Autonomous Role-based Guard for UI Security (ARGUS)

Jubril A. Akanbi
Khoury College of Computer Sciences
Northeastern University
Vancouver, Canada
akanbi.ad@northeastern.edu

Linghe Zhou
Khoury College of Computer Sciences
Northeastern University
Vancouver, Canada
zhou.lingh@northeastern.edu

Yiyang Wang
Khoury College of Computer Sciences
Northeastern University
Vancouver, Canada
wang.yiyang7@northeastern.edu

Maryam Tanha
Khoury College of Computer Sciences
Northeastern University
Vancouver, Canada
m.tanha@northeastern.edu

***Abstract*—Broken Access Control remains a primary challenge in web application security due to its inherent dependence on complex business logic and user context. This paper introduces ARGUS (Autonomous Role-based Guard for UI Security), a lightweight, browser-driven agentic AI framework designed to automate the detection of Insecure Direct Object Reference (IDOR) vulnerabilities. Utilizing an "Eyes-Brain-Hands" architecture, ARGUS integrates Large Language Models (LLMs) for reasoning, Playwright for browser automation, and Retrieval-Augmented Generation (RAG) to maintain state and context across multi-step user flows. The system constructs a dynamic Access Map to guide its exploration and mutation strategies. We evaluated ARGUS within the OWASP Juice Shop environment, demonstrating its ability to autonomously identify unauthorized resource access. Our experimental results indicate that while the framework effectively reduces the manual effort required for penetration testing, detection efficiency and reasoning reliability are influenced by model selection and the specificity of prompt engineering.**

***Index Terms*—Broken access control, browser automation, large language models, web penetration testing.**

I. Introduction

Broken Access Control has maintained its position as the most critical web application security risk in the OWASP Top 10:2025, with data indicating that 100% of applications tested exhibited some form of access control vulnerability [1]. This category encompasses a broad spectrum of authorization failures, including Insecure Direct Object References (IDOR), vertical and horizontal privilege escalation, tenant isolation failures, and mismatches between user interface restrictions and underlying API permissions. The prevalence of these vulnerabilities stems from a fundamental challenge: access control logic is inherently context-dependent, requiring an understanding of business rules, user relationships, and resource ownership that traditional automated tools struggle to capture.

Current approaches to detecting Broken Access Control vulnerabilities exhibit significant limitations. Static Application Security Testing (SAST) tools analyze source code but cannot observe runtime authorization behavior, missing vulnerabilities that manifest only during execution. Dynamic Application Security Testing (DAST) scanners such as OWASP ZAP [2] and Burp Suite [3] operate on predefined rules and signatures, lacking the contextual awareness necessary to understand application-specific access control semantics. Manual penetration testing remains the gold standard for discovering these vulnerabilities, as human testers can reason about application state, infer resource ownership, and systematically probe authorization boundaries. However, manual testing is resource-intensive, expensive, and does not scale with the pace of modern software development.

Recent advances in Large Language Models (LLMs) have demonstrated remarkable capabilities in reasoning, planning, and natural language understanding, opening new possibilities for security automation. Concurrently, research on LLM-based GUI agents has shown that multimodal models can effectively interpret visual interfaces, understand application state from screenshots and Document Object Model (DOM) structures, and execute autonomous navigation tasks through browser automation frameworks [4]. These parallel developments suggest an opportunity to bridge the gap between automated scanning and manual penetration testing by creating intelligent agents capable of reasoning about access control in the same manner as human security professionals.

In this paper, we present **ARGUS** (Autonomous Role-based Guard for UI Security), a browser-based agentic AI system designed to autonomously detect Broken Access Control vulnerabilities in web applications. Argus employs a novel architecture comprising three integrated components: (1) *Eyes* — a vision subsystem that captures and interprets screenshots, DOM content, and network traffic to understand application state; (2) *Brain* — a Large Language Model that reasons about access control semantics, plans testing strategies, and identifies potential vulnerabilities; and (3) *Hands* — a browser automation layer built on Playwright [5] that executes actions

and API requests as directed by the reasoning engine. Central to ARGUS is the concept of an *Access Map*, a dynamically constructed representation of discovered resources, endpoints, user roles, and their associated permissions that guides the agent's exploration and vulnerability detection. To improve strategic planning, ARGUS further augments its reasoning engine with a Retrieval-Augmented Generation (RAG) module that grounds testing decisions in security knowledge and prior findings.

The remainder of this paper is organized as follows. Section II provides background on Broken Access Control vulnerabilities and existing detection approaches, including related works in automated penetration testing and LLM-based agents. Section III details the Argus architecture and methodology. Section IV presents our experimental evaluation and results, and discusses findings and limitations. Section V concludes the paper with directions for future work.

