From Prompt to Pwn: Browser-Empowered LLM Agents for Web Penetration Testing

DIMITRIOS KALOPISIS, University of Twente, The Netherlands

Abstract

Penetration testing is the practice of simulating real-world attacks on information systems to uncover vulnerabilities before adversaries exploit them. It is labor-intensive because testers must gather intelligence, plan multi-step exploits, interact with diverse interfaces, and document every action with precision and accuracy. Recent progress in large language models (LLMs) has sparked efforts to automate parts of this workflow. Prior studies have shown that LLM agents can generate commands, interpret tool output, and chain tasks, yet they lack programmatic control over a web browser. That omission limits their ability to test the client-side logic that dominates modern web applications.

This paper examines whether adding browser access to LLM-driven agents closes that gap. We build two agents: a command-line baseline and a browser-enabled variant that can click, type, read the Document Object Model, and watch network traffic. Both are evaluated on twenty-seven Web Security labs covering nine common web vulnerabilities. The browser-enabled agent solves 66.7 percent of the labs, compared to 40.7 percent for the baseline, expands coverage from three to seven vulnerability classes, and achieves the largest gains on medium-difficulty tasks. These findings confirm that giving agents control over a web browser is a major step forward for automated penetration testing. The next challenges are to incorporate precise timing analysis and support for out-of-band interactions, enabling the process to become fully autonomous.

1 INTRODUCTION

AI has rapidly become a cornerstone of modern cybersecurity strategy. Google Cloud's 2024 State of AI & Security survey of 2486 security practitioners reveals that 55 % of organizations intend to deploy generative-AI security tooling within the next 12 months, and 19 % already plan to apply it to attack-simulation and red-team automation. Yet only 63 % believe these tools will clearly improve their defenses, while a 33 % skills-gap remains the top implementation barrier, signaling an enthusiasm–readiness gulf that makes disciplined penetration testing more critical than ever. [4]

Penetration testing, or pentesting, is still vital for protecting systems, but requires many work hours and there are not enough skilled testers [1, 10]. Recent progress in artificial intelligence, and especially in large language models (LLMs), therefore, has renewed interest in automating parts of this work [7]. Early studies showed that classical AI and machine learning methods can help identify vulnerabilities, but also pointed out scaling and real-time analysis problems that remain today [10]. The arrival of instruction-tuned LLMs has pushed research beyond basic tasks such as simple classification or sending random inputs and toward full attack workflows, making expert techniques easier to share and speeds up each test cycle [6].

Despite this progress, fully autonomous LLM-based pentesting is not yet fully solved. Exploratory studies such as *PentestGPT* demonstrate that current models can reliably generate tool commands, interpret outputs, and chain sub-tasks, yet they struggle to retain long-range context, to reason strategically across multiple attack paths and to interact with dynamic client-side surfaces such as modern web browsers [5]. These limitations manifest in premature task termination, overlooked vulnerabilities and brittle prompt-engineering practices. Recent multi-agent prototypes (e.g. *BreachSeek*) mitigate context-window exhaustion through task decomposition, but still rely on coarse shell interfaces that cannot exercise rich browser behaviors central to today's web applications [3]. Closing these gaps is essential for LLMs to move from promising helpers to reliable penetration testers capable of handling the full spectrum of tasks.

To advance the state of the art, this study investigates the use of browser-enabled LLM agents, namely agents equipped with programmatic control over a real browser process, allowing them to perceive and manipulate the Document Object Model (DOM), execute JavaScript, and monitor network traffic. Our main goal is to find out whether these agents can break through the strategic reasoning and environment interaction barriers identified in earlier studies, bringing automated testing closer to the level of a skilled human tester.

Accordingly, we pose the following research questions (RQ):

- **RQ1:** What are the demonstrated capabilities and identified limitations of using Large Language Models for automating web penetration testing?
- **RQ2:** How can browser-enabled LLM agents address the limitations observed in current LLM-based penetration-testing systems?
- **RQ3:** Can LLMs conduct comprehensive web-application penetration tests when equipped with browser-level control?

