Overcoming Scanner Blindness: A Clean-Slate, Template-Driven DAST Framework (Case Study: OWASP Juice Shop)

Tudor Cristian Lacatus Cosma

***Abstract*—The transition from Multi-Page Applications (MPAs) to Single Page Applications (SPAs) has rendered traditional crawling methodologies insufficient, creating a phenomenon known as "scanner blindness." Conventional Dynamic Application Security Testing (DAST) tools struggle to discover attack surfaces generated dynamically via client-side execution because they historically rely on static HTML parsing rather than DOM evaluation. This study proposes a template-driven DAST framework that combines Katana-based endpoint discovery (JavaScript crawling and XMLHttpRequest (XHR) endpoint extraction) with YAML-defined active checks. Templates support endpoint placeholders (e.g., expanding across discovered API routes) and data-driven field-name auto-discovery patterns for common concepts such as privilege/role-like fields, tokens, and identifiers.**

***Index Terms*—DAST, Single Page Applications, Scanner Blindness, Idempotency, OWASP Juice Shop, State-Aware Scanning.**

I. Introduction

The evolution of web application development over the past two decades has been defined by a fundamental migration of logic, state, and rendering responsibilities from the server to the client. The industry has transitioned from the deterministic, synchronous nature of Server-Side Rendering (SSR) to the fluid, event-driven paradigm of Client-Side Rendering (CSR) and Single Page Applications (SPAs). While this architectural shift is often discussed in the context of Search Engine Optimization (SEO) challenges [1], it has created a parallel and critical deficit in automated security testing.

Consequently, Dynamic Application Security Testing (DAST) tools, originally architected to crawl static HTML structures and inject payloads into synchronous HTTP forms, now face a significant coverage gap described in literature as "scanner blindness." This phenomenon occurs when scanners fail to identify attack surfaces that are generated dynamically via client-side execution. For example, legacy scanners rely on static `<a href="...">` tags to discover routes; however, modern SPAs handle navigation via JavaScript event listeners (e.g., `onclick` events that mutate the Document Object Model (DOM)). Without a JavaScript execution engine, a scanner cannot trigger these transitions, effectively leaving significant portions of the application untested [2]. Modern frameworks, such as Angular, React, and Vue, present opaque surfaces to traditional tools, concealing vulnerabilities behind complex execution contexts and virtualized states.

To quantify the impact of these architectural shifts, the research community relies on empirical evaluations of tool effectiveness. Recent studies, such as those by Qadir et al. (2025), assess scanner performance by measuring the count and severity of detected vulnerabilities and mapping them to standardized industry benchmarks like the OWASP Top 10 [3]. These comparative frameworks are essential for identifying gaps in scanner coverage against real-world applications.

A. Justification of Test Environment

This project utilizes OWASP Juice Shop [4] as the primary testing environment. Juice Shop is an intentionally vulnerable web application originally developed by Bjoern Kimminich and first released in 2014 as part of the OWASP (Open Web Application Security Project) community. As of 2026, it is the most-starred OWASP project on GitHub with over 50,000 stars, reflecting widespread adoption for security training, capture-the-flag competitions, and tool evaluation. The application is actively maintained and serves as a modern benchmark for testing DAST scanners against contemporary JavaScript-based architectures.

While modern alternatives like "Broken Crystals" offer valuable benchmarks, they are often structured as collections of isolated, unrelated vulnerability test cases. In contrast, Juice Shop simulates a monolithic, interdependent e-commerce workflow where vulnerabilities are gated behind complex state transitions. In such environments, security testing frequently depends on preserving an authentication context (e.g., a bearer token) and executing request sequences in the correct order to satisfy preconditions (e.g., login before accessing protected APIs). This architectural cohesion makes Juice Shop a strong candidate for stress-testing a scanner's ability to overcome "blindness" in deep navigational structures, while also highlighting the limits of purely stateless request fuzzing.