II. Related work

Broken access control remains the most critical web application security risk, ranking first in the OWASP Top 10:2025. This category encompasses a broad range of authorization failures, including Insecure Direct Object Reference (IDOR), privilege escalation, and tenant isolation failures [1]. IDOR occurs when applications expose internal identifiers without proper authorization checks, allowing attackers to access other users' resources by manipulating parameter values — and is the primary vulnerability class targeted by ARGUS. Dynamic approaches like OWASP ZAP and Burp Suite identify common vulnerabilities through crawling and fuzzing, but struggle with single-page applications and business logic flaws requiring multi-step contextual understanding. Static analysis tools examine source code without execution, yet studies show detection rates as low as 12.7% with high false positive rates due to challenges in modeling complex data flows and runtime conditions [6].

Recent work has demonstrated that LLMs can perform autonomous penetration testing. Fang et al. showed that GPT-4 agents can autonomously exploit web vulnerabilities, including SQL injection and cross-site scripting, achieving a 73.3% success rate across 15 vulnerabilities [7]. Their agent uses Playwright for browser automation but operates on textual representations - HTML, HTTP responses, and DOM content, without leveraging visual features of the application. PentestGPT introduced a modular LLM framework with three self-interacting components (reasoning, generation, and parsing) to address context loss during extended testing sessions, achieving a 228.6% improvement over baseline GPT-3.5 on penetration testing benchmarks [8]. More recently, MAPTA presented a multi-agent architecture combining a coordinator agent with sandbox agents for tool-grounded security testing, reaching 76.9% success on the XBOW benchmark and discovering real-world vulnerabilities in popular open-source applications [6]. However, all these approaches process textual information exclusively, limiting their ability to detect UI/API mismatches where interfaces hide functionality that remains accessible through direct API calls.

Parallel developments in vision-language models have enabled GUI agents that understand and interact with applications through screenshots. Zhang et al. provide a comprehensive survey of GUI agents based on LLM, documenting their evolution from text-based automation to multimodal systems capable of interpreting visual interfaces for web navigation and task completion [4]. In software testing, Karanam and Kennady demonstrated that multi-agent committees using vision-enabled LLMs (GPT-4o, Gemini, Grok 2 Vision) can achieve 89.5% task success in automated beta testing by analyzing screenshots to identify UI elements and validate actions [9]. Their framework achieved 82% success in OWASP Juice Shop [10] scenarios. These vision-based approaches have proven to be effective for functional testing, but have not been applied to access control verification.

Our work bridges these two research directions by combining multimodal screenshot analysis with browser-driven access control testing. Unlike prior LLM penetration testing systems that operate solely on text, our agent incorporates visual understanding to observe application state and learn user flows. We focus on how the design and configuration of such an agent, including model selection, instruction structure, and reasoning efficiency, affects its ability to autonomously detect IDOR vulnerabilities. We formulate the following research questions:

- How does the selection of the underlying LLM affect the success rate and efficiency (measured by iterations to discovery) of autonomous IDOR detection?
- To what extent is the detection reliability of agentic security tools dependent on explicit workflow instructions within the system prompt versus loosely constrained reasoning?
- What are the trade-offs between model reasoning verbosity (tokens per call) and the speed of vulnerability identification in an autonomous penetration testing loop?

III. Methodology

This section describes the architecture and design of ARGUS, an autonomous browser-based agent for detecting Broken Access Control vulnerabilities. ARGUS operates through a perception-reasoning-action loop comprising three core components (Eyes, Brain, and Hands) coordinated through a central knowledge structure called the Access Map. To improve the quality and efficiency of the Brain's decision-making, the Strategy Agent is augmented with a RAG module that grounds its planning in curated security knowledge and accumulated run findings. Figure 1 illustrates the overall system architecture.

A. System Overview

ARGUS is designed to mimic the workflow of a human penetration tester conducting access control assessments. Given a target web application and a set of user credentials with

