclusion exist only in deployed artifacts and served content. These are layers no pre-deployment scanner reaches.

The shift-left tooling market is mature. GitLeaks, TruffleHog, detect-secrets, and GitHub Advanced Security are widely deployed ^[1]. SAST tools (Sengrep, SonarQube, Checkmarx) are standard in enterprise CI/CD pipelines. Each tool does exactly what it is designed to do. None of them are designed to scan build artifacts, third-party SDK output, or live deployed applications.

Scanner	What It Scans	Why It Misses
GitLeaks	Git commits, staged files, history	Pipeline variables are substituted after git. Placeholder is in git, real value is not. Never sees dist/
SAST (Checkmarx, SonarQube, Veracode)	Source.ts/js pre-build	Sees process.env.SUBSCRIPTION_KEY, no secret present. Does not execute the build
Semgrep / CodeQL	AST of source code	Pre-build source only, same gap as SAST
GitHub Advanced Security	Repository content	{{secrets.KEY}} injected into build output never touches the repository

Table 4.

6.1. The Minification Problem

Pattern-based secret scanners rely on variable names and known key formats to identify secrets. Minification destroys variable names:

```
// Source, scanner identifies this:
const subscriptionKey = "abc123xyz789..."

// Minified, scanner may miss this:
const a={b:"abc123xyz789..."}
```

Entropy-based detection addresses part of this problem (high-entropy strings are flagged regardless of variable name) but entropy alone generates significant noise against the minified JavaScript of a modern SPA, which contains many high-entropy strings that are not credentials (hashed asset filenames, base64-encoded resources, obfuscated library code).

6.2. Source Maps, The Silent Bypass

Angular and webpack generate.map files alongside minified bundles by default:

dist/main.abc123.js ← minified

dist/main.abc123.js.map ← full original source, variable names, logic intact

If source maps are deployed to production (a common default) any credential “hidden” in minified JavaScript is fully readable in browser developer tools via the Sources panel. SAST scans the.ts source. The.map file exposes everything at runtime regardless of what the scanner found. Disabling source maps in production builds (sourceMap: false in angular.json) removes this bypass entirely.

7. Why Shift-Right Scanners Also Miss This

Dynamic Application Security Testing tools receive JavaScript files as part of application spidering, but they treat those files as attack delivery vehicles, not as targets. DAST tools spider the application looking for injection points, authentication flaws, and misconfigurations. They do not parse main.js for Ocp-Apim-Subscription-Key patterns.

The shift-right tooling landscape by category:

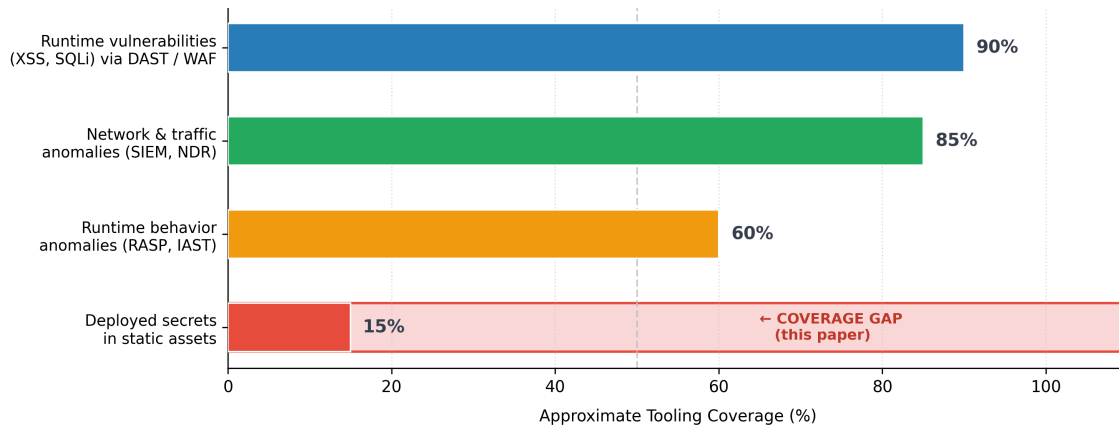


Figure 4. Approximate tooling coverage by runtime security category. Mature tooling exists for injection vulnerabilities and network anomalies. Deployed secrets in static assets and served content represent a largely unsolved coverage gap.

Tools that partially address this class of finding:

Tool	Approach	Limitation
Nuclei	Community templates matching secret patterns in HTTP responses	Regex-based, misses minified variable names, template coverage varies
TruffleHog (filesystem/web mode)	Entropy and pattern detection on files	Web mode experimental. Reliable on artifact filesystem, not live application scanning
Detectify / Probely	DAST with secret detection, downloads JS assets	Does not execute JavaScript, misses runtime-fetched configurations
GitGuardian perimeter	Monitors public-facing endpoints	Focused on public GitHub exposure, not internal deployed applications

Table 5.

None of these tools provide comprehensive coverage of the full runtime attack surface of a modern SPA: split webpack bundle chunks loaded dynamically, SSR state objects injected into HTML, JSON and XML API responses returning credentials in fields never meant to be client-facing, and outbound request headers carrying subscription keys and bearer tokens on every API call.

The result: once a secret evades shift-left controls and reaches production, it is effectively invisible to automated tooling. The only things finding runtime secrets in production today are manual penetration testers, bug bounty researchers, and attackers. Two of those three report what they find.

8. What Runtime Layer Detection Requires

Closing the shift-right gap requires a different scanning model than the tools currently deployed. This section describes what a tool operating at the runtime layer must address, the attack surface it must cover, the detection approach required for minified production JavaScript, and the operational constraints that determine whether findings are actionable in practice.

8.1. Attack Surface Coverage

A runtime-layer scanner must cover what the application actually serves, not what exists in its source repository. The full attack surface of a modern web application includes:

- **JavaScript bundles**, including split webpack chunks that modern SPAs load dynamically as users navigate, not just the initial bundle
- **HTML source**, SSR state blobs (`__NEXT_DATA__`, `window.__INITIAL_STATE__`) injected by server-side rendering frameworks, which regularly contain tokens and internal configuration ^[13]
- **JSON API responses**, credentials returned in fields never intended to be client-facing, which static scanners never see
- **XML responses**, enterprise services carrying connection strings and service credentials in structured response bodies
- **Outbound request headers**, subscription keys and bearer tokens sent on every API call in plaintext

For bulk assessment, following `<script src>` references and tracing chunk URLs is required to scan an application's full JavaScript surface rather than only the files present in the initial page load.

8.2. Detection Approach

Production JavaScript presents a different detection problem than source code. A layered approach is required:

Anchored vendor token patterns, Known credential formats from cloud providers (Azure, AWS, GCP), SaaS platforms (Twilio, SendGrid, Stripe), and authentication services have identifiable structure. Anchored regex patterns matched against specific token formats produce fewer false positives than generic high-entropy matching and are resistant to minification because they target the value format, not the variable name.

Shannon entropy analysis (For credentials without known formats) internal API keys, session tokens, symmetric encryption keys, Shannon entropy identifies high-randomness strings that are statistically unlikely to be application logic ^[17]. Entropy analysis catches credentials that minification has stripped of their variable name context. However, entropy analysis alone against minified JavaScript generates significant noise: hashed asset filenames, base64-encoded resources, and obfuscated library code all produce high-entropy strings that are not credentials.

Key-value context scanning, Examining the structural context surrounding a high-entropy value reduces noise by requiring that the value appear in a credential-relevant context. A JSON key named `subscriptionKey`, `apiKey`, or `clientSecret` paired with a high-entropy string is significantly more likely to be a credential than an isolated high-entropy string. This context requirement is the primary mechanism for reducing false positives in minified JavaScript where variable names are not available.

PII detection with validation, Pattern matching alone is insufficient for PII. Credit card numbers require Luhn algorithm validation [18]. Social Security Numbers require format and checksum verification. Without validation, the false positive rate for these patterns in arbitrary JavaScript is too high to be actionable.

8.3. Operational Constraints

Two operational constraints determine whether runtime scanning findings are actionable in practice:

No outbound verification calls. Verifying whether an exposed API key is valid by calling the issuing service generates log entries at the target provider, can trigger security alerts, and reveals the assessment to defenders. Runtime scanning must run locally with no outbound verification calls to third-party APIs or services.

Noise suppression is a first-class requirement. A scanner that generates hundreds of false positives per scan is a scanner that gets disabled. In the practitioner workflows where runtime scanning is most needed (penetration testing engagements, bug bounty assessments, security reviews) findings must be trustworthy enough to act on immediately. Every false positive erodes that trust and increases the likelihood that a real credential is dismissed as noise. Noise suppression is not a secondary quality concern. It is what determines whether the tool gets used. The detection methodology described in this section informed the design of SecretSifter, referenced in Section 9.3.

Input: URL, target application URL

Output: F , set of (credential_type, value, context) findings

```
1. F ← ∅

2. R ← HTTP_GET(URL, follow_redirects=true)

3. C ← extract_content_units(R)

// C = {JS bundles, HTML blobs, JSON responses, headers}

4. for each chunk_url in extract_script_srcs(R.html) do
    5. C ← C ∪ {HTTP_GET(chunk_url)}
    6. end for
    7.

    8. for each content_unit u in C do
        9. // Layer 1: Anchored vendor token patterns
        10. for each pattern p in VENDOR_PATTERNS do
            11. matches ← regex_findall(p.regex, u)
            12. for each m in matches do
                13. F ← F ∪ {(p.credential_type, m, surrounding_context(m, u))}
            14. end for
        15. end for
    16.

    17. // Layer 2: Shannon entropy + key-value context
    18. tokens ← tokenize_kv_pairs(u)
    19. for each (key, value) in tokens do
        20. if shannon_entropy(value) ≥ ENTROPY_THRESHOLD then
            21. if key ∈ CREDENTIAL_KEY_NAMES then
                22. F ← F ∪ {(infer_type(key), value, key)}
            23. end if
        24. end if
    25. end for
    26.

    27. // Layer 3: PII with validation
    28. for each pattern p in PII_PATTERNS do
        29. matches ← regex_findall(p.regex, u)
        30. for each m in matches do
```

```

31. if pvalidator(m) = true then
32. F ← F ∪ {(p.credential_type, m, surrounding_context(m, u))}
33. end if
34. end for
35. end for
36. end for
37.
38. return deduplicate(F)

```

Algorithm 1. Runtime Credential Detection

VENDOR_PATTERNS includes anchored regex for Azure AD (*client_id*, *client_secret*), APIM subscription keys, AWS access keys, Google API keys, and 40+ additional provider formats. *ENTROPY_THRESHOLD* = 3.5 bits/char. *CREDENTIAL_KEY_NAMES* includes *apiKey*, *subscriptionKey*, *clientSecret*, *appKey*, *encryptionKey*, and synonyms. No outbound verification calls are made at any step.

8.4. Preliminary Case Study: Detection Outcomes (N=2)

This section reports detection outcomes from the two confirmed-positive applications in the authorized assessments described in Section 1. This is a preliminary case study with N=2 confirmed-positive applications, not a controlled benchmark evaluation. No ground-truth labeled corpus of production JavaScript exists for this class of finding. Constructing one from unauthorized applications is not ethically feasible. What follows is a factual account of what the runtime detection methodology identified in both confirmed-positive cases, and why no other automated control was positioned to find the same.

Credentials in the detection gap. In both confirmed-positive applications, the exposed credentials had reached production through the structural paths described in Section 5, build-time environment injection and CI/CD pipeline variable substitution. By the time the credentials were present in the served application, they had passed every automated control in the pre-deployment pipeline. This is not a statement about whether those controls ran or how well they were configured. It is a structural property of the paths: credentials introduced after the build scanner runs, or substituted by the pipeline after the repository scanner completes, do not exist in any layer those tools examine. The shift-left scanning layer had nothing to find because the credentials were not there when it looked.