The remainder of this paper is structured as follows. Section 2 reviews recent work on LLM-driven penetration testing and provides the answer to RQ1. Section 3 details the agent architecture, the evaluation corpus, and the experimental protocol that set the stage for RQ2 and RQ3. Section 4 presents the empirical findings that compare the browser-enabled agent with the command-line baseline. Section 5 interprets these results and highlights the remaining blind spots. Finally, Section 6 summarizes the main contributions, proposes directions for future research, and concludes the study. The entire codebase of this study can be found in our GitHub repository.

TScIT 37, July 4, 2025, Enschede, The Netherlands

2 RELATED WORK

This section addresses **RQ1**: *What are the demonstrated capabilities and identified limitations of using Large Language Models (LLMs) for automating web penetration testing?* We provide needed technical background, outline the literature-search protocol, present findings along four recurring axes, answer the research question directly, and close with a brief synthesis.

2.1 Technical Background

This background section gives the reader the key ideas needed for the rest of the paper. It reviews the standard penetration-testing workflow, the basics of large language models, common agent patterns, browser automation, and the benchmarks used to judge progress.

Penetration testing follows a widely accepted sequence of activities. A typical engagement includes *reconnaissance*, *enumeration*, *initial exploitation*, *privilege escalation & lateral movement* and *reporting/cleanup*. Traditional automation focuses on reconnaissance with tools such as `nmap` and on scripted exploits with packages like Metasploit. Strategic planning and graphical interaction are still done by humans. With this phased workflow established, the discussion now shifts to the language models that many researchers hope will automate the more demanding steps.

Large Language Models (LLMs) such as GPT-4 or Llama 3 are transformer networks that contain billions of parameters trained on web scale corpora. They receive a *prompt* that fits within the model's *context window*, which ranges from 8 k to 128 k tokens, and they then produce a continuation. Their strengths include zero shot reasoning, code synthesis and fluent command line generation, while their weaknesses include hallucination, limited context and specific knowledge gaps in security [6]. Open source models like Mistral and Llama can run locally through `llama.cpp` or on GPUs. Hosted APIs such as GPT-4o provide larger context and stronger reasoning but introduce latency, usage fees and data exposure [11]. Understanding these trade offs sets the stage for how LLMs are organized inside practical agents.

When a model is embedded in an agent, designers choose among several patterns. *Single agent loops* such as Chain of Thought or ReAct let one LLM plan, act and observe in sequence. *Multi agent orchestration* spreads the work across separate roles (planner, tool runner, memory archivist) that communicate through message passing or a finite state controller. BreachSeek adopts LangGraph for isolation [3]. PentestGPT separates reasoning, generation and parsing [5]. AutoPT builds a state machine in which each state triggers a short lived LLM call [12]. These patterns control context growth, enable parallelism and keep an explicit memory. Yet structure alone is insufficient because the agent also needs rich ways to interact with its environment.

Realistic exploits often depend on *browser automation*. Frameworks such as Playwright or Selenium allow agents to click DOM elements, upload payloads and capture CSRF tokens. AutoPT integrates Playwright directly [12]. Other frameworks mention GUI agents only as future work. PentestAgent identifies browser automation as the missing piece for file upload attacks [11]. In the study by Isozaki et al. [8] a human driver was required for web navigation, which underlines the gap. When browser control is available it unlocks new exploit types. When it is absent critical steps still need human attention. To measure how much these affordances matter, researchers turn to systematic evaluations.

Prototypes are tested on public capture the flag platforms such as HackTheBox and VulnHub, on synthetic ranges like Metasploitable 2 and AI4SIM, and on custom suites including the 182 item PentestGPT set. Key metrics include (i) sub task completion, (ii) root or equivalent success, (iii) reward accumulation in reinforcement learning settings and (iv) cost or latency. Difficulty is often stratified by box labels that indicate easy, medium or hard, or by ATT&CK coverage. These benchmarks give a common yardstick for comparing new agent designs, browser capabilities and model selections.