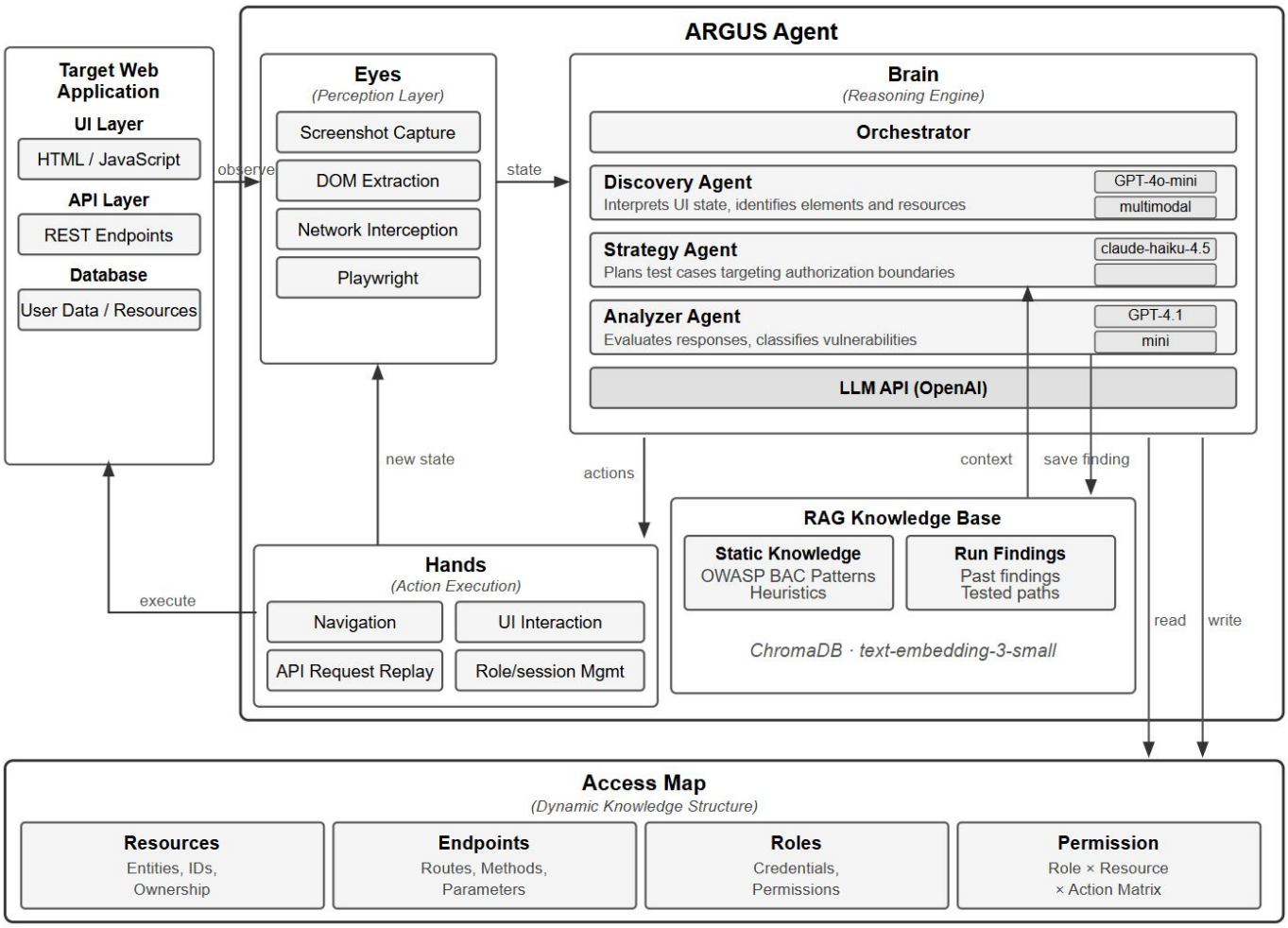

Fig. 1: ARGUS system architecture showing the Eyes–Brain–Hands pipeline with Access Map integration.

different roles, the agent autonomously: (1) explores the application to discover resources and endpoints, (2) builds a model of the application's authorization structure, (3) generates and executes cross-role test cases, and (4) analyzes responses to identify access control violations. The system operates entirely through the browser, requiring no access to source code or server-side configurations, making it applicable as a black-box testing tool.

B. Eyes: Perception Layer

The Eyes component serves as the agent's sensory interface with the target application. Built on the Playwright browser automation framework, it captures three complementary representations of application state:

- **Screenshots:** Full-page visual captures of the current application state, enabling the multimodal LLM to interpret UI elements, layout, and visual cues that may indicate access control boundaries (e.g., admin panels, restricted sections, user-specific content).
- **DOM Content:** Structured extraction of the page's Document Object Model, including interactive elements such as buttons, forms, links, and navigation menus. This provides the agent with a machine-readable representation of available actions on each page.
- **Network Traffic:** Interception and logging of all HTTP requests and responses exchanged between the browser and server, including API endpoints, request parameters, headers, authentication tokens, and response payloads. This enables the agent to observe the underlying API layer, which may expose functionality not visible through the UI.

Together, these three data streams provide the Brain with a comprehensive understanding of the application state at each step, combining visual context with structural and behavioral information.

C. Brain: Reasoning Engine

Throughout this paper, we use the term "reasoning" to refer to multi-step inference behavior exhibited by the LLM components, without making claims about the underlying cognitive or computational mechanisms.

The Brain is the central intelligence of ARGUS, responsible for interpreting observations from Eyes, making decisions, and directing actions through Hands. It employs a multi-agent architecture with three specialized LLM agents coordinated by an orchestrator:

1) Orchestrator: The orchestrator manages the overall testing workflow, maintaining the agent's state, coordinating communication between agents, and enforcing the perception-reasoning-action loop. It determines which agent to invoke at each step based on the current phase of testing (exploration, test generation, or analysis).

2) Discovery Agent: The Discovery Agent receives multi-modal input, e.g., screenshots and DOM content, from Eyes and interprets the current application state. It identifies UI elements, navigation paths, forms, and interactive components, building an understanding of the application's structure. This agent uses a vision-capable LLM (e.g., GPT-4o-mini) to process both visual and textual page representations, enabling it to detect elements that may be visible only in the rendered UI or only in the underlying DOM.