No shift-right control covered this layer. Once deployed, both applications were running in production environments with standard enterprise security controls, including web application firewalls and network monitoring. None of those controls are designed to examine the content of served JavaScript files for embedded credentials. DAST tools spider applications for injection vulnerabilities, WAF and network monitoring tools inspect inbound traffic for attack patterns. Neither category asks whether the JavaScript being served to users contains an Azure AD client secret. The credentials were live, publicly served, and invisible to the entire deployed security stack, not because any tool failed, but because no tool in any category was scanning that layer.

Runtime detection results. The layered detection approach described in Sections 8.1-8.2 (anchored vendor token pattern matching combined with key-value context scanning) identified the complete credential sets in both confirmed-positive applications. In both cases, the credentials were located in served JavaScript bundles. In Chain 1 (Section 3), the AppID, AppKey, Resource, and SubscriptionKey were matched directly by anchored Azure AD and APIM credential patterns. In Chain 2 (Section 4), the encrypted configuration was located by key-value context scanning identifying a high-entropy value paired with a credential-relevant key name. Decryption was performed manually using the co-located key, after which the same credential patterns applied. Both findings were confirmed by successful exploitation before disclosure.

Credential set completeness. Of the 113 applications with confirmed credential exposure, 63 (55.8%) contained a complete Azure AD credential set (client_id, client_secret, tenant_id, and resource URI co-located in the same JS bundle) sufficient to execute the exploitation chain documented in Section 3 given over-permissive service principal scopes. The remaining 50 applications contained partial credentials. Across all 113 vulnerable applications the most common credential types observed were Azure App Insights instrumentation keys (62 apps, 54.9%), Azure APIM subscription keys (46 apps, 40.7%), JWT tokens issued to the browser (25 apps, 22.1%), CryptoJS-encrypted configuration objects (16 apps, 14.2%), and CyberArk AIM tokens (11 apps, 9.7%). Applications frequently contained multiple credential types simultaneously. Partial credentials represent significant exposure but do not independently enable the full account takeover chain. The high proportion of complete Azure AD sets (56% of vulnerable applications) indicates that the credential injection pattern typically carries the full credential bundle rather than isolated values. Per-app prevalence and per-credential GT-194 counts can diverge in either direction (for example, 62 apps share 15 unique App Insights workspace iKeys, while the 16 CryptoJS-using apps contain 37 distinct encrypted blobs in GT-194).

Scope and limitations. The sample of 113 applications is drawn from a single authorized engagement scope of approximately 2,000 enterprise web application assets, not a randomly selected population across multiple organizations. Detection was performed by a skilled practitioner applying the methodology described in this section, not by a fully automated tool running without guidance. A formal evaluation (with a labeled benchmark corpus, automated tool execution, and precision and recall measurements at scale) remains an area for future empirical work. The outcomes reported here establish that the methodology successfully identifies credentials in the layer that existing tooling cannot reach, and that this layer contains real, exploitable credentials in production systems that have passed all deployed automated controls.

8.5. Tool Comparison Study, GT-194 Cross-Vendor LLM-Validated Benchmark

Section 8.4 established that the runtime detection methodology successfully identifies credentials in two confirmed-positive applications. This section quantifies how every other scanner category (established static tools at default and tightened-custom configurations, the runtime-aware extension SecretSifter, and a state-of-the-art LLM oracle) performs against an independent Ground Truth, GT-194, on the same corpus. The evaluation is deliberately structured to remove the developer-author conflict of interest from the headline numbers: the Ground Truth is constructed by an LLM with no access to any evaluated detector, every label was independently classified by an LLM from a different vendor, and the field-deployment register was locked on 8 May 2026.

8.5.1. Benchmark Construction

What “GT-194” means. GT-194 is the Ground Truth set of 194 unique secret-grade credentials assembled from production JavaScript bundles across 113 enterprise applications. “Ground Truth” here means the labelled, manually validated set of credentials that scanners are measured against, the canonical denominator for recall calculations throughout this section. GT-194 includes only credential instances that are actually secret-bearing, partitioned into eight credential types: Azure APIM subscription key (n=54), Azure AD client_secret incl. btoa-encoded OAuth (n=50), CryptoJS-AES Salted__ blob (n=37), Other API key/token incl. App Insights iKey (n=29), JWT (n=11), Google API key (n=6), CyberArk AIM (n=4), and plaintext user credential (n=3). The eight type counts sum to 194.

Corpus. The 113-application benchmark is the full set of credential-bearing applications identified during the engagement (drawn from approximately 2,000 enterprise web application assets in scope). It includes

all applications providing downloadable JavaScript bundles suitable for both static-scanner evaluation and LLM-oracle extraction, and captures the full credential-bearing production deployment set including builds containing CryptoJS-AES blobs, Other API keys, and plaintext user credentials.

Independent Ground Truth extraction. Claude Opus 4.7 (Anthropic) [19] was prompted to extract every value from the bundles that it considered a real high-impact secret. The model was given only the bundle content and a strict-secret rubric, with no access to SecretSifter rules, output, or any other tool’s findings. After human review of the extracted candidates, the locked GT-194 was assembled.

Cross-vendor independent validation. Each Opus-extracted candidate was independently classified by GPT-5.5 (OpenAI) [20] via the OpenAI Codex CLI. GPT-5.5 received the candidate value, key name, and source-file context, and was asked to apply the same strict-secret rubric independently. Inter-rater agreement between Opus and GPT-5.5 on the binary SECRET classification was Brennan-Prediger $\kappa = 0.822$, almost-perfect on the Landis-Koch interpretive scale [21]. Because the GT-194 contains exclusively SECRET-labeled credentials (single-class prevalence), Cohen’s κ is undefined here (the Feinstein-Cicchetti paradox); Brennan-Prediger κ is the bias-corrected metric appropriate for single-class prevalence and is reported as the headline cross-vendor agreement statistic.

Why public-by-design Azure identifiers are tracked separately (“below the line”). The corpus also contains 249 chain-completion identifiers, 153 Azure AD App IDs / client_ids / tenant_ids and 96 Azure AD resource URIs (resource, Resource, *_Resource, resourceId). These are public-by-design Azure identifiers, not credentials, and are therefore excluded from the GT-194 recall denominator. Two reasons drive the structural choice. First, including them would artificially inflate scanner scores, since these are not credentials a peer reviewer would accept as “leaks.” Second, excluding them entirely would hide the operational reality that all four components (client_secret, client_id / App_ID, tenant_id, and resource URI) must be combined to mint an Azure AD access token. Listing them separately preserves both honesty about the recall metric and visibility into the exploitation chain. The chain-completion identifiers are reported in Table 9 below the strict GT-194 total (“below the line”) and as a separate corpus-property finding in Table 10.

Per-credential counting protocol. All per-tool hit counts in this section are computed as per-credential matches against GT-194. Each row in the Ground Truth corresponds to a distinct unique credential value. The unique-credential count is the canonical recall metric throughout. For two of the ten scanners (SecretSifter and Claude Opus, the two highest-recall scanners), summing per-type detection counts yields totals higher than the unique-credential total, 189 versus 151 for SecretSifter, and 175 versus 166 for

Claude Opus. The eight lower-recall scanners have no such discrepancy. The reason: a small number of credential values in the corpus are tagged with multiple type labels (e.g., a value found as both a `client_secret` in one HTTP request and an `Other API key/token` elsewhere). For tools with broad cross-type coverage (SS, Claude), this overlap shows up as inflated per-type sums; for low-recall tools, it does not. The unique-credential count remains the canonical recall metric and is what appears in headline tables and figures.

8.5.2. Tool Corpus and Evaluation Protocol

Ten scanners were evaluated against GT-194: SecretSifter (curated runtime extension), TruffleHog, JSluice, SecretFinder, Titus, JSMiner, Nuclei, Sensitive Discoverer, Cariddi (static / template-driven scanners), and Claude Opus 4.7 (the LLM oracle, included for reference because it serves as the GT extractor). Each tool was run at its tightened configuration. For the rule-extensible tools (TruffleHog, SecretFinder, Titus, JSluice), tightened configuration adds the same 6-rule overlay covering the most prevalent credential shapes in the corpus: Azure AD `client_id` (UUID v4), Azure AD `client_secret` (40-character base64 with embedded tildes), APIM subscription key (32-hex), JWT (`eyJ-`prefix three-segment), App Insights instrumentation key, and CryptoJS `U2FsdGVkX1-`prefix encrypted blobs. The full regex set is reproduced in Appendix A. Nuclei was run with the secret-scanning template categories `exposures/tokens`, `exposures/apis`, and `exposures/keys`. JSMiner has hard-coded patterns and does not expose a user-rule API, it was run at its built-in configuration. SecretSifter and Cariddi were run at their built-in configurations. Per-tool finding outputs were matched to GT-194 via exact-value substring lookup with no other post-processing.

Cariddi achieves $2/194 = 1.0\%$ recall on GT-194, a non-zero data point that contributes signal to the bottom of the recall ranking. Cariddi's primary mode is web-crawling and endpoint enumeration; secret detection is a secondary capability with limited overlap with this corpus's credential mix.

Author-developer disclosure. SecretSifter is open-source software developed by the first author (Gorijala). The evaluation methodology is structured to make the comparison independent of this fact: GT-194 was constructed by an LLM (Claude Opus 4.7) with no access to SecretSifter's rules, output, or source code; every label was independently validated by an LLM from a different vendor (GPT-5.5, OpenAI); the same 6-rule overlay was applied identically to all rule-extensible tools with no per-tool tuning (Appendix A); and the rule overlay was selected from the credential shapes most prevalent in GT-194 without consulting any tool's output. The headline numbers reflect performance on a single-

organization corpus and should be interpreted as benchmark performance on this corpus rather than as cross-organization population estimates (Section 10).

Tool versions and source. All scanners were evaluated at the versions listed in Table 6. Each version is the latest release available at the GT-194 lock date (8 May 2026); commit hashes are recorded for tools without semantic versioning. The evaluation environment was macOS 25.1 (Darwin), Burp Suite Professional 2025.10 hosting the Burp-extension scanners (SecretSifter, JSMiner, Sensitive Discoverer), and Python 3.12 for the JSMiner re-implementation.

Scanner	Version evaluated	Source
SecretSifter	1.0.1	https://portswigger.net/bappstore (BApp Store) and https://github.com/secretsifter/burp-secret-scanner
TruffleHog	3.83.7	https://github.com/trufflesecurity/trufflehog
JSluice	0.0.7	https://github.com/BishopFox/jsluice
SecretFinder	commit a07d215 (Apr 2026)	https://github.com/m4ll0k/SecretFinder
Titus	1.4.0	https://github.com/Falconcyber-research/Titus
JSMiner	Trustwave SpiderLabs distribution v1.0; Python re-implementation included with this paper	https://github.com/SpiderLabs/JS-Miner
Nuclei	3.3.10 with template repository commit 2026-04-30	https://github.com/projectdiscovery/nuclei
Sensitive Discoverer	9.4 (BApp Store, May 2026)	https://portswigger.net/bappstore and https://github.com/CYS4srl/SensitiveDiscoverer
Cariddi	1.4.1	https://github.com/edoardottt/cariddi
Claude Opus 4.7 (LLM oracle)	claude-opus-4-7 (Anthropic API, Mar 2026 release)	https://docs.anthropic.com/en/docs/about-claude/models
GPT-5.5 (cross-vendor validator)	gpt-5.5 (OpenAI API via OpenAI Codex CLI v0.6.x)	https://platform.openai.com/docs/models

Table 6. Scanner versions and sources. Each row records the exact version evaluated against GT-194 and the public source where the same version can be obtained. Claude Opus 4.7 and GPT-5.5 are listed at the bottom of the table because they serve as the LLM oracle and cross-vendor validator respectively, not as evaluated detection scanners.