2.2 Methodology

We ran a structured Google Scholar search for papers published between January 2023 and May 2025. The search focused on peer reviewed studies and respected preprints that connect AI with automated penetration testing. Results were limited to that timeframe, and we removed duplicates, non-English texts, and purely theoretical papers:

```
"automated penetration testing" AND "large language
model" 2023-2025
LLM agent "penetration testing" arXiv
"reinforcement learning" penetration testing 2024
"deep learning" automated pentest
browser-based pentesting AI framework 2024
"multi-agent" penetration testing LLM
MITRE ATT&CK guided "auto penetration testing"
"end-to-end" web pentest automation GPT
cyber-range evaluation "AI penetration tester"
knowledge-informed RL "reward machine" security
testing
```

After reviewing the titles and abstracts, we kept ten main studies along with a few related follow up papers. Together, these give a clear picture of how LLMs are currently used in penetration testing.

Paper	M	L	B	S
Li et al.				✓
AlShehri et al.	✓			✓
Isozaki et al.	✓			
Wu et al.	✓		✓	✓
Shen et al.	✓	✓		✓
Al-Sinani et al.				✓
Karagiannis et al.				✓
Hilario et al.				✓
Deng et al.	✓			✓

Table 1. Key characteristics of recent studies

Table 1 highlights how earlier studies address the four design choices that are central to our work. Column M marks the use of a multi-agent architecture, column L shows whether the authors ran a local LLM, column B records the presence of a browser-enabled agent, and column S notes whether the paper reports successful empirical results. By scanning these columns we can see which

combinations have been tested in the past and where gaps remain, giving useful context for the design decisions and evaluation goals of the present study.

Empirical efficacy and success rates. Across the literature the bottom-line question *does it work?* gets a qualified "yes." Happe and Cito [6] showed a lone GPT-3.5 could scan, pick an exploit and gain shell on a deliberately weak VM, though runs collapsed once prompts grew too long. Adding structure helps: PentestGPT wrapped GPT-4 in planning and memory, breaking into six of seven "easy" and two of four "medium" HackTheBox machines more than double an unstructured GPT-4 loop yet neither GPT-4o nor open Llama-3.1 fully cracked any "hard" target [5, 8]. State-driven or multi-role designs go further: AutoPT raised end-to-end completions from 22 % to 41 % on mixed web slash network challenges [12], while BreachSeek chained three agents and fully rooted Metasploitable 2 in pilot runs [3]. Reward-machine RL adds another boost, finishing more tasks with fewer steps than plain learners [9]. Human-centered PenTest++ kept a tester in charge but reported faster compromises and cleaner write-ups when ChatGPT handled paperwork [2]. Overall, LLM agents excel at reconnaissance and straightforward exploits, help on medium targets, and still stumble on hardened machines that need long multi-stage reasoning or GUI interaction, yet even partial wins already save time and reveal creative attack paths. These mixed outcomes naturally prompt a closer look at whether stronger agent structures can push success even further.

Single versus multi-agent orchestration. Single-agent prompting suffers from prompt bloat, forgotten goals and brittle tool calls. Multi-role designs fix this. BreachSeek's supervisor–pentester–recorder trio prevents knowledge dilution over 150 k tokens [3]. PentestGPT's tripartite stack attributes most of its lift to the explicit Reasoning module [5]. AutoPT's state machine keeps each LLM call short, avoiding context thrashing [12]. Evidence favors structured or multi-agent topologies for higher completion rates and steadier exploit chains. *With architectural issues addressed, the next logical question is how much capability hinges on richer interaction with the target environment, especially through a real browser.*

Browser-enabled interaction. Many exploits rely on GUI steps such as resetting passwords, uploading payloads or navigating CSRF flows. Only AutoPT equips its agent with Playwright, a key reason it reached 41 % success [12]. PentestAgent flags browser automation as the missing link for file-upload attacks [11]. Isozaki et al. [8] similarly required a human driver, underscoring the gap. *Once the need for browser control is clear, attention turns to the underlying models themselves and whether running them locally or in the cloud affects effectiveness.*