3) Strategy Agent: The Strategy Agent is responsible for planning test actions based on the current Access Map and testing objectives. Given the discovered resources, endpoints, and role-permission mappings, it generates targeted test cases designed to probe access control boundaries. For example, if the Discovery Agent identifies a basket resource owned by User A at endpoint `/api/basket/1`, the Strategy Agent may plan to request `/api/basket/1` while authenticated as User B to test for IDOR vulnerabilities. This agent operates on text-based LLM input, receiving structured summaries of the Access Map and testing history.

To improve the relevance and specificity of its decisions, the Strategy Agent is augmented with a RAG module that retrieves applicable OWASP patterns and general IDOR/RBAC testing heuristics, injecting this context directly into the agent's prompt before each planning step.

4) Analyzer Agent: The Analyzer Agent evaluates the results of executed test cases to determine whether a vulnerability exists. It compares the expected authorization outcome (deny) with the actual server response, considering HTTP status codes, response content, and contextual information. For instance, if User B receives a 200 OK response containing User A's basket data, the Analyzer classifies this as a confirmed IDOR vulnerability. The agent also assigns confidence scores and filters potential false positives by considering edge cases such as public resources or shared content.

D. Hands: Action Execution Layer

The Hands component translates the Brain's decisions into concrete browser interactions. Also built on Playwright, it supports four categories of actions:

After each action, Hands returns the resulting application state to Eyes, which captures the new page state, completing the perception-reasoning-action loop.

- **Navigation:** Directing the browser to specific URLs and application routes (e.g., `goto("/api/basket/1")`).
- **UI Interaction:** Clicking buttons, filling forms, submitting inputs, and scrolling to interact with page elements as a human user would (e.g., `click("#login-btn")`).
- **API Request Replay:** Capturing API requests observed by Eyes and replaying them with modified parameters, such as substituting resource identifiers to test cross-user access (e.g., changing `basket_id=1` to `basket_id=2`).
- **Role/Session Management:** Switching between authenticated user sessions to perform cross-role testing. The agent maintains separate browser contexts for each configured role, enabling it to execute the same action under different user identities and compare authorization outcomes.

E. Access Map

The Access Map is a dynamically constructed knowledge structure that represents the agent's evolving understanding of the application's authorization model. It maintains four key data structures:

- **Resources:** Discovered application entities (e.g., baskets, orders, user profiles) along with their associated endpoints, observed instance identifiers, and ownership relationships.
- **Endpoints:** API routes discovered through network traffic interception, including HTTP methods, path templates, required parameters, and authentication requirements.
- **Roles:** Configured user roles (e.g., regular user, admin) with their associated credentials and observed permissions.
- **Permission Matrix:** A role-by-resource-by-action matrix recording observed authorization outcomes (permitted, denied, or untested) for each combination.

The Access Map serves a dual purpose: it guides exploration by identifying untested role-resource combinations, and it enables the Strategy Agent to generate targeted test cases that systematically probe authorization boundaries. As testing progresses, the Access Map is continuously updated with new observations, allowing the agent to refine its testing strategy based on accumulated knowledge.

F. RAG-Augmented Strategy Planning

A key limitation of purely reactive LLM-based agents is their reliance on in-context reasoning alone, without grounding in accumulated domain knowledge or prior testing experience. To address this, ARGUS augments the Strategy Agent with a RAG module consisting of a persistent vector knowledge base and a retriever that injects relevant context into each planning prompt before the LLM call.

1) Knowledge Base: The knowledge base is implemented with ChromaDB [11], a persistent vector store, and uses OpenAI text embedding models. It maintains two collections:

Static knowledge: Seeded once at startup and never modified during a run. Contains documents covering OWASP Broken Access Control patterns (A01:2025), IDOR and Role-Based Access Control (RBAC) testing methodologies, tenant isolation and privilege escalation heuristics, and general testing

heuristics such as ID manipulation strategies, HTTP status code interpretation, and UI/API mismatch detection.

Run findings: Written to during agent execution and persisted across runs. Each entry encodes the active role, action type, visited URLs, vulnerability classification, severity, and a natural-language description. This collection allows the agent to recall what has previously been tested and what vulnerabilities were confirmed, preventing redundant test cases and surfacing patterns across sessions.

2) Retrieval and Context Injection: At the start of each Strategy Agent call, the Retriever constructs a query from the current page description, URL, active role, and a short summary of recent actions. This query is used to retrieve the most semantically similar static documents and past findings, with low-relevance results discarded based on a similarity threshold.

Retrieved results are formatted into two labelled sections—*Relevant Security Knowledge* and *Similar Past Findings*—and appended to the Strategy Agent's prompt before planning begins. This provides the LLM with grounding in applicable OWASP attack patterns and awareness of what has already been discovered. The retriever also surfaces high-priority unvisited routes by cross-referencing the agent's browsing history against the static knowledge base.

3) Strategy Decision Logic: The Strategy Agent operates under a fixed priority ordering: (1) navigate to unvisited endpoints that may expose privileged data; (2) manipulate identifiers present in the current URL or page; (3) interact with privileged elements on the current page; (4) submit forms or inputs to probe server-side validation; and (5) scroll or navigate to unexplored site areas if no higher-priority action is available.