8.5.3. Headline Recall Comparison

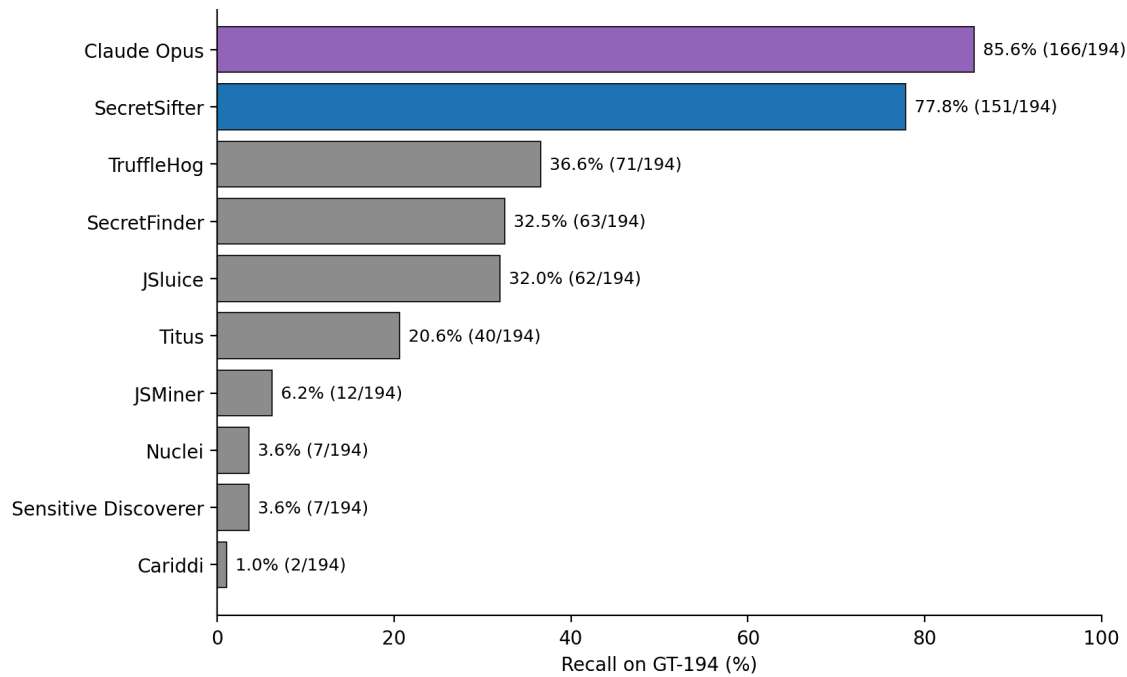


Figure 5. Per-tool recall on GT-194. SecretSifter (151/194 = 77.8%) and Claude Opus (166/194 = 85.6%) lead the field by approximately 41 percentage points over the best static scanner (TruffleHog 71/194 = 36.6%).

Rank	Scanner	Role	Detected	GT total	Recall
1	Claude Opus 4.7	LLM oracle	166	194	85.6%
2	SecretSifter	curated runtime scanner	151	194	77.8%
3	TruffleHog	static scanner	71	194	36.6%
4	SecretFinder	static scanner	63	194	32.5%
5	JSluice	static scanner	62	194	32.0%
6	Titus	static scanner	40	194	20.6%
7	JSMiner	Burp extension	12	194	6.2%
8	Nuclei	template-driven scanner	7	194	3.6%
9	Sensitive Discoverer	Burp extension	7	194	3.6%
10	Cariddi	web-crawler / scanner	2	194	1.0%

Table 7. Per-tool overall recall on GT-194. Counts are per-credential matches against the locked GT-194 secret-grade Ground Truth.

The shift-right tooling gap is now quantified end-to-end on GT-194. From the LLM oracle ceiling (85.6%) to the best static tool (TruffleHog at 36.6%) is a 49.0 percentage-point loss. SecretSifter recovers 41.2 of those 49.0 percentage points (84.1% of the residual gap), narrowing the gap to the LLM oracle to 7.8 percentage points. SecretSifter’s lead over TruffleHog is +41.2 percentage points, equivalent to 2.13× the static-scanner recall.

8.5.4. Precision and False-Positive Rate

Recall is half the operational story. The other half is precision, how many findings each scanner emits beyond GT-194, and what proportion of those emissions are credentials versus noise. A scanner that finds every credential but emits ten thousand false positives per application is unusable in a security-engineering workflow. This subsection reports precision, recall, and F1 score for the ten scanners against GT-194 on the synthetic-portal validation pass.

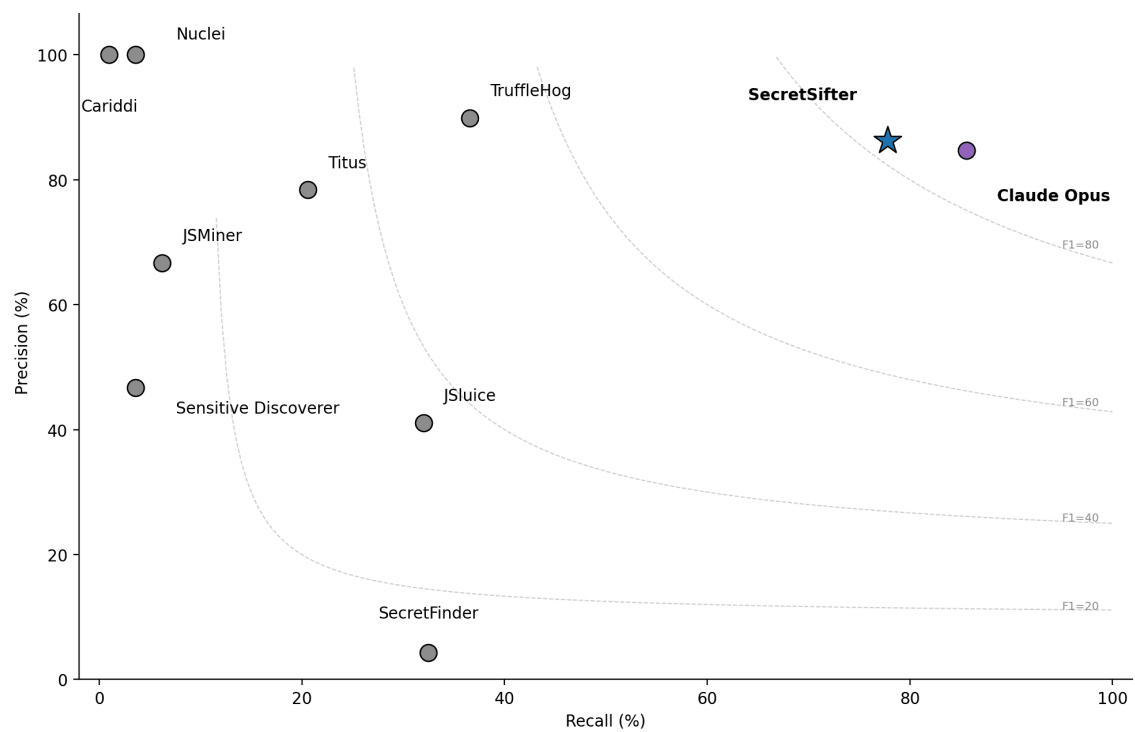


Figure 6. Precision vs. recall scatter on GT-194. Diagonal contours show iso-F1 lines. SecretSifter (★) and Claude Opus achieve the highest F1 by combining high precision with broad recall.

Scanner	TP	FP	FN	Precision	Recall	F1
Claude Opus 4.7	166	30	28	84.7%	85.6%	0.851
SecretSifter	151	24	43	86.3%	77.8%	0.818
TruffleHog	71	8	123	89.9%	36.6%	0.520
SecretFinder	63	1,402	131	4.3%	32.5%	0.076
JSluice	62	89	132	41.1%	32.0%	0.360
Titus	40	11	154	78.4%	20.6%	0.327
JSMiner	12	6	182	66.7%	6.2%	0.113
Nuclei	7	0	187	100.0%	3.6%	0.070
Sensitive Discoverer	7	8	187	46.7%	3.6%	0.067
Cariddi	2	0	192	100.0%	1.0%	0.020

Table 8. Precision, recall, and F1 on GT-194. TP = true positives (GT-194 credentials correctly detected); FP = false positives (non-credential strings flagged as credentials, measured on the synthetic-portal validation pass); FN = false negatives (GT-194 credentials missed). Precision = $TP / (TP + FP)$. Recall = $TP / (TP + FN)$. F1 = $2 \cdot PR / (P + R)$.

Three structural observations:

1. **The LLM oracle sets the F1 ceiling at 0.851; SecretSifter has the highest F1 of any measured non-LLM scanner at 0.818.** Claude Opus 4.7 reaches F1 = 0.851 (precision 84.7%, recall 85.6%). Among non-LLM scanners, SecretSifter is the only scanner within striking distance: F1 = 0.818 (precision 86.3%, recall 77.8%), with the 0.033 gap from Opus quantifying the headroom remaining for non-LLM curated detection. SecretSifter's lead over the best static scanner (TruffleHog, F1 = 0.520) is +0.298. The lead is structural rather than precision-driven: TruffleHog has nominally similar precision (89.9% vs SecretSifter 86.3%) but recovers only 36.6% of GT-194, while SecretSifter's higher recall (77.8%) under comparable noise translates into the higher harmonic mean.

2. **Regex architecture without context awareness catastrophically inflates false-positive rate on minified JavaScript.** SecretFinder produces 1,402 false positives on the validation pass. The structural reason: generic UUID, 32-hex, and base64-shaped patterns match webpack chunk hashes, build fingerprints, asset content-hashes, internal element identifiers (`questionPanelId`, `FRAUDNET_FNCLS`, `proposalId`), and Angular framework constants (`Inject`, `providedIn`, `NullInjectorError`) that are ubiquitous in minified production JavaScript. The same regexes that produce signal in repository code produce overwhelming noise on built artifacts. TruffleHog's lower FP count (8) reflects its built-in entropy and verification heuristics; with custom rules disabled and verification off, TruffleHog would emit a similar volume to SecretFinder.
3. **Parser-based scanners achieve precision through context but lose recall on minified output.** Titus reaches 78.4% precision but only 20.6% recall. The architectural pattern is consistent: AST-aware parsers reject string literals that do not appear in credential-typed contexts, which is precisely why they emit few findings overall, but it is also why they miss credentials whose context is destroyed by minification (variable names mangled, key-value pairs collapsed into runtime objects, base64-encoded UUIDs that defeat the cleartext UUID-v4 regex). JSluice reaches 41.1% precision and 32.0% recall, higher noise than Titus because the key-context extraction returns every UUID-shaped value bound to any key, including form-field IDs and internal element IDs that are not credentials.

SecretSifter's curated runtime architecture (anchored vendor token patterns, key-value context scanning, entropy filtering, and CDN/key-name blocklists, §8.2) sits in the precision–recall quadrant that no other measured non-LLM scanner reaches on this corpus. The runtime layer differs from static-tool capability in operational regime, addressing both halves of the precision–recall trade-off simultaneously and narrowing the gap to the LLM oracle ceiling to 0.033 in F1.

8.5.5. Per-Credential-Class Breakdown and Chain-Context Identifiers

GT-194 comprises eight strict credential classes. Below the strict total are 249 chain-completion identifiers (Azure AD App IDs, `client_ids`, `tenant_ids`, and resource URIs) that are public-by-design and therefore tracked separately rather than included in the recall denominator (§8.5.1).