B. Methodological Approach

Unlike general-purpose tools such as OWASP ZAP or Burp Suite, which often operate against long-lived targets and can suffer from "state pollution" (where previous actions contaminate the database for subsequent tests), this solution supports an optional "Clean-Slate" methodology when the target can be provisioned as an ephemeral lab. Research by Arcuri on automated API testing emphasizes Resetting the System Under Test (SUT), specifically the database state, before test execution to ensure scientific validity [5]. The concrete lab-mode workflow and Docker integration are detailed in Section IV-A.

C. Contributions

This work makes the following practical contributions toward mitigating scanner blindness in JavaScript-heavy web applications:

- **Clean-slate lab execution:** an optional lab mode that provisions an ephemeral Dockerized target per scan run to improve reproducibility and reduce state pollution during experiments.
- **Scaffolded app profiling:** an initialization workflow that discovers endpoints and generates an application profile to reduce the cost of onboarding new targets and authoring templates.
- **Template-driven, multi-step active checks:** YAML-defined request sequences with variable extraction/interpolation to express stateful workflows (e.g., authentication, token propagation) beyond single-request signatures.
- **Reproducible and extensible methodology:** a tiered template approach (generic baseline + app-specific overlays) intended to be easily reproduced and expanded to additional applications with minimal rework.
- **Standards-aligned reporting:** automatic mapping of findings to OWASP Top 10 (2025) [6] categories to support executive-facing communication and benchmarking.

D. Ethical Considerations and Safety

All experiments were performed exclusively against deliberately vulnerable applications in a controlled local lab environment (OWASP Juice Shop as the main target, and Broken Crystals secondly), provisioned via Docker. Potentially destructive checks were executed only in the tool's clean-slate lab mode to isolate side effects and avoid persistent state pollution across runs. No scans were conducted against third-party systems, production deployments, or targets without explicit authorization.

II. State of the Art

A. Dynamic Application Security Testing (DAST)

Dynamic Application Security Testing (DAST) is a methodology for analyzing a web application in its running state, adopting the perspective of an external attacker. Unlike Static Application Security Testing (SAST), which operates as a "white-box" technique by inspecting source code, DAST operates as a "black-box" technique [7]. It possesses no prior knowledge of the internal architecture, logic flow, or server-side code; instead, it interacts with the application via the presentation layer (HTTP/HTTPS) to observe responses to induced stimuli.

While SAST acts as a code inspector, DAST acts as a behavioral analyst. This distinction is critical for identifying runtime vulnerabilities, such as misconfigurations or authentication logic flaws, that remain invisible during static code analysis.

B. The Challenge of Single Page Applications (SPAs)

The SPA shift, introduced in Section 1, turns discovery from deterministic link-following into stateful exploration. In the literature, the core difficulty is that reachable states are not enumerated a priori; they emerge from client-side execution and user-triggered transitions.

1) The Crawling Deficit: This architecture fundamentally disrupts the standard crawling model. A vulnerability (e.g., a broken "Delete User" endpoint) may only exist in a "Deep State" reachable solely via a specific sequence of DOM interactions: *Login* $\rightarrow$ *List Users* $\rightarrow$ *Select User* $\rightarrow$ *Click Options*.

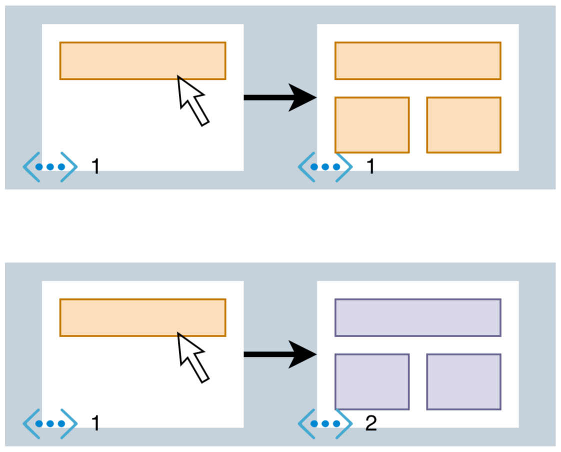

Fig. 1. Navigation Paradigms: Single Page Applications (Top) update content dynamically without changing the URL context (1 $\rightarrow$ 1), hiding states from traditional crawlers. Multi-Page Applications (Bottom) perform full navigations (1 $\rightarrow$ 2) visible to static scanners.