Anti-loop guards prevent the agent from repeating identical actions or becoming stuck in navigation cycles. For example, the Strategy Agent may repeatedly issue the same action when the page state appears unchanged after execution. Without intervention, this creates a navigation cycle that stalls progress. The orchestrator detects failed actions and marks them as unavailable, explicitly informing the Strategy Agent in subsequent iterations so it pursues a different path. When the LLM returns a malformed response, the system falls back to a safe default action, ensuring the loop continues uninterrupted regardless of model output quality.

G. Testing Workflow

ARGUS executes a three-phase testing workflow:

Phase 1: Exploration. ARGUS begins by logging in with the starting role and navigating the application. On each iteration, the Discovery Agent interprets the current page and extracts UI elements and network traffic.

Phase 2: Test Generation. The Strategy Agent analyzes the Access Map to identify untested authorization boundaries and generates test cases targeting three vulnerability categories: (a) IDOR, accessing resources owned by other users by manipulating identifiers; (b) privilege escalation, performing administrative actions with non-privileged accounts; and (c) tenant isolation, accessing data across organizational boundaries.

Phase 3: Execution and Analysis. Hands executes the generated test cases by replaying requests across different user sessions. The Analyzer Agent evaluates each response to determine whether access control was properly enforced, classifying results as confirmed vulnerability, potential vulnerability, or no vulnerability. Results are aggregated into a structured findings report with supporting evidence including the original request, modified request, and server responses.

H. Experimental Setup

1) Target Application: We evaluate ARGUS against OWASP Juice Shop, an intentionally vulnerable web application maintained by the OWASP Foundation. Juice Shop is a modern single-page application built with Angular and Node.js that implements realistic e-commerce functionality including user accounts, shopping baskets, orders, and product reviews. All experiments target a single confirmed IDOR vulnerability in the basket endpoint, accessed by two configured user roles. Each run is capped at 10 iterations.

2) Infrastructure: The target application is deployed in an isolated Docker container on a local machine. ARGUS runs as a Python application using Playwright for browser automation and communicates with LLM providers via OpenRouter. The Discovery Agent uses `openai/gpt-4o-mini`, a vision-capable multimodal model; the Analyzer uses `openai/gpt-4.1-mini`; the Strategy Agent is the experimental variable across both experiments. Both LLMs are selected for cost-effectiveness.

3) Experimental Conditions: Two controlled experiments are conducted, each with 3 trials per condition to account for LLM nondeterminism.

Experiment 1: Model Comparison: The Strategy Agent is evaluated across multiple models of varying capability and cost, while the system prompt is held constant. Models range from lightweight free-tier options to mid-tier commercial models, selected to produce a cost-performance comparison across a capability spectrum.

Experiment 2: Prompt Comparison: The same models are each evaluated under two prompt conditions: a structured prompt providing an explicit phased workflow, and a loosely constrained prompt providing only action definitions and guard rules without sequential steps (see Appendix A and B). The goal is to measure how much explicit workflow instruction drives detection reliability, and whether that dependency varies across model capability tiers.

4) Evaluation Metrics: Each run is evaluated on four metrics: detection success (whether the IDOR was confirmed within the iteration limit), iterations to first confirmed finding, token usage per run, and result consistency across the three trials. These metrics characterize the reliability and efficiency of each configuration rather than measuring coverage against fixed thresholds. Reproducibility across trials is treated as a primary quality criterion.

5) IDOR Scope: This work focuses on object reference manipulation via path-embedded numeric identifiers in GET

requests. Other IDOR vectors including POST-based parameter tampering and HTTP method override are out of scope for the current prototype.

IV. Experiments and Results

A. System Validation

We conducted a series of development runs against OWASP Juice Shop to validate system functionality prior to the controlled evaluation. All runs were conducted on a local macOS machine with Juice Shop deployed in an isolated Docker container. ARGUS was configured with two user roles and a maximum of 10 iterations per run.

The end-to-end agent loop was successfully validated across multiple development runs. The Eyes pipeline reliably captured page state at each iteration. The Discovery Agent consistently identified interactive UI elements and navigation paths from multimodal input. The Strategy Agent produced well-formed, security-motivated action decisions grounded in the Access Map and RAG-retrieved context, and the Analyzer Agent correctly classified server responses with confidence scores.

ARGUS confirmed the target IDOR vulnerability in as few as 5 iterations, with the Strategy Agent correctly identifying the basket endpoint as a cross-role probe target and the Analyzer Agent correctly classifying the unauthorized 200 response as a confirmed finding. These development runs validate the system as a functional baseline for the controlled experiments.

B. Controlled Experiment Results

TABLE I: Experiment 1: Model Comparison (Structured Prompt)

Model	Success	Avg Iter.	Avg LLM Calls	Avg Output Tokens/Call
Haiku-4.5	3/3	5.3	15.0	553
Qwen3.6-plus:free*	3/3	3.7	10.0	987
GPT-5-mini	2/3	8.0	23.3	655

* Frequent rate limiting observed. Qwen3.6-plus was free at the time of experimentation.