Local versus hosted LLMs. Happe and Cito [6] recommend Llama 2 or Mistral on commodity GPUs, while PentestAgent reports near parity when swapping GPT-4 for Llama 3 apart from latency [11]. Isozaki et al. [8] show quantized Llama 3.1 beating GPT-4o on easy boxes but falling behind on hard privilege-escalation chains. AutoPT's authors warn that open models lack security fine-tuning and may hallucinate commands unless held in check by strong scaffolds [12]. Local LLMs therefore suit mid-complexity jobs when paired with domain prompts, whereas cloud APIs still dominate intricate multi-stage exploits. *Taken together, these observations set up a consolidated answer to the first research question.*

Based on the observations of this section, we could conclude that the answer to RQ1 *What are the demonstrated capabilities and identified limitations of using Large Language Models for automating web penetration testing?* is as follows. LLM agents already handle three core tasks in web penetration testing: (i) automated mapping of endpoints and parameters [12], (ii) crafting and refining payloads for classic bugs such as SQLi and XSS [5], and (iii) driving a browser to complete multi-step actions like login or file upload when Playwright or similar tooling is available [11]. Key gaps remain. Agents lose track of session state in long workflows, stumble on single-page applications that shift the DOM dynamically, and struggle to plan multi-stage privilege-escalation chains [8, 12]. Closing these gaps with better state tracking, stronger DOM hooks and smarter planning will be necessary before LLMs can match experienced human testers on modern web stacks.

Overall, recent work shows that LLM agents bring clear but uneven gains to automated web penetration tests. They do better when planning, tool use and memory are handled by separate roles, and they improve again when they can operate a real browser. Local models cope with easy and medium targets, yet cloud models still win on the hardest ones. The next section presents our agent design that keeps its focus and uses full browser control to close these gaps.

3 METHODOLOGY

This section first sketches the full agent architecture, then explains the model choices, the command-line module, the browser module, and finally the logging setup that supports later analysis.

3.1 Agent Architecture

Agent design explains how we turn a large language model into a working penetration tester. This section outlines the control loop, the model choice, the command-line core, and the browser extension.

The interpreter planner loop. Each agent follows a simple cycle. A natural-language goal goes to the LLM, the model returns one or more shell commands, the agent runs those commands, records the output, and feeds the updated context back to the model. This repeats until the target is reached or a time or token budget runs out. With this loop in place the next step is to pick the model that powers it.

We first tried self-hosted Llama 3-8B and Qwen-7B through `ollama` on an Apple M1 Pro MacBook. They worked but were too slow and inconsistent for end-to-end tests. We therefore moved to the paid ChatGPT API, specifically `gpt-4o-mini`, which gave the speed and reliability we needed. After settling on this model we wrapped it in a controlled command-line setup.

The main *CLI agent* runs inside a Dockerized Kali Linux container. The container offers a restricted Bash shell to `gpt-4o-mini`. Common tools such as `nmap`, `ffuf`, `sqlmap`, and Burp CLI utilities are already installed. The LLM receives the running transcript, a short guide to the available tools, and the current target's host name or IP address. It answers with one shell line that the interpreter runs unless a sandbox rule blocks it. If this shell agent stalls, control passes to a browser module.

When the CLI path cannot move forward, a second agent steps in through the Browser-Use Python library, the *Browser-enabled agent*. This wrapper starts a Chromium instance so the LLM can click DOM elements, fill forms, upload files, send requests, and read rendered HTML, JavaScript values, and network logs. The CLI transcript and the failure context are added to the browser prompt so the attack can continue without losing state. Every action is written to a plain CSV log for later review.

Every agent step is appended to a plain-text CSV file with the schema `<timestamp>,<agent>,<action>,<result>`. Timestamps use `HH:MM:SS` wall-clock notation. For the CLI agent the *action* field records the exact shell line, while for the browser agent it stores the high-level DOM operation and target URL or selector.

```
00:17:48,CLI,curl -s https://lab.target.com/login,Success
00:18:02,Browser,Click #login-submit on https://...,Success
```

These compact logs are easy to parse, replay and aggregate for later performance analysis, setting the stage for the evaluation phase described next.