As noted by Doupé et al., traditional scanners that fail to execute JavaScript or track DOM mutations view SPAs as a single URL, missing the majority of the attack surface [2]. This transforms crawling from a simple URL enumeration task into a complex Graph Traversal Problem, where nodes (states) are not pre-defined but must be inferred dynamically by transitions (edges) [8].

C. The DAST Operational Workflow

While implementation details vary between tools, the archetypal DAST workflow follows a structured progression: Crawling $\rightarrow$ Fuzzing $\rightarrow$ Analysis.

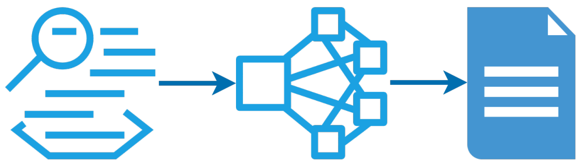

Fig. 2. The DAST Automation Pipeline: Discovery, Processing, and Reporting.

1) Phase 1: Crawling (Discovery): The objective is to map the "Attack Surface." Modern scanners must move beyond passive HTML spidering to JavaScript-aware discovery.

- **JavaScript-Aware Discovery:** Discovery can be augmented with JavaScript-capable crawlers (e.g., tools that parse and execute client-side code) to enumerate candidate URLs that are not present as static `href` links.
- **XHR Endpoint Collection:** Collecting endpoints associated with XHR-style traffic improves coverage on SPA backends where critical API paths are reached primarily through background requests.
- **Scope Note:** The approach studied here prioritizes backend endpoint discovery and template-driven HTTP checks; it does not attempt to fully reproduce UI-driven state transitions (e.g., multi-step click flows).

2) Phase 2: Fuzzing vs. Scanning: It is critical to distinguish between these terms, as they represent different depths of analysis.

- **Vulnerability Scanning:** This is deterministic and signature-based. It checks for the presence of known attributes (e.g., "Is the Apache version outdated?" or "Is debug=true exposed?"). It asks: *Is X present?*
- **Fuzzing:** This is non-deterministic and experimental. It involves generating invalid or unexpected data to observe how the application logic handles anomalies. For example, injecting `{'name': 'John' OR '1'='1'}` tests for SQL injection. It asks: *What happens if I break the rules?*

The Role of State in Fuzzing: Effective fuzzing requires context. A naive fuzzer fails because it sends payloads out of context (e.g., sending a checkout payload before authentication is established). The necessity of "State-Awareness" has been rigorously established in network security research. For instance, Natella et al. demonstrated that stateful fuzzers (those that infer and track the internal protocol state of a server) achieve significantly higher code coverage and bug detection rates than stateless fuzzers [9]. While Natella's work focused on network protocols, the same principle applies to modern web applications: tests are more effective when they preserve an authentication context and execute prerequisite request sequences. In the current implementation, this state is represented primarily as a single authentication context (e.g., a bearer token) and an ordered list of template requests; the scanner does not execute full client-side UI state transitions.

3) Phase 3: Analysis (The Oracle Problem): The final phase interprets the application's reaction. This confronts the "Oracle Problem": distinguishing between a secure failure (a handled 500 error) and an insecure failure (a crash, data leak, or successful bypass).

III. Market Stratification and Tool Capabilities

The landscape of Dynamic Application Security Testing (DAST) has undergone significant stratification. As web architectures have matured from server-side monoliths to complex client-side Single Page Applications (SPAs), the market has split into two distinct categories: widely adopted open-source frameworks driven by community contributions, and enterprise-grade commercial solutions characterized by advanced automation and proprietary crawling algorithms [10].