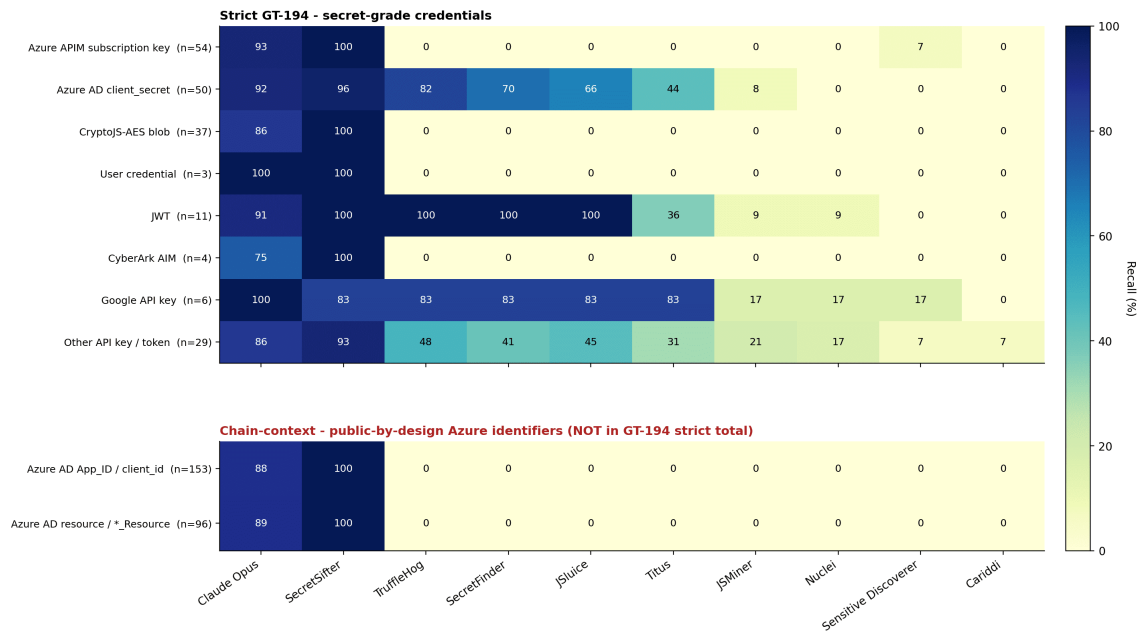


Figure 7. Per-credential-type recall heatmap on GT-194. Top panel: strict GT-194 secret-grade credentials. Bottom panel (separated by a red bold header): chain-context, public-by-design Azure identifiers (NOT in the GT-194 strict total). Cells show per-type recall percentage.

Section	Type	n	Claude	SS	TH	SF	JS	Tit	JSM	Nuc	SD	Car
strict_GT	Azure APIM subscription key	54	50	54 ★	0	0	0	0	0	0	4	0
strict_GT	Azure AD client_secret (incl. btoa OAuth ×2)	50	46	48	41	35	33	22	4	0	0	0
strict_GT	CryptoJS-AES blob	37	32	37 ★	0	0	0	0	0	0	0	0
strict_GT	User credential (plaintext + btoa)	3	3	3	0	0	0	0	0	0	0	0
strict_GT	JWT	11	10	11	11	11	11	4	1	1	0	0
strict_GT	CyberArk AIM	4	3	4	0	0	0	0	0	0	0	0
strict_GT	Google API key	6	6	5	5	5	5	5	1	1	1	0
strict_GT	Other API key / token (App Insights iKey ×15, Braintree ×1)	29	25	27	14	12	13	9	6	5	2	2
STRICT GT-194 TOTAL	(unique credentials)	194	166	151	71	63	62	40	12	7	7	2
chain_context	Azure AD App_ID / client_id / tenant_id	153	135	153 ★	0	0	0	0	0	0	0	0
chain_context	Azure AD resource / *_Resource	96	85	96 ★	0	0	0	0	0	0	0	0
CHAIN-CONTEXT SUBTOTAL	(separate, public-by-design)	249	220	249 ★	0	0	0	0	0	0	0	0

Table 9. Per-credential-type recall on GT-194 with chain-context identifiers below the line. Cells show detected count / type denominator. Below-the-line entries are excluded from the strict GT-194 recall denominator because they are not credentials, but are listed because they are required to convert leaked client_secrets into valid access tokens. Column abbreviations: Claude = Claude Opus 4.7, SS = SecretSifter, TH = TruffleHog, SF = SecretFinder, JS = JSluice, Tit = Titus, JSM = JSMiner, Nuc = Nuclei, SD = Sensitive Discoverer, Car = Cariddi.

★ = SecretSifter is the only detector for that type.

For the eight lower-recall scanners, the per-type column sums match the unique-credential totals in the highlighted Strict GT-194 total row exactly. For SecretSifter and Claude Opus, the per-type sums (189 and 175 respectively) exceed the unique totals (151 and 166) because some credentials in the corpus are tagged with multiple type labels (§8.5.1). The unique credential count remains the canonical recall metric.

The chain-context subtotal, $249 / 249 = 100\%$ for SecretSifter, 0 for every other static scanner, quantifies SecretSifter’s unique contribution to chain-completion. These identifiers are not credentials and therefore not part of the recall comparison, but operationally they are required to convert a leaked Azure AD client_secret into a working access token. Without surfacing them, a leaked secret cannot be reduced to an exploitable chain.

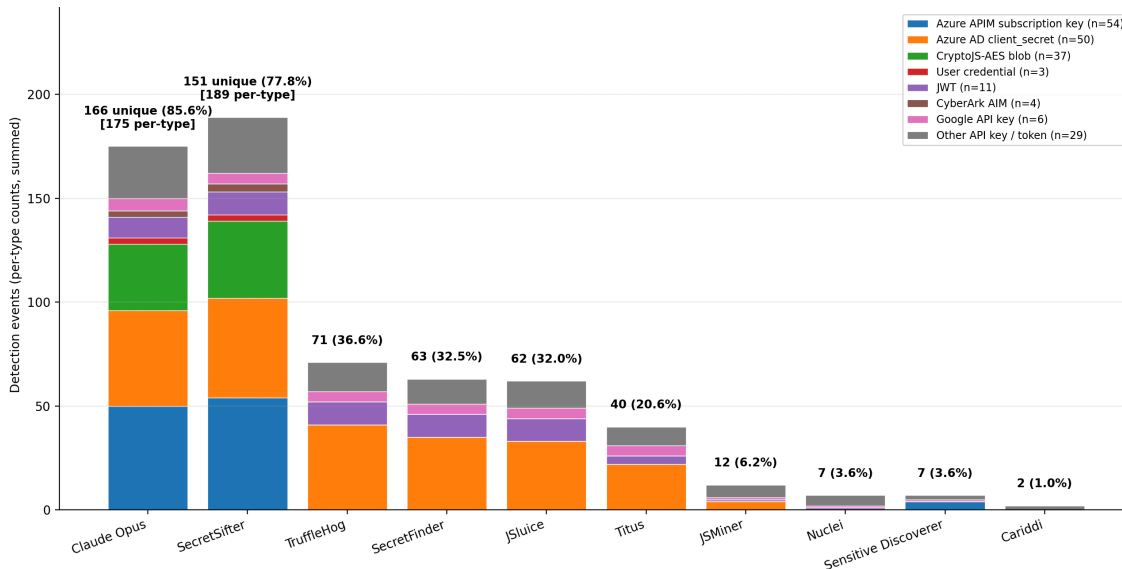


Figure 8. Per-tool detection breakdown by credential type on GT-194. Each stacked bar shows what each tool detects, decomposed by credential type. Bar heights are per-type detection events summed; the label above each bar shows the unique-credential total and recall percentage.

8.5.6. Chain-Completion in the Production Corpus

In Azure AD’s OAuth client-credentials flow, four components, client_secret, client_id (App_ID), tenant_id, and resource URI, must be combined to mint an access token. We measured how often these components are co-located in the same client-side JavaScript bundles. This is a corpus property, not a tool-comparison

metric: the question is “in how many production applications is the full token-mint chain leaked client-side?” rather than “which scanner finds the most chains?”

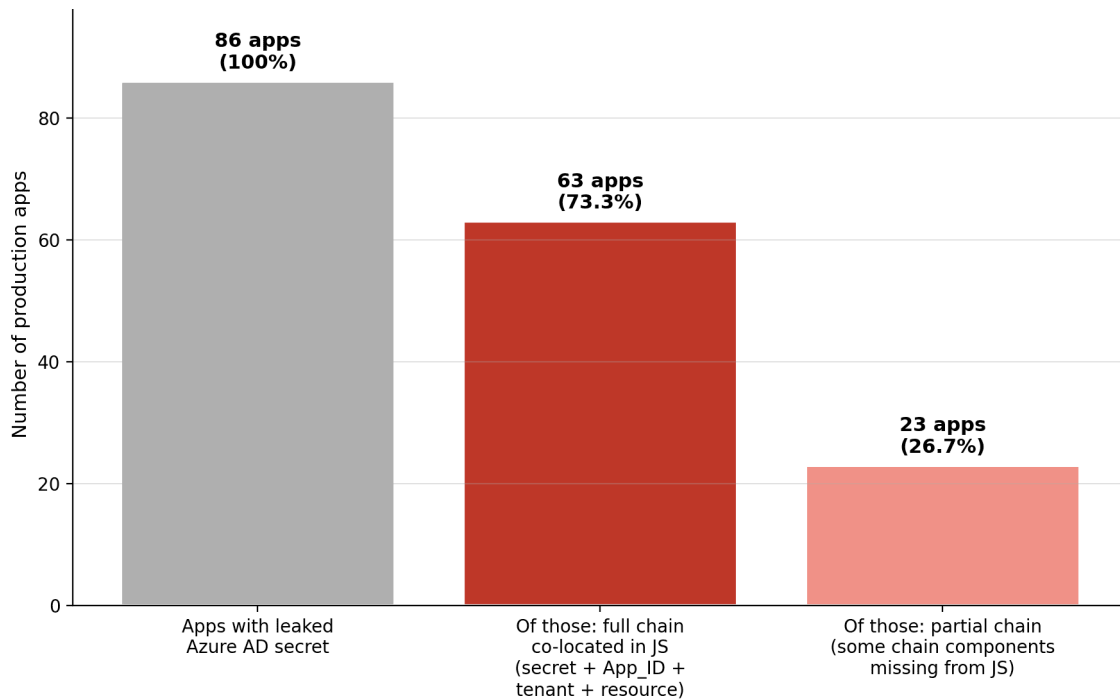


Figure 9. Chain-completion in the production corpus. Of 86 production applications where an Azure AD secret (client_secret or APIM subscription key) is exposed in client-side JavaScript, 63 (73.3%) have all four chain components co-located in the same JS bundles, so the full token-mint chain is reachable from browser-visible code with no additional reconnaissance. The remaining 23 applications (26.7%) have a partial chain (some chain components missing from the JS).

Finding	Count	% of secret-exposed apps
Apps where Azure AD secret is exposed in client-side JS	86	100%
Apps where full token-mint chain (secret + App_ID + tenant + resource) is co-located in same JS bundles	63	73.3%
Apps where chain components are partially present (tenant or resource missing)	23	26.7%

Table 10. Chain-completion in the production corpus. Counts and percentages relative to the 86 applications where an Azure AD secret is exposed in client-side JavaScript.

This finding characterizes the corpus, not any specific scanner. It establishes that for the majority of credential-bearing applications in this engagement, the entire authentication chain, not just the secret, is reachable from passive browser inspection. An attacker who recovers the `client_secret` from a JavaScript bundle does not need to perform additional reconnaissance to identify the `App_ID`, `tenant_id`, or resource URI; all four are already co-located in the same delivered artifacts. The exploitation chains documented in Sections 3 and 4 are therefore not edge cases but representative instances of a structural pattern present in roughly three-quarters of secret-exposed production deployments.

Note on tool-comparison framing. Chain reachability is reported as a corpus property rather than as a per-tool metric. “73.3% of secret-exposed apps have a full chain leaked” is a fact about the production deployment landscape, independent of which scanner one uses. The per-tool comparison story is captured in Tables 7, 8, 9 and 11; Table 10 reports the corpus property.