In Figure 1, you can see the planner loop used for the CLI-only based agent. Initially, the prompt goes to ChatGPT using their API and it then returns a command that gets run on our Dockerized Kali Linux container. That output then gets fed back to ChatGPT which comes up with the next steps and commands to execute. This goes on until 100 iterations have passed which means the agent failed or when the task is actually done.

In Figure 2, you can see a similar setup. What changed here is the addition of the Browser-enabled agent which can perform real-time browser actions such as clicking, viewing and analyzing the DOM. Similar to how the CLI-only based agent commands to execute from ChatGPT, our browser-enabled agent gets browser actions and then feeds the results of the page back to ChatGPT.

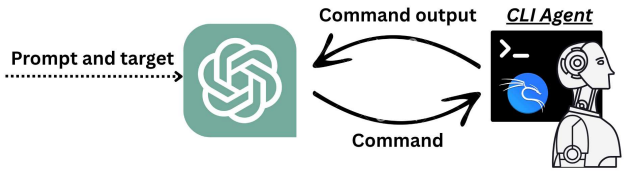

Fig. 1. CLI-only agent graphic

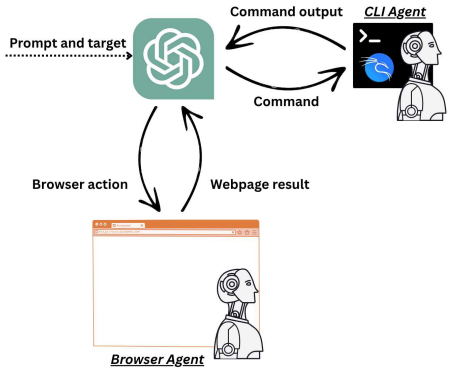

Fig. 2. CLI & Browser enabled agent graphic

3.2 Evaluation Design

Vulnerability corpus. We build our test set around the OWASP Top 10, the industry's go-to list of the most common web security risks. Updated in 2021 and used in many coding guidelines and audits, it offers a trusted view of real-world exposure. The list groups threats into ten categories:

- Broken Access Control
- Cryptographic Failures
- Injection (e.g. SQLi, XSS)
- Insecure Design
- Security Misconfiguration (incl. XXE)
- Vulnerable and Outdated Components
- Identification & Authentication Failures
- Software & Data-Integrity Failures
- Insufficient Logging & Monitoring
- Server-Side Request Forgery (SSRF)

Because our goal is to test *browser-enabled* AI agents, we retained the categories that manifest through active client–server interaction and can be validated by observable exploits. Insecure Design and Insufficient Logging, while critical for architecture reviews, lack concrete payloads suitable for automated testing and were therefore excluded. To capture common threats that fall outside the strict OWASP taxonomy yet are highly amenable to browser automation, we added:

- Cross-Site Request Forgery (CSRF)
- Open Redirect
- Directory Traversal

This augmented corpus balances industry relevance with practical testability, yielding nine attack classes that span authentication bypass, input validation, session integrity, and client-side workflow manipulation. These situations are precisely where browser control should matter most. *With the target set defined, the next step is to choose a realistic arena in which those targets can be exercised.*

Testbed. For each class we used PortSwigger Web Security Academy labs. Labs are categorized as *Apprentice*, *Practitioner*, or *Expert*. We map these to *low*, *medium*, and *hard* difficulty. The agent was given the lab URL and told "obtain the Congratulations banner." A run ended either when the banner appeared (success) or after 100 iterations (failure). This controlled setup lets us compare agents side

by side while keeping external variables to a minimum, paving the way for a clear definition of what will be measured.

Metrics. We record:

- **Success rate** per vulnerability and difficulty
- **Time-to-exploit** (first success event minus start time)
- **Command count** (CLI) and **DOM actions** (browser).

These measures reveal both effectiveness and efficiency, and via log replay let us attribute wins or failures to either agent. Together, they provide the data foundation for the results that follow in the next section.