A. Global Market Dynamics

The global market for application security is expanding, driven by the increasing complexity of web applications and the ubiquity of open-source components. Recent audits reveal that **81% of commercial codebases contain high or critical-risk vulnerabilities**, necessitating robust automated testing [11]. This expansion is further accelerated by the "transitive dependency" problem, where **64% of open-source components are dependencies of dependencies**, making manual tracking nearly impossible and necessitating sophisticated scanning engines [11].

B. Open-Source Ecosystem

Open-source tools remain the backbone of the security research community. However, without enterprise-grade funding, they frequently face maintenance gaps and reliance on heavy external dependencies for DOM rendering [10].

1) OWASP Zed Attack Proxy (ZAP): OWASP ZAP stands as the de-facto standard for open-source DAST. It is favored for its seamless integration into CI/CD pipelines and ability to perform both active and passive scans [10]. Recent benchmarks indicate that ZAP performs consistently well against traditional vulnerabilities, achieving respectable recall rates (0.75–0.77) in controlled environments [12]. However, performance gaps exist; while ZAP can discover URLs in AJAX-heavy applications via its AJAX Spider, independent benchmarks note it often produces higher false positive rates compared to commercial engines [12].

2) ProjectDiscovery Nuclei: A rapidly emerging tool is Nuclei, which shifts from heuristic scanning to template-based matching. Unlike ZAP, Nuclei relies on external inputs (e.g., from tools like Katana) to map the attack surface rather than performing native crawling. Its architecture is built around YAML-based templates that define specific requests, allowing it to identify known vulnerabilities (CVEs) and misconfigurations. In comparative studies, Nuclei demonstrated a **zero false-positive rate** on standard benchmarks [10]. Updated benchmarks conducted for this study confirm that when equipped with custom application-specific templates and authentication context, Nuclei successfully detects critical business logic flaws.

3) Legacy and Niche Tools: The open-source ecosystem also includes tools that have struggled to adapt to the SPA era or have been deprecated.

- **Wapiti:** A command-line based scanner that functions as a "black-box" fuzzer. While recent versions have introduced a "headless" mode to parse JavaScript, this relies on external browser automation (GeckoDriver), introducing significant performance overhead compared to integrated crawling engines.

- **Arachni:** Once a high-performance framework, Arachni is now considered discontinued. The official repository was archived in 2021, rendering it incompatible with modern ECMAScript standards and actively unmaintained.

C. Commercial Landscape

Commercial scanners distinguish themselves through "Smart Crawling" capabilities, proprietary engines designed to execute JavaScript and construct a complex state graph of the application.

1) Burp Suite Professional: Widely regarded as the industry gold standard, Burp Suite Professional combines automated scanning with a powerful manual testing toolkit [10]. Recent empirical comparisons rank it as the most effective overall scanner, achieving the highest precision (0.73) and recall (0.90) in controlled experiments [12]. Its primary strength lies in its advanced crawling engine, which can navigate complex client-side states and authentication sequences that often baffle open-source alternatives.

2) Acunetix: Acunetix focuses heavily on enterprise automation. It employs technology specifically engineered to crawl HTML5 and JavaScript-heavy applications. In comparative benchmarks, Acunetix demonstrated a more **conservative scanning strategy** compared to ZAP, generating fewer total HTTP requests while still maintaining competitive vulnerability detection rates [13]. This efficiency makes it a preferred choice for organizations prioritizing low network overhead.

D. Comparative Summary

The divergence between open-source and commercial tools is most visible in the "Crawling Deficit." Commercial tools like Burp Suite utilize mature state engines that require less manual configuration to achieve high coverage.

IV. Proposed Methodology

A. Architectural Design: The "Clean-Slate" Engine

A primary failure mode in existing automated scanning is "State Pollution." When a scanner executes destructive payloads (e.g., deleting a user account or altering a database schema), it alters the System Under Test (SUT). Subsequent tests may then run against a corrupted state, leading to False Negatives (the vulnerability exists, but the precondition is broken) or False Positives (a crash occurs due to missing data, not a bug).