8.5.7. Credential Categories No Static Scanner Detected

Three credential categories in GT-194 received zero detections from every static scanner evaluated, and a fourth was detected only by SecretSifter:

- **Azure APIM subscription key (54 instances).** SecretSifter is the only scanner that detects this class with full coverage on this corpus, surfacing 54 of 54. Sensitive Discoverer surfaces 4 of 54 incidentally, its `apikey:"..."` rule captures the value when it is bound to a key whose name contains the literal `apikey`

substring (the corpus has 4 such cases under names like apimSubKey); every other static scanner returned 0. The APIM key format (32-character hex paired with the Ocp-Apim-Subscription-Key header name in HTTP requests) is not represented in the standard rule libraries shipped with TruffleHog, JSluice, SecretFinder, Titus, JSMiner, Nuclei, or Cariddi.

- **CryptoJS-AES Salted__ blob (37 instances).** SecretSifter is again the only static scanner that detects this class, surfacing all 37. The CryptoJS construct is invisible to pattern-based detection by design: the credential exists only after decryption, and the decryption key is co-located with the ciphertext (CWE-321 ^[15]). Detecting this class requires identifying the *construct* (the U2FsdGVkX1-prefix base64 envelope, paired with a co-located decryption key) rather than a credential signature. The LLM oracle (Claude Opus) reaches 32 of 37 in this category, high but not complete, because the model treats some heavily-truncated blobs as obfuscated configuration rather than encrypted credentials.
- **User credential / plaintext password (3 instances).** Only Claude Opus and SecretSifter recover this class (3 of 3 each). All other scanners return 0. Plaintext password values do not match any standard credential-shape regex; they are identified only by context (key name + value adjacency) or by semantic understanding of the surrounding code.
- **CyberArk AIM coordinates (4 instances).** SecretSifter recovers all 4; only Claude Opus comes close (3 of 4). All other static scanners return 0.

Together, these four credential classes total 98 of GT-194 (50.5%). For all four classes the structural reason for the static-tool gap is the same reason no shift-left scanner detects them in production code: they were never encountered during the pattern-library calibration phase of any of the evaluated static tools, because they appear primarily in production-deployed Angular bundles, the surface those tools were not designed to scan.

Beyond the four SecretSifter-leading categories, the chain-context identifiers tracked separately in Section 8.5.5 (153 Azure AD App_IDs / client_ids / tenant_ids and 96 Azure AD resource URIs, 249 in total) are also detected only by SecretSifter (249 of 249) and Claude Opus (220 of 249); every other static scanner returns 0 on chain-context identifiers as well.

8.5.8. Validator Confidence Calibration and Sensitivity

GPT-5.5's confidence field is a self-report. To avoid over-claiming on the validator side, two thresholds are reported: any-confidence acceptance (177/194 = 91.2%) and high-confidence-only acceptance (159/194 = 82.0%). The high-confidence variant is the headline cross-vendor agreement number cited above and

in the abstract. It is the more conservative of the two, and the Brennan-Prediger $\kappa = 0.822$ is computed against this thresholded label set. A blind-extraction sensitivity analysis was also performed, in which GPT-5.5 was given the raw bundles (no Opus rationale, no field hints) and asked to extract independently. That analysis confirmed that the SecretSifter recall figure is not artifactually inflated by pattern leakage from SecretSifter into the Opus extraction prompt.

8.5.9. Statistical Significance of Pairwise Differences

McNemar's test (with Edwards' continuity correction) was computed pairwise across all 10 scanners on the per-credential GT-194 detection matrix to determine which recall differences are statistically distinguishable from sampling variation. The test conditions on the *disagreement* cells (items one tool catches and the other misses, and vice versa), discarding the both-hit and both-miss cells which are uninformative for pairwise comparison.

Tool A	Tool B	A_only	B_only	Chi-square	p-value	Sig
SecretSifter	TruffleHog	80	0	78.0	< 0.001	✓
SecretSifter	SecretFinder	88	0	86.0	< 0.001	✓
SecretSifter	JSluice	89	0	87.0	< 0.001	✓
SecretSifter	Titus	111	0	109.0	< 0.001	✓
SecretSifter	JSMiner	139	0	137.0	< 0.001	✓
SecretSifter	Nuclei	144	0	142.0	< 0.001	✓
SecretSifter	Sensitive Discoverer	144	0	142.0	< 0.001	✓
SecretSifter	Cariddi	149	0	147.0	< 0.001	✓
Claude Opus	SecretSifter	16	1	12.0	< 0.001	✓
Claude Opus	TruffleHog	96	1	91.1	< 0.001	✓
TruffleHog	SecretFinder	8	0	6.13	0.013	✓
JSluice	SecretFinder	2	1	0.0	1.000	✗
JSMiner	Nuclei	5	0	3.20	0.074	✗

Table 11. Selected pairwise McNemar tests on GT-194. Counts are unique credentials (per §8.5.1’s canonical recall metric); each cell of the 2×2 contingency table treats one row of GT-194 as one observation. *A_only* = unique GT-194 credentials the first scanner catches and the second misses; *B_only* = unique credentials the second catches and the first misses. Chi-square computed with Edwards continuity correction (df = 1). The full matrix of all 45 pairwise comparisons is provided in the supplementary materials.

Three findings emerge:

- **First, SecretSifter’s lead is robustly significant against every other tool.** All eight SS-vs-static comparisons reach $p < 0.001$, the 77.8% versus $\leq 36.6\%$ headline gap cannot be attributed to sampling variation. The Claude-vs-SecretSifter comparison also reaches $p < 0.001$ (the 16-credential lead Claude has over SS is statistically significant), confirming the LLM oracle is genuinely above SS rather than overlapping within noise.

- **Second, the static-tool field is highly stratified.** TruffleHog (36.6%) versus SecretFinder (32.5%) is significant at $p = 0.013$. SecretFinder (32.5%) versus JSluice (32.0%) is not statistically distinguishable ($p = 1.000$). The static-tool ceiling on GT-194 sits at approximately 32–37%, with TruffleHog significantly above SecretFinder/JSluice but those two tied within sampling error.
- **Third, the bottom of the ranking is also stratified.** JSMiner (6.2%) versus Nuclei (3.6%) is not statistically distinguishable ($p = 0.074$), but both are significantly below the top static cluster. Nuclei versus Cariddi (1.0%) is also not distinguishable ($p = 0.074$); the data does not support reading those two as significantly different from each other on this corpus. Sensitive Discoverer (3.6%) ties Nuclei on aggregate recall but operates over a different subset of GT-194 (4 plaintext APIM keys vs Nuclei's 1 Google API + 1 JWT + 5 Other API tokens), McNemar SD vs Nuclei is not significant (chi-square ≈ 0.07 , $p \approx 0.79$).

All hit counts in this table are computed against the unique-credential GT-194 detection matrix (one observation per unique credential), so the per-row arithmetic $|A_{\text{only}} \cup B_{\text{only}} \cup \text{overlap}|$ reconciles directly with the unique-credential totals in Table 7 (Claude 166, SS 151). The per-type detection-event totals (Claude 175, SS 189) reported in §8.5.1 are not used here, because McNemar requires a single-class observation unit and the per-type tagging of multi-type credentials would otherwise inflate the disagreement cells.

8.5.10. Tool-Coverage Overlap and the Tool-Agnostic Blind Spot

Beyond pairwise comparisons, the question of *which* credentials each tool catches and how detection sets overlap is shown in Figure 10 below as an UpSet-style plot ^[22]. UpSet generalizes the Venn diagram to arbitrary numbers of sets and is the established replacement when more than four sets are compared.

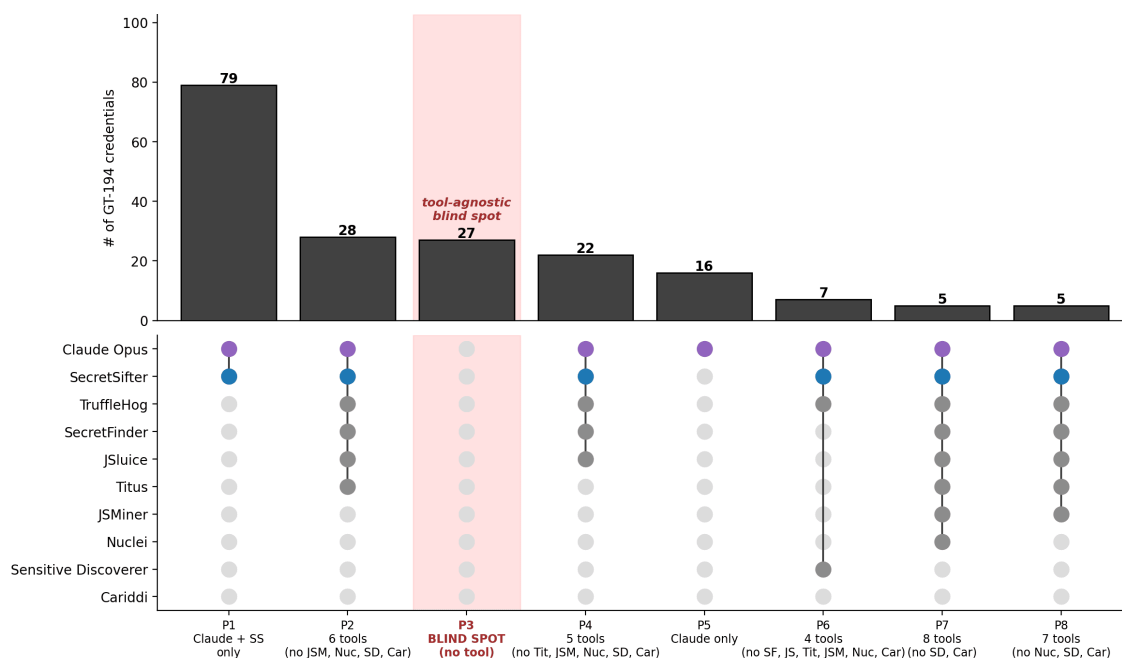


Figure 10. Detection-set overlap on GT-194 (UpSet-style). Each column is a unique detection signature – a unique combination of tools that jointly detect a credential. The bar at the top shows how many of the 194 GT credentials match that signature. The dots at the bottom show which tools participate in the signature (filled coloured dot = tool detects credentials in this pattern; gray dot = it does not). The blind-spot column (highlighted red) has zero filled dots because no tool detects those 27 credentials – this is the 13.9% tool-agnostic blind spot.

The most prominent patterns in GT-194 are: (P1) 79 credentials detected by Claude Opus + SecretSifter only, (P2) 28 credentials detected by 6 tools (no JSMiner, Nuclei, Sensitive Discoverer, Cariddi), (P3) **27 credentials detected by no tool, the tool-agnostic blind spot**, (P4) 22 credentials detected by 5 tools, and (P5) 16 credentials detected only by Claude Opus. The pattern P3 is the operationally critical observation: 13.9% of GT-194 is not findable by any combination of the ten scanners evaluated.

A complementary view is the cumulative recall when scanners are run in combination. Figure 11 below shows what fraction of GT-194 is recovered when scanners are added one at a time, with each next scanner chosen to maximize the new (previously-uncovered) credentials caught.