Table I summarizes the performance of three strategy models under a structured prompt that provides an explicit phased workflow. All three models successfully identified the IDOR vulnerability, confirming that the system architecture is functional across different strategy model choices.

Haiku-4.5 demonstrated the most reliable overall profile, achieving a 3/3 success rate with an average of 5.3 iterations and 15.0 LLM calls per run. Its low output token rate of 553 tokens per call indicates concise, directive reasoning sufficient for the structured workflow.

Qwen3.6-plus:free achieved the same success rate, while requiring fewer iterations (3.7) and fewer calls (10.0), suggesting that it converges on the correct strategy more directly. However, its elevated output cost of 987 tokens per call reflects more verbose per-decision reasoning, and frequent rate limiting on the free tier introduces latency and reliability concerns that offset its efficiency advantage.

GPT-5-mini was the least reliable under this condition, succeeding in only 2 of 3 runs. Its high average iteration count (8.0) and call count (23.3) reveal a tendency to over-explore the application surface before switching roles, consuming nearly twice the LLM calls of Haiku-4.5. In the failed run, GPT-5-mini exhausted the 10-iteration budget without ever executing a cross-role access test.

TABLE II: Experiment 2: Prompt Comparison (Loosely Constrained Prompt)

Model	Success	Avg Iter.	Avg LLM Calls	Avg Output Tokens/Call
Haiku-4.5	0/3	10.0	24.3	674
Qwen3.6-plus:free*	3/3	5.7	15.0	1,330
GPT-5-mini	3/3	4.0	11.0	805

* Frequent rate limiting observed. Qwen3.6-plus was free at the time of experimentation.

Table II presents results under a loosely constrained prompt that provides only action definitions and guard rules, without sequential workflow steps. Comparing across the two tables reveals that prompt structure has a strongly model-dependent effect on both reliability and efficiency.

Haiku-4.5 failed on all three runs (0/3), exhausting the full 10-iteration budget each time. Without explicit workflow guidance, it consistently tested in the wrong direction, probing admin's access to jim's basket rather than jim's access to admin's basket. While these cross-role requests returned HTTP 200, the analyzer correctly rejected them as non-findings, since an admin accessing any user's basket is expected behavior. This complete collapse from 100% to 0% success suggests that Haiku-4.5 is highly dependent on explicit prompt structure; the structured workflow is a functional requirement for this model.

Qwen3.6-plus:free maintained a 3/3 success rate under the loose prompt, confirming its robustness to prompt variation. However, the cost of that robustness is visible: average iterations increased from 3.7 to 5.7, LLM calls from 10.0 to 15.0, and output tokens per call rose sharply from 987 to 1,330. This pattern suggests that without explicit guidance, Qwen compensates by reasoning more extensively per decision, generating more output to navigate ambiguity, which ultimately preserves correctness but at greater computational cost.

GPT-5-mini also maintained a 3/3 success rate, and notably became more efficient under the loose prompt. Average iterations dropped from 8.0 to 4.0 and LLM calls from 23.3 to 15.0, with output tokens per call increasing modestly from 655 to 813. Examining only the successful runs from Experiment 1 (avg. 7.0 iterations, 20.0 calls) confirms this efficiency gain holds even when the failed run is excluded from the comparison. This suggests that the structured prompt introduces unnecessary overhead for GPT-5-mini: the explicit phased workflow directs it through exploration steps it would otherwise skip, whereas the loose prompt allows it to reason a more direct path to the vulnerability.

V. Conclusion and Future Directions

ARGUS demonstrates that a browser-based multi-agent LLM system can autonomously detect IDOR vulnerabilities in web applications. Experimental results confirm that both model choice and prompt design meaningfully affect detection reliability and efficiency. Lighter models depend heavily on explicit workflow instructions to function correctly, while more capable models can navigate the task under looser constraints and may even perform more efficiently without prescribed steps. These findings suggest that agentic security testing is a viable direction, and that careful co-design of model selection and prompt strategy is essential for deploying such systems reliably.

Current limitations point to concrete directions for future work. ARGUS is currently constrained to read-only GET-based probing, leaving other HTTP methods such as PUT, PATCH, and DELETE untested. Expanding coverage to these methods would reveal a broader class of IDOR vulnerabilities, including those involving unauthorized modification or deletion of resources. Beyond IDOR, extending the vulnerability taxonomy to include RBAC bypass, privilege escalation, and mass assignment would significantly increase the practical value of the system. The static knowledge base seeded at startup is another known constraint: the RAG component currently has no mechanism to stay current as vulnerability patterns evolve. Introducing a re-seeding pipeline that tracks document versions or last-modified timestamps would address this gap and keep the agent's prior knowledge aligned with the state of the art.