To solve this, the proposed scanner implements an **Idempotent Testing Architecture**.

- **Integration:** The scanner integrates with Docker Engine via CLI automation.
- **Workflow:** In lab mode, the scanner provisions a fresh, ephemeral container per scan run.
- **Scientific Justification:** This approach enforces "Test Isolation" for reproducible experiments.

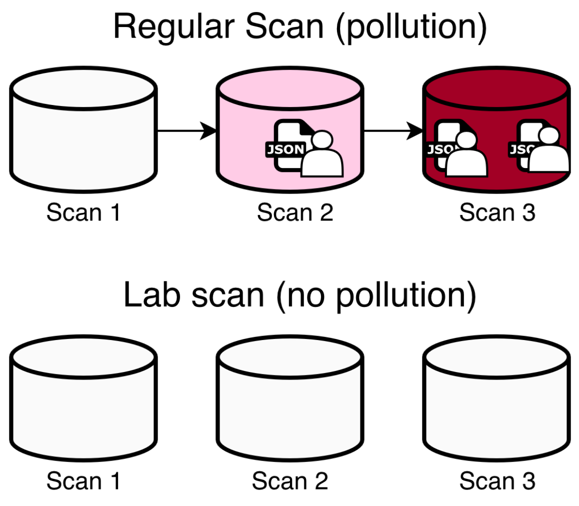

Fig. 3. Visualizing State Pollution: Traditional scans (top) accumulate artifacts that may interfere with subsequent tests. The proposed Clean-Slate approach (bottom) resets the state, ensuring idempotency.

B. Logic-Oriented Detection Modules

The core differentiation of this tool is its focus on business logic vulnerabilities and Insecure Direct Object References (IDOR), expressed as explicit YAML templates. These categories have traditionally required human intuition because they do not necessarily trigger HTTP 500 crashes or obvious error messages. The framework operationalizes detection by executing parameterized request sequences (templates) against discovered endpoints, using matchers to recognize anomalous yet successful outcomes. This balances generality (generic templates) with target specificity (app templates), addressing practical "scanner blindness" by ensuring that modern API-heavy surfaces are actively exercised rather than inferred from static HTML alone [2].

In the current implementation, templates can be expanded over discovered endpoints using path macros such as `@all@` (expand over all cached endpoints, substituting each cached endpoint's URL path) and `@api@` (expand over cached endpoints using several modes). In numeric-suffix and fallback modes, `@api@` expands primarily over cached endpoints whose paths contain `/api/` or begin with `/api`; in named-suffix mode, suffix matching may also select non-`/api` endpoints depending on the cached endpoint paths. Templates can also expand payload field names using auto-discovery placeholders of the form `{{autodiscover:<type>}}` (e.g., `privilege`, `status`, `token`, `id`), where the candidate field-name lists are loaded from a JSON configuration. The current implementation expands at most one auto-discovery placeholder per request body.

1) Business Logic: Semantic Validation (The "Negative Cost" Problem): Standard scanners rely heavily on HTTP status codes to determine pass/fail conditions. If an injection

```yaml
id: generic-input-validation
info:
  name: Input Validation - Empty/Null Values
  owasp_category: A07:2025
  severity: medium

requests:
  - name: Empty required field
    method: POST
    path: "@api@/users@"
    headers:
      Content-Type: application/json
    body: '{"email":"","password":"Test1234!"}'
    matchers:
      - type: status
        condition: in
        status: [200, 201]
      - type: word
        part: body
        words: ['"status":', '"data":', '"id":']
        condition: or
    matchers_condition: and
    on_match:
      vulnerability: EMPTY_FIELD_ACCEPTED
      message: Endpoint accepts empty required field

  - name: Null required field
    method: POST
    path: "@api@/users@"
    body: '{"email":null,"password":"Test1234!"}'
    matchers:
      - type: status
        condition: in
        status: [200, 201]
```

Listing 1. Sample Logic Template: Input Validation

Fig. 4. An abbreviated YAML template demonstrating the definition of active checks using endpoint expansion (@api@) and success-criteria matchers.

returns `500 Internal Server Error`, it flags a potential issue. If it returns `200 OK`, it assumes safety. This reliance on HTTP status is insufficient for logic flaws, where the server typically processes the request successfully but produces an invalid business outcome, a limitation highlighted in state-aware scanning research [2], [13].