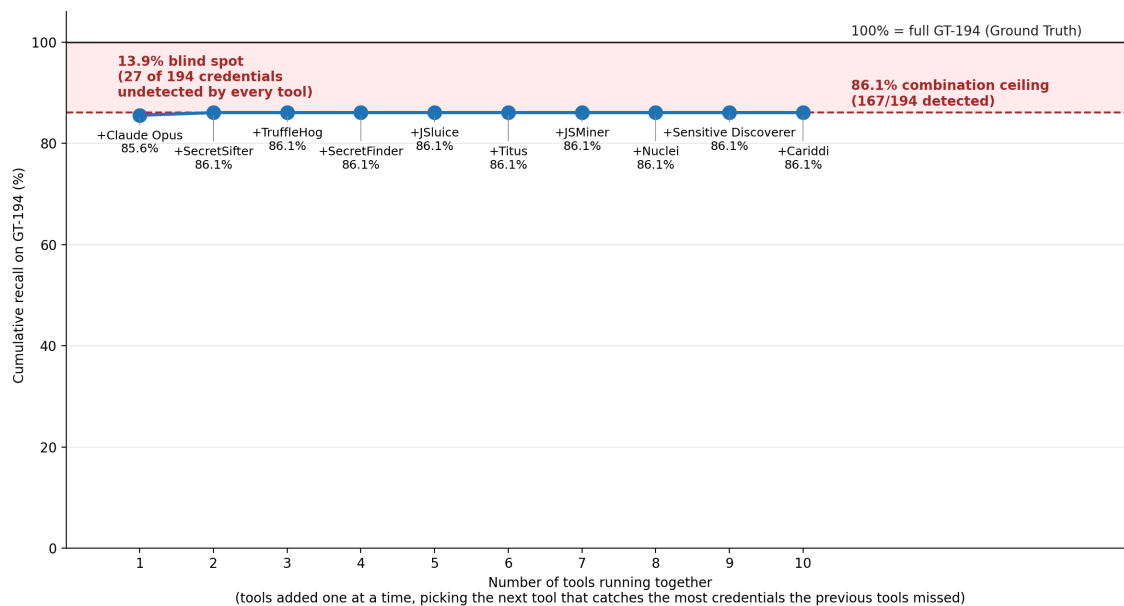


Figure 11. Coverage saturation. Tools are added one at a time, picking the next tool that catches the most credentials the previous tools missed. Even after running every scanner together, the curve plateaus at 86.1% (167 of 194 detected) – leaving 27 of 194 GT credentials (13.9%) that no tool detects. This tool-agnostic blind spot is the core motivation for runtime credential exposure research beyond static scanning.

Three structural observations:

1. **The tool-agnostic blind spot is real, sized, and stable across measurement angles.** 27 of GT-194 (13.9%) credentials are missed by every scanner. This is visible from the UpSet plot (Figure 10, the all-empty column) and from the coverage-saturation curve (Figure 11, the dashed line plateau at 86.1%). Both numbers derive from the same underlying detection matrix and are guaranteed consistent. The blind spot is dominated by CryptoJS-AES blobs that the LLM oracle did not surface during extraction, plus a small number of Other API key/token instances with non-standard shapes.
2. **SecretSifter's lead comes from disjoint findings, not from a superset relationship to static tools.** Pattern P1 (79 credentials caught by Claude + SS only) and pattern P5 (16 caught by Claude only) together account for 95 of the 167 detectable credentials, meaning 95 of 167 credentials are caught only by the LLM oracle and SecretSifter, not by any static tool. Among the static tools, intersection patterns dominate over exclusive contributions: TruffleHog and SecretFinder rarely catch a credential that SecretSifter misses. This addresses the structural concern that a curated tool's

headline recall might reflect overlap with established detectors rather than additive coverage. On GT-194, SecretSifter's contribution is primarily additive, not redundant.

3. Adding more scanners delivers diminishing returns past two. Figure 11 shows that after Claude Opus (85.6%) and SecretSifter (+0.5 percentage points to 86.1%), no additional scanner adds any new detections. Tools 3 through 9 contribute 0 marginal coverage because every credential they catch is already in the union of Claude + SS. This is a direct empirical claim about the runtime credential exposure detection landscape: in 2026, the operationally meaningful question is not which static scanner to choose but whether one is willing to invest in either an LLM oracle or a runtime-aware curated scanner. Static tools are complementary to each other only on the easy targets; on the structural blind spots (Azure AD client_secret, APIM keys, CryptoJS blobs), they all fail together.

8.6. Comparison with Concurrent Population-Scale Measurement

Demir et al. ^[2] reported the first population-scale dynamic measurement of credential exposure on the rendered web, covering 10M HTTP Archive landing pages, 1,748 verified credentials, and 14 vendor-API-verifiable service classes. Their work and ours measure different properties of the same exposure surface. Demir et al. establish population-scale prevalence on credential classes whose vendors publish verification endpoints. This work establishes per-tool detection performance at enterprise depth on eight credential classes that lack public verification endpoints, including the CryptoJS-encrypted-configuration construct that anchors the 13.9% tool-agnostic blind spot identified above. Demir et al. uniquely contribute population prevalence at 10M-page scale, longitudinal persistence (mean twelve months exposure), vendor-API verification across fourteen service classes, and remediation-outcome measurement across 2,435 notified entities. This work uniquely contributes a per-tool benchmark across ten scanners on a locked ground truth, cross-vendor LLM-validated ground-truth construction ($\kappa = 0.822$), pairwise McNemar significance testing (all static-vs-runtime gaps $p < 0.001$), chain-completion analysis showing 73.3% of affected applications co-locate the full Azure AD token-mint chain client-side, and structural-class detection of the CryptoJS-AES construct. Figure 12 visualizes the complementary detection scope at the credential-class level. Both studies reach the same systemic conclusion. Credential exposure on the rendered web is widespread, persistent, and structurally undetected by repository-only scanning.

Figure 12. Credential-class detection scope by methodology. Left column: Demir et al. [31] verified-credential counts per category, normalized to 0-100. Right column: number of credentials of each class in the locked GT-194 benchmark (n in benchmark). Empty cells indicate absence from the corresponding study. The diagonal pattern illustrates the complementarity: vendor-API verification reaches the population-scale services in the upper rows; multi-tool detection plus an LLM-anchored ground truth reaches the structural and enterprise classes in the lower rows.

Figure 12. Credential-class detection scope by methodology. Left column: Demir et al. [31] verified-credential counts per category, normalized to 0-100. Right column: number of credentials of each class in the locked GT-194 benchmark (n in benchmark). Empty cells indicate absence from the corresponding study. The diagonal pattern illustrates the complementarity: vendor-API verification reaches the population-scale services in the upper rows; multi-tool detection plus an LLM-anchored ground truth reaches the structural and enterprise classes in the lower rows.

9. Recommendations

9.1. Immediate, Days

Rotate all exposed credentials immediately, across all environments. AppKeys, SubscriptionKeys, and any other credentials found in client-side code should be treated as compromised and rotated before any other remediation step [23]. Rotation does not confirm they were abused, but it closes the window. Critically, rotation must be applied simultaneously across production, UAT, SIT, and pre-production environments. Because credentials are injected at build time, every environment that ran a build received the same secret. Remediating production while leaving UAT exposed means an attacker who extracted credentials from a lower environment retains access. Remediation is only complete when the credential is removed from the build pipeline and rotated everywhere at once.

Audit APIM subscription key product scope. Each exposed subscription key maps to a product in APIM containing one or more APIs. Access the APIM portal and identify exactly which APIs each key grants access to. This determines the blast radius and informs incident response scope.

Review Azure AD app registration permissions. Check whether the application's app registration uses Application permissions (acting as the application itself, no user context) or Delegated permissions

(acting as the signed-in user). Application permissions combined with a client secret in browser code grant the ability to access all users' data, not just the current user's. Remove Application permissions that are not strictly required.

Enable APIM anomaly alerting. Check APIM Analytics for anomalous usage of the exposed subscription key, off-hours access, high-volume calls, unexpected geographic sources. Check Azure AD sign-in logs for client_credentials grants originating from browser IP ranges. This flow is designed for server-to-server communication and should not be initiated from end-user browsers regardless of application architecture.

Disable source maps in production builds. Set sourceMap: false in angular.json under the production configuration. Do not deploy.map files alongside production bundles.

9.2. Short-Term, Weeks

Implement the Backend for Frontend (BFF) pattern. The correct long-term architecture removes all service credentials from the browser entirely [24].

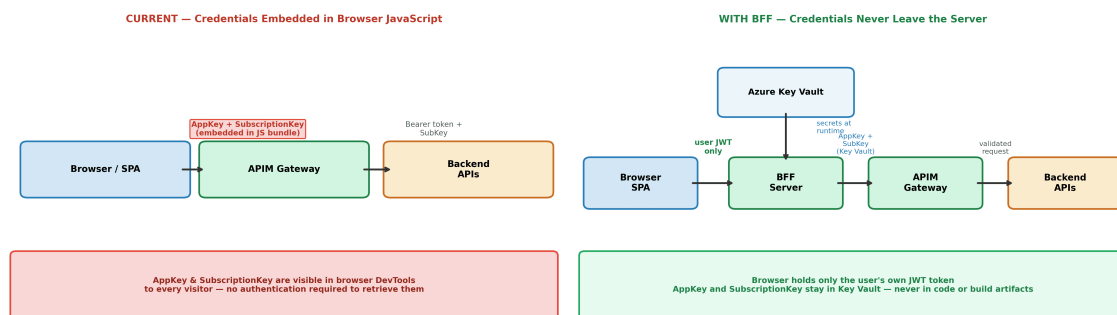


Figure 13. Current architecture (left) versus the Backend for Frontend pattern (right). In the current architecture, AppKey and SubscriptionKey are embedded in the browser JavaScript bundle. The BFF pattern moves all service credentials to a server-side proxy that holds them in Azure Key Vault, so the browser holds only the user's own JWT.

The BFF stores AppKey and SubscriptionKey in Azure Key Vault or App Service configuration, never in code or environment files that reach the build artifact. It receives the user's Azure AD JWT, validates it, and makes downstream calls using its own server-side credentials. The browser never holds anything other than the user's own access token.

Implementation options by effort level:

Option	Effort	Notes
Azure Function proxy	Low	Lightweight, scales automatically, fits alongside existing Azure infrastructure
Node.js / Express proxy	Low-Medium	Familiar to Angular teams, can share TypeScript types
APIM validate-jwt policy	Low	APIM validates user JWT and rejects requests without one, SubscriptionKey removed from browser entirely

Table 12.

Correct the service principal scope. Replace Application permissions with Delegated permissions where possible. An application acting on behalf of a user should see only what that user is authorized to see, not every user's data. This is the configuration that converted a credential exposure into an account takeover chain in both cases documented above.

Restrict MSAL.js token caching to in-memory storage. The MSAL.js default caches tokens in localStorage, which is accessible to any JavaScript executing on the page, including XSS payloads [25]. Configuring cacheLocation: "memory" limits token extraction risk at the cost of tokens being lost on page refresh.

Audit CORS policy on APIM. Development CORS policies allowing wildcard origins (*) with credentials enabled are commonly left open in production. With exposed credentials and permissive CORS, a malicious third-party site can make authenticated API calls using a visiting user's session. Lock CORS to specific, explicitly allowed origins.

For non-Azure stacks: The same architectural principle applies across providers. AWS applications should move Cognito client secrets and API Gateway keys to Lambda or ECS task roles using IAM instance profiles, never into the React bundle. GCP applications should replace Firebase service account keys in the browser with Firebase App Check and server-side token exchange. SaaS credentials (Twilio,

Stripe, SendGrid) should be proxied through a server-side endpoint in all cases, no SaaS secret key belongs in client-side JavaScript regardless of platform.