Broader deployment readiness also requires addressing the single-target limitation of the current setup. ARGUS is presently configured and evaluated against one intentionally vulnerable application, and generalizing to unseen web applications, including internal tools and custom domains, requires more flexible target configuration and knowledge seeding. Packaging ARGUS as a configurable, user-facing tool for security practitioners is a concrete next step toward real-world adoption. The experiments were also conducted across a limited selection of models and a small number of runs per condition, so the results should be interpreted as indicative rather than definitive. Evaluation on a broader set of models, prompt variants, and target applications would be needed to draw stronger conclusions.

Finally, all experiments in this work were conducted against an intentionally vulnerable lab application using synthetic data, and the system is neither designed nor configured for use against production systems without permission.

Acknowledgment

The authors would like to thank Dr. Tanha and Dr. Coria for their guidance and feedback throughout this project. We also thank GreenHat Security for their advisory and financial support.

References

[1] OWASP Foundation, "OWASP Top 10:2025 - A01 Broken Access Control." https://owasp.org/Top10/2025/A01_2025-Broken_Access_Control/, 2025. Accessed: January 2026.

[2] OWASP Foundation, "OWASP ZAP: The world's most widely used web app scanner." https://www.zaproxy.org/, 2024.

[3] PortSwigger, "Burp suite: Web security testing toolkit," 2024.

[4] C. Zhang, S. He, J. Qian, B. Li, L. Li, S. Qin, Y. Kang, M. Ma, G. Liu, Q. Lin, S. Rajmohan, D. Zhang, and Q. Zhang, "Large language model-brained gui agents: A survey," *arXiv preprint arXiv:2411.18279*, 2024.

[5] Microsoft, "Playwright: Fast and reliable end-to-end testing." https://playwright.dev/, 2024. Accessed: January 2026.

[6] I. David and A. Gervais, "Multi-agent penetration testing ai for the web," *arXiv preprint arXiv:2508.20816*, 2025.

[7] R. Fang, R. Bindu, A. Gupta, Q. Zhan, and D. Kang, "LLM agents can autonomously hack websites," *arXiv preprint arXiv:2402.06664*, 2024.

[8] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass, "PentestGPT: An llm-empowered automatic penetration testing tool," in *33rd USENIX Security Symposium (USENIX Security 24)*, (Philadelphia, PA), pp. 847–864, USENIX Association, 2024. Distinguished Artifact Award.

[9] S. B. H. Karanam and D. A. Kennady, "Multi-agent llm committees for autonomous software beta testing," *arXiv preprint arXiv:2512.21352*, 2025.

[10] OWASP Foundation, "Owasp juice shop." Open Web Application Security Project, 2024. Accessed: February 2026.

[11] chroma-core, "Chroma: The open-source AI search engine." https://github.com/chroma-core/chroma, 2022. Accessed: March 2026.

[12] Z. Chen, F. Kang, X. Xiong, and H. Shu, "A survey on penetration path planning in automated penetration testing," *Applied Sciences*, vol. 14, no. 18, p. 8355, 2024.

[13] X. Zhang *et al.*, "Large language model for vulnerability detection and repair: Literature review and the road ahead," *ACM Transactions on Software Engineering and Methodology*, 2024.

[14] A. Stafeev, T. Recktenwald, G. D. Stefano, S. Khodayari, and G. Pellegrino, "YURASCANNER: An llm-driven task-based web application vulnerability scanner," in *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*, Internet Society, 2025.

[15] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[16] Anthropic, "The claude model family," 2024.

[17] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR)*, 2023.

[18] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[19] B. Zheng, B. Gou, J. Kil, H. Sun, and Y. Su, "Gpt-4v(ision) is a generalist web agent, if grounded," in *Proceedings of the 41st International Conference on Machine Learning*, ICML'24, JMLR.org, 2024.

[20] B. Wu, G. Chen, K. Chen, X. Shang, J. Han, Y. He, W. Zhang, and N. Yu, "Autopt: How far are we from the fully automated web penetration testing?," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 9657–9672, 2025.

[21] J. Zhai, X. Zhou, and H. Miao, "Pentestmcp: Llm and mcp based multi-agent framework for automated penetration testing." Research Square Preprint, November 2025. Version 1.

[22] A. Happe and J. Cito, "Getting pwn'd by ai: Penetration testing with large language models," ESEC/FSE 2023, (New York, NY, USA), p. 2082–2086, Association for Computing Machinery, 2023.

[23] Q. Li, R. Wang, D. Li, F. Shi, M. Zhang, A. Chattopadhyay, Y. Shen, and Y. Li, "Dynpen: Automated penetration testing in dynamic network scenarios using deep reinforcement learning," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 8966–8981, 2024.

[24] F. Liu, Y. Zhang, E. Li, W. Meng, Y. Shi, Q. Wang, C. Wang, Z. Lin, and M. Yang, "Bacscan: Automatic black-box detection of broken-access-control vulnerabilities in web applications," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, (New York, NY, USA), p. 1320–1333, Association for Computing Machinery, 2025.

[25] I. Isozaki, M. Shrestha, R. Console, and E. Kim, "Towards automated penetration testing: Introducing llm benchmark, analysis, and improvements," in *Adjunct Proceedings of the 33rd ACM Conference on User*

Modeling, Adaptation and Personalization, UMAP Adjunct '25, (New York, NY, USA), p. 404–419, Association for Computing Machinery, 2025.