The Flaw: In the Juice Shop environment, it is possible to add products to the basket with a negative quantity (e.g., `quantity: -5`). The server processes this mathematically, resulting in a negative total price (effectively refunding the user). The HTTP response is often a valid `200 OK`.

The Solution: This check is implemented as an active template that injects anomalous numeric values (e.g., negative quantities) and flags cases where the server responds successfully to inputs that should be rejected by the business tier. The approach mirrors invariant-based testing and intent synchronization proposed by Yin et al. for enhancing vulnerability detection in modern web applications [13].

2) IDOR: Resource Deviation Strategy: Insecure Direct Object References (IDOR) remain a "Blind Spot" for scanners. As noted by Rennhard et al., automated detection of these flaws is notoriously difficult because scanners lack the semantic understanding of the application's authorization model to distinguish between legitimate and illegitimate access to data objects [14]. While strong verification often benefits from comparing access patterns across different authorization contexts (e.g., User A vs. User B), the current implementation focuses on detecting deviations within a single authenticated context.

The Solution: The scanner implements a *Template-Driven Resource Enumeration* strategy across discovered API endpoints, expressed as templates that expand over API paths and append candidate identifiers (e.g., `/1`). Verification remains matcher-driven: templates may use status checks as a lightweight triage signal, and can be strengthened with content matchers (e.g., patterns for PII fields) when target semantics are known. In practice, the framework supports running scans in both unauthenticated and authenticated modes to compare behaviors and highlight broken access control.

- **Discovery:** The scanner maps endpoints and applies template-driven enumeration by appending candidate identifiers (e.g., `/1`) across discovered endpoints.
- **Attack Vector:** The scanner replays requests with modified identifiers (e.g., `GET /api/Basket/124`) while maintaining a bearer-token authentication context when provided.
- **Verification:** Templates apply matchers to identify suspiciously successful access (e.g., `200 OK`) and, when target semantics are known, may add content matchers for sensitive resource patterns (e.g., email, address, or transaction details).

3) JSON Web Token (JWT) Algorithm Confusion: This targets JWT algorithm confusion (also known as the "none" algorithm attack) [15]. A template helper can forge a JWT with `alg=none` by rewriting the header and stripping the signature (e.g., `jwt_none({{bearer_token}})`).

The forged token is replayed against endpoints requiring authentication to test for signature-verification bypass. This specific misconfiguration is frequently missed by generic fuzzers because they typically treat tokens as opaque strings. To successfully execute this attack, the scanner must decode the Base64 structure, parse the JSON header, modify the `"alg"` directive specifically, and re-encode the payload without the signature, a structured manipulation that random payload injection cannot achieve.

C. Operational Interface

The framework is implemented as a command-line interface application developed in Python. The operational workflow consists of two primary stages designed to enforce the separation between discovery and active exploitation.

- **Initialization Phase:** This stage performs the "Discovery" tasks. It invokes the crawling engine to map the target's attack surface, caches the discovered endpoints (both static navigational links and XHR endpoints), and scaffolds the configuration descriptors. This prepares the workspace without sending active attack payloads.
- **Scanning Phase:** This stage executes the "Attack" logic. The engine loads the cached endpoints and iterates through the enabled vulnerability templates (both generic and app-specific). It supports selective execution of individual templates and configurable authentication contexts (e.g., authenticated vs. unauthenticated scans).