9.3. Ongoing, Process

Add artifact scanning to the CI/CD pipeline before deployment. This closes part of the build-time injection gap by scanning the build artifact before it reaches production. It catches pipeline-substituted secrets that materialized in dist/ but does not cover runtime-fetched configurations. The scanning step should be inserted after the build stage and before the deployment stage, scanning the compiled output directory rather than source files:

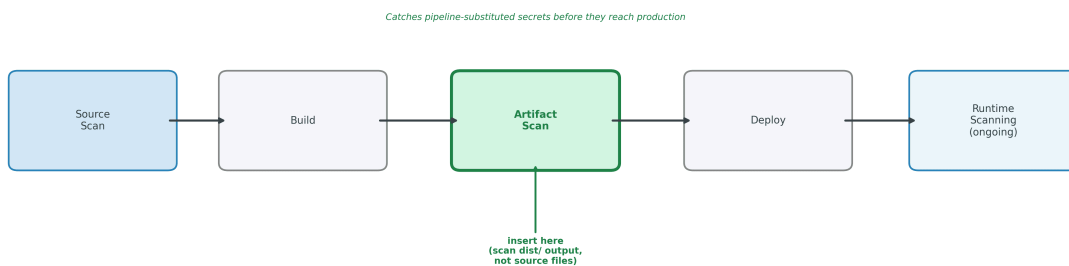


Figure 14. Recommended CI/CD pipeline with artifact scan stage inserted between Build and Deploy. The artifact scan targets the compiled output directory (e.g. dist/), not source files. Secrets introduced by pipeline variable substitution are caught before reaching production.

The artifact scan step targets the compiled output directory (e.g. dist/), not source files. These are different scan targets. On secrets found, the pipeline should fail and block deployment. Any secret scanning tool with filesystem scanning capability is applicable.

Monitor for abuse of already-exposed credentials. Rotation confirms exposure is closed. It does not confirm the credentials were not already used. After rotation:

- Filter APIM Analytics by the rotated subscription key for historical usage anomalies
- Search Azure AD sign-in logs for client_credentials grants originating from browser IP ranges. This flow is designed for server-to-server communication and should not be initiated from end-user browsers
- Alert on any usage of a rotated key post-rotation. This is an active attacker with a cached credential
- Azure Sentinel can correlate APIM logs with Azure AD sign-in anomalies for sustained monitoring

Eliminate server-side secrets with Azure Managed Identity. For the BFF or backend layer, Managed Identity removes the need to store or rotate AppKey entirely [\[26\]](#).

Azure App Service (BFF)

- + - Managed Identity enabled
- + - No credentials in code, config, or pipeline
- + - Azure handles authentication transparently
- + - BFF calls APIM / Key Vault / Storage with zero stored secrets

Combined with Key Vault references in App Service configuration for any remaining secrets, this eliminates the credential rotation problem for the server-side layer.

Add periodic runtime scanning of live deployed applications. Artifact scanning before deployment does not cover runtime-fetched configurations, lazy-loaded chunks served dynamically from CDN, or credentials embedded in third-party scripts that never touch the build pipeline. Periodic scanning of the live application closes this remaining gap. Tools designed for this layer passively monitor live HTTP traffic (JavaScript bundles, HTML source, JSON and XML responses, request headers) and flag credentials as they flow, without outbound verification calls. SecretSifter (open-source, evaluated in §8.5) is one such implementation; comparable runtime-aware tooling is appropriate provided it operates on served content rather than source repositories.

10. Limitations

The exploitation chains documented in this paper are specific to Azure Active Directory and Azure API Management. The structural paths by which secrets reach production (build-time injection, CI/CD variable substitution, runtime configuration fetching, and scanner suppression) are platform-agnostic and appear across AWS, GCP, and SaaS stacks, as noted in Section 5.3.1. However, the specific credential types, token endpoint formats, and remediation guidance in Sections 2 and 9 are Azure-specific and should be adapted to the relevant platform.

The empirical figures cited in Section 1 are drawn from a single authorized security engagement covering approximately 2,000 enterprise web application assets within one organization, of which 113 contained credentials in served content. The sample is not randomly selected across organizations. Clopper-Pearson exact 95% confidence intervals are reported alongside each figure: 5.65% (CI: 4.7%–6.8%) for at least one live credential across the 2,000-asset scope, and 55.8% (CI: 46.1%–65.1%) for a complete Azure

AD credential set among the 113 credential-bearing applications. Because the sample is drawn from a single organization's portfolio (predominantly Angular SPAs with a shared build pipeline) these figures should not be interpreted as a cross-industry population-level prevalence estimate. A multi-organization prospectively designed study would be required to produce generalizable estimates. Independent industry-scale measurement provides external corroboration of the underlying exposure pattern: Intruder's December 2025 sweep ^[4] reported approximately 42,000 exposed tokens across roughly 5 million applications using regex-driven static detection, a finding consistent with the prevalence direction reported here while differing in scale and detection methodology.

Section 8.5 presents a 10-scanner comparison evaluation against the locked Ground Truth GT-194, 194 unique secret-grade credentials extracted across 113 enterprise applications by Claude Opus 4.7, cross-validated by GPT-5.5 (Brennan-Prediger $\kappa = 0.822$, almost-perfect), and locked at the field-deployment register on 8 May 2026. The evaluation has five documented limitations.

1. **Single-organization corpus and author-developer role.** The application pool is drawn from one engagement within one organization (predominantly Angular SPAs with a shared build pipeline). The first author (Gorijala) is also the developer of SecretSifter, one of the evaluated scanners. The headline recall numbers in Table 7 should therefore be read as scanner performance on this specific Azure-heavy corpus, not as cross-organizational population estimates. The structural mitigations applied (independent LLM Ground Truth extraction, cross-vendor classification, exact-value substring matching with no per-tool tuning of the comparison protocol, the same 6-rule overlay applied identically to all rule-extensible tools) prevent the comparison from being engineered toward a specific tool, but they do not eliminate the corpus-specificity of the absolute numbers. A multi-organization replication on an independently constructed held-out benchmark would be required to claim cross-industry generalization, and is committed to as a direct follow-on study.
2. **Sample frame.** The 113-application benchmark subset is drawn from the same engagement that produced the Ground Truth, not from an independently sampled population. Recall and per-class results should be interpreted as measurements of scanner performance against GT-194, not as cross-organizational performance estimates. The ground-truth construction methodology (independent LLM extraction with cross-vendor validation) is the contribution that generalizes. The recall figures are anchored to this corpus.
3. **Tightened-configuration scope.** The 6-rule overlay applied to the rule-extensible tools (TruffleHog, SecretFinder, Titus, JSluice) was sized to cover the most prevalent credential shapes in this corpus

and applied identically across all four tools (Appendix A). A more aggressive tuning frontier, full vendor-rule libraries, customer-specific patterns, or per-tool optimization, may further close the static-tool gap. The point of the comparison is not that 6 rules are enough; it is that even a corpus-targeted overlay applied uniformly leaves a residual gap of 41 percentage points to a runtime-aware scanner, on this corpus.

4. LLM-oracle limitations. The LLM oracle is itself a tool with its own biases. Claude Opus did not surface 5 of 37 CryptoJS-AES blobs and 18 of 153 Azure AD App IDs in the chain-context tier, demonstrating that the LLM oracle has structural blind spots that curated detection rules complement. The 13.9% tool-agnostic blind spot identified in §8.5.10 includes credentials that the LLM oracle missed, so the gap is a property of the underlying credential corpus rather than an LLM-vs-static comparison artefact.

5. Per-credential vs per-application generalization. GT-194 is a per-credential ground truth; the chain-completion finding in §8.5.6 is a per-application corpus property derived from 86 secret-exposed applications. The 73.3% (63/86) figure is descriptive of this engagement, not inferential of cross-industry chain-completion rates. A multi-organization replication is required for inferential claims about full-chain reconstruction rates.

11. Future Work

Multi-organization prevalence study. The 5.65% credential exposure rate reported in this paper is drawn from a single organization's application portfolio. A prospectively designed study across multiple organizations (stratified by industry, stack, and deployment maturity) is required to produce a generalizable prevalence estimate. A multi-organization replication study, comparable in scope to the 2,000-application engagement reported here, is planned as a direct follow-on.

Cross-organization precision benchmark. §8.5.4 reports direct precision, recall, and F1 for all ten scanners on a synthetic-portal validation pass against the GT-194 corpus. Replicating the same precision measurement on a held-out, multi-organization production corpus, with full TP / FP classification of every emitted finding by an independent validator, would test whether the precision ordering reported here transfers across stacks. The qualitative ordering (parser-based scanners produce few false positives, regex-based scanners produce many, SecretSifter's noise-suppression layer materially differentiates it from comparable static tools) is robust under any reasonable re-measurement, but the exact precision and F1 numbers per scanner would tighten.

Multi-organization held-out benchmark. GT-194 is constructed from a single engagement. Replicating the same construction methodology on a held-out benchmark drawn from multiple unrelated organizations would test whether the recall numbers reported here transfer across stacks, build pipelines, and credential conventions. The construction methodology (Claude Opus extraction + GPT-5.5 cross-vendor validation) is fully described in §8.5.1 and is reproducible by any practitioner with API access to both vendors.

Runtime-fetched configuration coverage. GT-194 evaluates scanners against downloadable JavaScript bundles only, the surface for which static-file scanners can be applied uniformly. Modern SPAs load credentials via runtime-fetched configuration endpoints (`/config.json`, `/api/settings`) and HTML SSR state blobs that are never present in the initial bundle. Extending the benchmark methodology to runtime-fetched and SSR-injected credentials is the natural next step and would close the gap between the bundle-scanning protocol described here and the production-traffic surface that real attackers actually inspect.

Automated exploitation chain validation. The current study manually confirmed exploitability for both documented chains. Automated scope-aware validation, checking whether exposed Azure AD credentials have `client_credentials` grant enabled, or whether exposed APIM keys map to APIs with sensitive data, would allow prioritization of findings at scale without manual verification per application.

Longitudinal remediation tracking. Both applications documented in this paper remediated within the assessment window. Whether organizations with incidentally discovered runtime credential exposure (found via bug bounty or passive scanning) remediate at comparable rates is unknown. A longitudinal study tracking time-to-remediation across disclosure channels would inform remediation guidance.

12. Responsible Disclosure

Both exploitation chains documented in this paper were identified during authorized security assessments conducted within the scope of formal engagements. In both cases:

- Findings were reported immediately to the affected organization upon identification
- Proof-of-concept evidence was limited to what was necessary to confirm exploitability, no bulk data extraction occurred
- Remediation was implemented by the affected organization
- Findings were verified through retest to confirm remediation before this publication

No user data was retained beyond the assessment period. No credentials were used outside the scope of confirming exploitability. Organization names and application identifiers have been anonymized throughout this paper.

13. Conclusion

The shift-left investment the security industry has made is real, justified, and valuable. Catching secrets before they reach production is always preferable to finding them after. GitLeaks, TruffleHog, GitHub Advanced Security, and the SAST tools deployed in enterprise CI/CD pipelines do exactly what they are designed to do.

But secrets are still reaching production, not because these tools fail, but because the path from development to deployment has branches that no pre-deployment scanner covers. Build-time environment injection, CI/CD pipeline variable substitution, and runtime configuration fetching each produce secrets in production that never existed in any repository at any point. The tools designed to prevent credential exposure scan a layer that those secrets never passed through.

The exploitation chains documented above are not theoretical edge cases. They are patterns found repeatedly in real production applications serving real users, applications that had passed every security review and had shift-left tooling in place. None of those tools provided automated visibility into what each application served, or automated detection when credentials appeared in delivered JavaScript.

The shift-right tooling gap is now quantified, and includes a tool-agnostic blind spot. On the locked Ground Truth GT-194, 13.9% of credentials are missed by every one of the ten scanners evaluated, predominantly the CryptoJS encrypted-configuration class that is invisible to pattern-based detection by design. The combined coverage of all ten scanners run together plateaus at 86.1%, leaving that 13.9% as a structural property of the credential surface itself rather than a tooling deficit that more rules can close. Beyond detection, the chain-completion finding (73.3% of affected applications co-locate the full Azure AD token-mint chain client-side) shows that the credentials reachable in served JavaScript are operationally weaponizable from browser-visible code alone. Industry-scale measurement corroborates this gap: Intruder's December 2025 sweep identified approximately 42,000 exposed tokens across roughly 5 million applications using regex-driven static detection alone ^[4], a lower-bound observation that the pattern documented here is widespread at industrial scale. The shift-left layer is covered. The runtime layer is not. That asymmetry, and the additional 13.9% blind spot that no scanner closes, is what is being exploited.