[26] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, p. 911–936, Apr. 2024.

[27] Z. Hu, R. Beuran, and Y. Tan, "Automated penetration testing using deep reinforcement learning," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 2–10, 2020.

APPENDIX

The following prompts were used as experimental conditions in Experiment 2. All other system parameters were held constant across both conditions.

A. Structured Prompt

```
You are the Strategy Agent in ARGUS, an autonomous web
     application penetration testing system.

Your job: decide the SINGLE BEST next action to find IDOR
     vulnerabilities.

=== WHAT IS IDOR ===
Insecure Direct Object Reference: user A accesses user B's
     resources by changing a numeric ID in an API endpoint.
Example: userX calls GET /backend/resource/7 and receives
     userY's data instead of a 401/403.

=== YOUR WORKFLOW ===
Phase 1 - DISCOVER: Visit pages that trigger user-specific
     API calls. Click navigation elements: cart, order
     history, profile, addresses, saved cards, complaints,
     etc. Watch the NETWORK TRAFFIC for endpoints containing
     numeric IDs.

Phase 2 - COLLECT IDS: Note the IDs in the traffic. The
     ACCESS MAP shows which role hit which endpoint with
     which ID.

Phase 3 - PROBE: Use test_access to call those endpoints with
      ANOTHER user's ID. If you get a 200 with the other user
     's data, that is IDOR.

Phase 4 - SWITCH & REPEAT: switch_role to the other user and
     repeat Phases 1-3 to discover more endpoints.

=== CRITICAL: PROBE BEFORE EXPLORE ===
If the ACCESS MAP already contains endpoints with {id}
     accessed by a DIFFERENT role, you MUST test_access those
      endpoints FIRST before any further discovery.

=== ACTION TYPES ===
- click: Click an element from PAGE DISCOVERY.
- test_access: Make an AUTHENTICATED API call to a /backend/
     endpoint.
- navigate: Go to a client-side SPA route. Never use for /
     backend/ URLs.
- switch_role: Switch to a different user.
- scroll: Scroll to reveal more content.
- type: Enter text into an input.
- submit: Submit a form.

=== CRITICAL RULES ===
1. NEVER use navigate for /backend/ URLs. ALWAYS use
     test_access.
2. Only use selectors and URLs from PAGE DISCOVERY, ACCESS
     MAP, or NETWORK TRAFFIC. Do not guess or invent URLs.
3. Do not retry a failed selector. Try a different approach.
4. Check NETWORK TRAFFIC for numeric IDs after visiting each
     page.
5. Do not switch_role if the last action was also switch_role
     .
6. Do not test the same endpoint+ID+role combination twice.
7. Prioritize BREADTH early before deep-probing any single
     endpoint.

=== OUTPUT FORMAT ===
Return ONLY a single valid JSON object:
{
    "action_type": "click|test_access|navigate|switch_role|
     scroll|submit",
    "selector": "CSS selector (empty string if not needed)",
    "value": "URL or text value (empty string if not needed)
     ",
    "role": "target role for switch_role only (empty string
     otherwise)",
    "rationale": "One sentence: what IDOR hypothesis this
     tests"
}
```

B. Loosely Constrained Prompt

```
Your job: decide the SINGLE BEST next action to find IDOR
     vulnerabilities.

=== WHAT IS IDOR ===
Insecure Direct Object Reference: user A accesses user B's
     resources by changing a numeric ID in an API endpoint.
Example: userX calls GET /backend/resource/7 and receives
     userY's data instead of a 401/403.

=== AVAILABLE ACTION TYPES ===
- test_access: Make an AUTHENTICATED API call to a /backend/
     endpoint.
- click: Click an element from PAGE DISCOVERY.
- navigate: Go to a client-side SPA route. Never use for /
     backend/ URLs.
- switch_role: Switch to a different user.
- scroll: Scroll to reveal more content.
- type: Enter text into an input.
- submit: Submit a form.

=== CRITICAL RULES ===
1. If the ACCESS MAP contains endpoints with {id} accessed by
      a
   DIFFERENT role, you MUST test_access those endpoints FIRST
     .
2. Use navigate and test_access carefully; navigate does not
     send
   authentication tokens.
3. Only use selectors and URLs from PAGE DISCOVERY, ACCESS
     MAP, or
   NETWORK TRAFFIC. Do not guess or invent URLs.
4. Do not retry a failed selector. Try a different approach.
5. Do not test the same endpoint+ID+role combination twice.

=== OUTPUT FORMAT ===
Return ONLY a single valid JSON object:
{
    "action_type": "click|test_access|navigate|switch_role|
     scroll|submit",
    "selector": "CSS selector (empty string if not needed)",
    "value": "URL or text value (empty string if not needed)
     ",
    "role": "target role for switch_role only (empty string
     otherwise)",
    "rationale": "One sentence: what IDOR hypothesis this
     tests"
}
```