- **Reporting:** The system generates artifacts to aid analysis, including a structured JSON output for machine parsing and a rendered HTML report for human review.

V. Experimental Evaluation

To validate the efficacy of the proposed solution, a comparative benchmark was conducted against ProjectDiscovery's **Nuclei** [16], representing the state-of-the-art in template-based open-source DAST scanners.

A. Benchmark Setup

The target environment was a locally provisioned instance of OWASP Juice Shop running on `http://localhost:3000`. Both scanners used the **same 14 custom templates** and an identical Bearer token for authenticated scanning. This ensures a fair comparison where detection capability depends solely on the scanning framework's features rather than template differences.

B. Results

The benchmark compared Nuclei v3.6.2 (with custom templates) against the proposed Squeezer framework on OWASP Juice Shop:

TABLE I
Comparative Scan Metrics on OWASP Juice Shop (Updated 2026-01-20)

Metric	Nuclei (Custom)	Squeezer
Scan Duration	0.24s	1.29s
Templates Used	14	14
Critical Findings	4	29
High Findings	3	8
Medium Findings	2	2
Total Matches	9	39
Total Unique Findings	6	18

C. Analysis

The benchmark results reveal important insights about template-driven DAST scanning:

Template Quality is Primary: Both Nuclei and Squeezer successfully detected critical business logic flaws when equipped with custom templates and authentication context. This confirms that detection capability depends primarily on template quality rather than scanning engine limitations. The earlier observation of "scanner blindness" in generic templates was attributable to lack of authentication context and application-specific logic, not fundamental engine constraints.

Coverage vs. Speed Trade-off: Nuclei completed its scan in 0.24s (approximately 5.4× faster than Squeezer's 1.29s), but detected only 9 matches (6 unique) compared to Squeezer's 39 matches (18 unique). Squeezer's longer scan time reflects its deeper inspection logic, endpoint expansion macros (`@api@`, `@all@`), and field-name auto-discovery capabilities.

Finding Density: Squeezer discovered 4.33× more total findings (39 vs 9) and 3× more unique vulnerabilities (18 vs 6) using the exact same number of templates (14). This demonstrates that each Squeezer template effectively generates multiple test cases through expansion and auto-discovery, whereas Nuclei templates typically execute a single fixed request.

Maintenance Efficiency: Squeezer achieved superior coverage with identical template count by using built-in expansion features that reduce manual template maintenance. Each Squeezer template effectively covers multiple endpoints through path macro expansion and multiple field variations via auto-discovery.

State Awareness: Both tools require explicit Bearer token authentication to reach protected vulnerable endpoints. The proposed framework's clean-slate lab mode provides additional reproducibility benefits by ensuring each scan runs against a fresh database state, eliminating cross-test contamination.

Use Case Differentiation: The results suggest distinct optimal use cases for each tool:

- **Nuclei:** Best suited for rapid security assessments, CI/CD pipeline integration, and scenarios where scan speed is prioritized over maximum coverage.
- **Squeezer:** Best suited for comprehensive security audits, penetration testing, and scenarios where maximum vulnerability coverage is critical.

VI. Conclusion

This study addresses SPA "scanner blindness" using a clean-slate, template-driven framework combining endpoint discovery with state-aware execution. Experiments against OWASP Juice Shop confirmed that template quality and authentication context, not the engine, are the primary drivers of detection. While Nuclei successfully identified critical flaws (e.g., path traversal, mass assignment, JWT issues) when equipped with custom templates, Squeezer offered significant advantages: 3× more unique findings (18 vs. 6) and 4.33× more total matches (39 vs. 9) using identical templates, achieved via endpoint expansion macros and field-name auto-discovery.

Benchmarks revealed a distinct trade-off: Nuclei is 5.4× faster (0.24s vs. 1.29s), optimizing it for CI/CD, while Squeezer's reproducible, deep-inspection logic is superior for comprehensive audits. Ultimately, the future of DAST lies not in faster generic fuzzing, but in smarter template design leveraging stateful execution and automated expansion.