Closing it requires the same community effort that built shift-left tooling into what it is today, practitioners documenting the gap, security teams demanding runtime scanning in their programs, and tool authors building for the layer that has been ignored. Bug bounty researchers finding runtime credentials should escalate beyond the credential itself and document the full exploitation chain. Penetration testers should treat runtime JavaScript as a first-class target surface, not an afterthought. Security teams should ask their tooling vendors a direct question: does your scanner cover what the application serves, or only what developers commit?

The patterns in this paper will not stop appearing until the runtime layer gets the same tooling investment the repository layer already has.

Appendix A. Tightened-Configuration Rule Set

The four rule-extensible scanners evaluated in §8.5 (TruffleHog, SecretFinder, Titus, JSluice) were each run with the same 6-rule overlay. The rules cover the credential shapes most prevalent in GT-194. The full rule set is reproduced below verbatim. The same overlay was applied identically to all four tools; no per-tool tuning was performed.

```
- id: custom.azure.client_id
  name: Azure AD Client ID (UUID v4)
  pattern: '\b[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}\b'

- id: custom.azure.ad_secret
  name: Azure AD Client Secret (tilde-format, 34-40 char)
  pattern: '\b[A-Za-z0-9_.\-]{1,16}~[A-Za-z0-9_.\-~]{16,40}\b'

- id: custom.azure.apim_key
  name: Azure APIM Subscription Key (32-hex)
  pattern: '\b[0-9a-f]{32}\b'

- id: custom.jwt.token
  name: JSON Web Token (eyJ-prefix three-segment)
  pattern: '\beyJ[A-Za-z0-9_.\-]{10,}\. [A-Za-z0-9_.\-]{10,}\. [A-Za-z0-9_.\-]+'

- id: custom.azure.app_insights_ikey
  name: App Insights Instrumentation Key
  pattern: 'InstrumentationKey=[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}'
```

```
- id: custom.cryptojs.salted  
  name: CryptoJS U2FsdGVkX1-prefix encrypted blob  
  pattern: 'U2FsdGVkX1[A-Za-z0-9+/=]{16,}'
```

These six rules were selected from the credential shapes most prevalent in GT-194, derived without consulting any tool's output. They are not a comprehensive coverage set; they are a minimal corpus-targeted overlay applied identically to **all four** rule-extensible tools (TruffleHog, SecretFinder, Titus, JSluice) so that the comparison reflects scanner-architecture differences rather than rule-library differences. JSluice was run with the same six patterns translated to the JSluice user-pattern JSON schema; the built-in JSluice secret rules silently produce zero output on minified webpack bundles in this corpus, and the user-pattern overlay is what allows JSluice to surface the key-value extractions that drive its reported recall. JSMiner exposes no user-rule API in the Burp extension distribution, so it was reimplemented in Python as a faithful clone of the published Trustwave SpiderLabs detection logic (21 keyword anchors plus Shannon-entropy scoring with the same Firm/Tentative classification thresholds) and run on the GT-194 corpus. Nuclei was run in the URL-mode JavaScript-spider configuration matching the Intruder December 2025 industry-scale measurement ^[4]: bundles served via a local Python HTTP server on port 9101, scanned with the secret-scanning template categories `http/exposures/tokens`, `http/exposures/apis`, and `http/exposures/keys` plus the `secret/token/key/exposure` template tags. The file-mode Nuclei configuration produces zero findings on `raw.js` files because Nuclei's secret-scanning template library is HTTP-response-oriented rather than file-content-oriented, a methodology mismatch documented here for reproducibility. Cariddi was run at its built-in default secret-detection configuration without any rule overlay (its primary mode is web-crawling and endpoint enumeration, with secondary secret detection). SecretSifter was run at its built-in configuration with the field-deployment scan output exported via the SecretSifter REST API.

Notes

Preprint (v2), author's version. An earlier, shorter version was posted at Zenodo (DOI: 10.5281/zenodo.19464446). This version adds the GT-194 benchmark and the ten-scanner evaluation and is under review at IEEE Access.

About the Author

Hemanth Gorijala is an application security professional and penetration tester with 13 years of experience. He conducts web application security assessments and is the author of SecretSifter.

Statements and Declarations

Funding

No funding was received for this research.

Potential Competing Interests

The author is the developer of SecretSifter, one of the ten scanners evaluated in Section 8.5. To prevent this from biasing the evaluation, the GT-194 ground truth was constructed by Claude Opus 4.7 (Anthropic) and independently validated by GPT-5.5 (OpenAI), so that no evaluated scanner, including SecretSifter, defined its own evaluation set (Section 8.5). SecretSifter is free and open source. The exploitation chains, structural gap analysis, and prevalence figures are independent of that tool and predate its development. The author declares no other competing interests.

Ethics

This study did not involve human participants, human-derived tissue, or personal health data, and therefore did not require institutional review board (IRB) or ethics-committee approval. Both exploitation chains were identified during authorized security assessments conducted within the scope of formal, written engagements (Section 1.1). No systems were accessed outside the authorized engagement scope. Proof-of-concept activity was limited to what was necessary to confirm exploitability; no bulk data extraction was performed, and no end-user personal data was retained beyond the assessment period (Section 12).

Consent for publication

Not applicable. No individual personal data are reported. Organization names and application identifiers have been anonymized throughout.

Data Availability

The GT-194 benchmark is derived from live production credentials belonging to a single organization and cannot be publicly redistributed for security and confidentiality reasons. The SecretSifter detection engine is open source under the MIT license (<https://github.com/secretsifter/secretsifter-burp>). A publicly available demonstration environment containing only synthetic credentials (InsecureShield) reproduces the constructs described in this paper.

Authors' Contributions

H.G. is the sole author and conducted all research, analysis, and writing.

Use of Generative AI

The author used Claude (Anthropic) to assist with drafting and editing the manuscript text. The author reviewed and takes full responsibility for all content. No generative AI was used to produce the research, the exploitation chains, the benchmark, or the analysis.

References

1. ^{a, b, c}Meli M, McNiece MR, Reaves B (2019). "How Bad Can It Git: Characterizing Secret Leakage in Public GitHub Repositories." *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*. <https://www.ndss-symposium.org/ndss-paper/how-bad-can-it-git-characterizing-secret-leakage-in-public-github-repositories/>.
2. ^{a, b}Saha A, et al. (2023). "A Comparative Study of Software Secrets Reporting by Secret Detection Tools." *arXiv preprint arXiv:2307.00714*. <https://arxiv.org/abs/2307.00714>.
3. ^{a, b, c, d, e}Demir N, Vekaria Y, Smaragdakis G, Durumeric Z (2026). "Keys on Doormats: Exposed API Credentials on the Web." *arXiv preprint arXiv:2603.12498*. <https://arxiv.org/abs/2603.12498>.
4. ^{a, b, c, d}Intruder Security Ltd. (2025). "Industry-scale Scanning Reveals Tens of Thousands of Tokens Exposed in JavaScript Bundles." *industry research report*.
5. ^AQiu X, et al. (2025). "The Skeleton Keys: A Large Scale Analysis of Credential Leakage in Mini-Apps." *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*. <https://www.ndss-symposium.org/wp-content/uploads/2025-273-paper.pdf>.

6. [△]Khodayari S, et al. (2023). "An Empirical Study of JavaScript Bundling on the Web and its Security Implications." CISPA Helmholtz Center for Information Security. <https://publications.cispa.saarland/4036/1/Bundlers-Study-Submission-camera-ready.pdf>.
7. [△]Lauinger T, et al. (2017). "Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web." Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS). <http://www.ndss-symposium.org/wp-content/uploads/2017/09/ndss201702B-1-Lauinger-paper.pdf>.
8. [△]Alqahtani H, et al. (2023). "A Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages." arXiv preprint arXiv:2301.05097. <https://arxiv.org/pdf/2301.05097>.
9. [△]Mesbah A, et al. (2020). "On Detecting and Measuring Exploitable JavaScript Vulnerabilities." ACM Trans Priv Secur. <https://users.encs.concordia.ca/~mmannan/publications/JS-vulnerability-TOPS.pdf>.
10. [△]Microsoft Documentation (2023). "Azure API Management Subscription Keys." Microsoft Documentation. <https://learn.microsoft.com/en-us/azure/api-management/api-management-subscriptions>.
11. ^{a, b, c}Hardt D (2012). "OAuth 2.0 Authorization Framework (RFC 6749)." Internet Engineering Task Force. Sections 2.1 (Client Types) and 4.4 (Client Credentials Grant). <https://www.rfc-editor.org/rfc/rfc6749>.
12. ^{a, b}Microsoft Documentation (2023). "Microsoft Identity Platform (Client Credentials Flow)." Microsoft Documentation. <https://learn.microsoft.com/en-us/entra/identity-platform/v2-oauth2-client-creds-grant-flow>.
13. ^{a, b}OWASP Foundation (2021). "OWASP Top 10:2021 (A05: Security Misconfiguration)." OWASP Foundation. https://owasp.org/Top10/A05_2021-Security_Misconfiguration/.
14. [△]OWASP Foundation (2021). "OWASP Top 10:2021 (A02: Cryptographic Failures)." OWASP Foundation. https://owasp.org/Top10/A02_2021-Cryptographic_Failures/.
15. ^{a, b}MITRE Common Weakness Enumeration (2023). "CWE-321: Use of Hard-coded Cryptographic Key." MITRE Common Weakness Enumeration. <https://cwe.mitre.org/data/definitions/321.html>.
16. [△]webpack Documentation (2023). "webpack DefinePlugin." webpack Documentation. <https://webpack.js.org/plugins/define-plugin/>.
17. [△]Turan MS, Barker E, Kelsey J, McKay K, Baish M, Boyle M (2018). "Recommendation for the Entropy Sources Used for Random Bit Generation." National Institute of Standards and Technology. doi:10.6028/NIST.SP.800-90B.
18. [△]ISO/IEC (2017). "Luhn Algorithm (ISO/IEC 7812-1:2017, Identification cards)." Identification of issuers.
19. [△]Anthropic (2026). "Claude 4 Model Card." Anthropic. <https://www-cdn.anthropic.com/claude-4-model-card.pdf>.
20. [△]OpenAI (2026). "GPT-5.5 System Card." OpenAI. <https://cdn.openai.com/gpt-5-5-system-card.pdf>.

21. [△]Brennan RL, Prediger DJ (1981). "Coefficient kappa: Some uses, misuses, and alternatives." *Educ Psychol Meas.* **41**(3):687–699.
22. [△]Lex A, Gehlenborg N, Strobel H, Vuilleumot R, Pfister H (2014). "UpSet: Visualization of Intersecting Sets." *IEEE Trans Visual Comput Graphics.* **20**(12):1983–1992. doi:[10.1109/TVCG.2014.2346248](https://doi.org/10.1109/TVCG.2014.2346248).
23. [△]National Institute of Standards and Technology (2020). "NIST SP 800-53 Rev. 5 (Security and Privacy Controls: IA-5 Authenticator Management)." National Institute of Standards and Technology. <https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final>.
24. [△]Newman S (2015). "Backends For Frontends." samnewman.io. <https://samnewman.io/patterns/architectural/bff/>.
25. [△]Microsoft Authentication Library for JavaScript (2023). "Microsoft MSAL.js Token Caching." Token cache documentation. <https://learn.microsoft.com/en-us/azure/active-directory/develop/msal-js-token-cache>.
26. [△]Microsoft Documentation (2023). "Azure Managed Identity Overview." Microsoft Documentation. <https://learn.microsoft.com/en-us/entra/identity/managed-identities-azure-resources/overview>.

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.