References

[1] K. Kowalczyk and T. Szandala, "Enhancing SEO in single-page web applications in contrast with multi-page applications," *IEEE Access*, vol. 12, pp. 11 597–11 614, 2024. [Online]. Available: https://doi.org/10.1109/ACCESS.2024.3355740

[2] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, T. Kohno, Ed. USENIX Association, 2012, pp. 523–538. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe

[3] S. Qadir, E. Waheed, A. Khanum, and S. Jehan, "Comparative evaluation of approaches & tools for effective security testing of web applications," *PeerJ Comput. Sci.*, vol. 11, p. e2821, 2025. [Online]. Available: https://doi.org/10.7717/peerj-cs.2821

[4] B. Kimminich and OWASP Foundation, "OWASP Juice Shop: Probably the most modern and sophisticated insecure web application," https://owasp.org/www-project-juice-shop/, 2025, accessed: 2026-01-20.

[5] A. Arcuri, "Restful API automated test case generation with evomaster," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, pp. 3:1–3:37, 2019. [Online]. Available: https://doi.org/10.1145/3293455

[6] OWASP Foundation, "OWASP Top 10: The Ten Most Critical Web Application Security Risks," https://owasp.org/www-project-top-ten/, 2025.

[7] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh, "Technical guide to information security testing and assessment," Tech. Rep. 115, 2008. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-115

[8] S. Choudhary, M. E. Dincturk, S. M. Mirtaheri, A. Moosavi, G. von Bochmann, G.-V. Jourdan, and I.-V. Onut, "Crawling rich internet applications: the state of the art," in *Center for Advanced Studies on Collaborative Research, CASCON '12, Toronto, ON, Canada, November 5-7, 2012*, H.-A. Jacobsen, Y. Zou, and J. Chen, Eds. IBM / ACM, 2012, pp. 146–160. [Online]. Available: http://dl.acm.org/citation.cfm?id=2399790

[9] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *CoRR*, vol. abs/2110.06253, 2021. [Online]. Available: https://arxiv.org/abs/2110.06253

[10] V. Somi, "A comparative analysis and benchmarking of dynamic application security testing (dast) tools," *Journal of Engineering and Applied Sciences Technology*, Feb. 2024. [Online]. Available: https://doi.org/10.47363/JEAST/2024(6)E139

[11] Black Duck, "2025 open source security and risk analysis report (ossra)," Black Duck, Technical Report, 2025. [Online]. Available: https://www.blackduck.com/content/dam/black-duck/en-us/reports/rep-ossra.pdf

[12] T. Yarphel and D. Rani, "Comparative performance analysis of web vulnerability scanners," *International Journal of Information Security*, vol. 4, no. 1, pp. 98–110, 2025. [Online]. Available: https://doi.org/10.34218/IJIS_04_01_005

[13] Z. Yin, Y. Xu, F. Ma, H. Gao, L. Qiao, and Y. Jiang, "Scanner++: Enhanced vulnerability detection of web applications with attack intent synchronization," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, Feb. 2023. [Online]. Available: https://doi.org/10.1145/3517036

[14] M. Rennhard, M. Kushnir, O. Favre, D. Esposito, and V. Zahnd, "Automating the detection of access control vulnerabilities in web applications," *SN Comput. Sci.*, vol. 3, no. 5, p. 376, 2022. [Online]. Available: https://doi.org/10.1007/s42979-022-01271-1

[15] S.-W. Jang and S.-H. Lee, "Vulnerabilities and encryption applications of jwt-based authentication methods," *Journal of Information Systems Engineering & Management*, vol. 10, no. 8s, pp. 377–384, 2025. [Online]. Available: https://doi.org/10.52783/jisem.v10i8s.1055

[16] ProjectDiscovery, "Nuclei: Fast and customizable vulnerability scanner based on simple YAML based DSL," https://github.com/projectdiscovery/nuclei, 2024, version 3.6.2.