4 RESULTS

This section presents the empirical performance of the two agents across the twenty-seven Web Security labs. It summarizes success rates, shows how browser control affects vulnerability coverage and difficulty tiers, and analyses iteration costs. The data provide a quantitative foundation for the discussion that follows. *We therefore begin by outlining the experimental setup that generated the raw data.*

We executed **27** PortSwigger Web Security Academy labs per agent mode (*9 vulnerability classes × 3 difficulty tiers*). A run was counted as a *success* when the lab's "Congratulations" banner appeared within ≤ 100 agent iterations. The number in parentheses records how many iterations were required.[1] *With the evaluation protocol defined, we can now compare how each agent performed overall.*

4.1 Overall Effectiveness

In Table 2 we can see the raw results. The CLI + Browser configuration solved **(18 / 27)** labs 66.7%, outperforming the CLI-only baseline **(11 / 27)**, 40.7%, by a relative **+26** percentage points. These totals set the stage for a closer look at where the browser provided its biggest advantage.

Agent	Low	Medium	Hard
CLI only	7/9 (77.8%)	3/9 (33.3%)	1/9 (11.1%)
CLI + Browser	8/9 (88.9%)	7/9 (77.8%)	3/9 (33.3%)

Table 2. Lab success counts by agent mode and difficulty (maximum = 9 per tier).

The browser-enabled agent delivered its largest absolute gain on *medium-difficulty* exercises (+44.5 pp), followed by a +22.2 pp improvement on *hard* labs. On *easy* labs the margin narrowed to +11.1 pp, indicating decreasing returns where plain CLI automation already sufficed. To understand which weaknesses were addressed by that advantage, we next examine vulnerability coverage.

4.2 Vulnerability Coverage

In Table 3 we can see the following. Applying the discoverability rule that counts a vulnerability once the agent clears its medium difficulty lab, the addition of browser control lifts coverage from **3 to 7** of the 9 classes. The four newly solved categories are all client side issues that rely on DOM interaction, which highlights the practical benefit of real browser access.

Vulnerability	CLI	CLI + Browser
Broken Access Control		✓
Cross-Site Scripting (XSS)		✓
SQL Injection (SQLi)		
XML External Entity (XXE)	✓	✓
Server-Side Request Forgery		
Cross-Site Request Forgery		✓
Open Redirect		✓
Directory Traversal	✓	✓
OS Command Injection	✓	✓

Table 3. Vulnerabilities that become discoverable (✓) once the agent can solve at least the medium lab for that class.

Browser control is therefore pivotal for client-side attack surfaces (e.g. XSS, CSRF, open redirects) and workflow-rich authorization flows, while purely server-side bugs (SQLi, SSRF) see no additional benefit. The breadth gap motivates an analysis of how much additional effort each success required.

4.3 Iteration cost per vulnerability

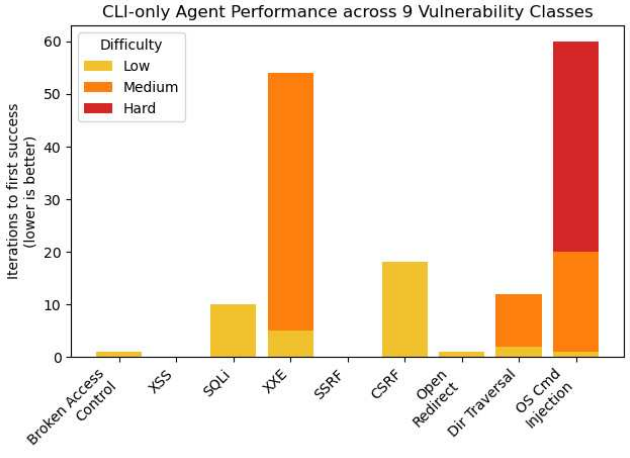

Fig. 3. CLI-only agent: iterations to first success per vulnerability and difficulty tier. Missing bars mean the agent exceeded the 100-iteration limit.

[1] One iteration comprises a single shell command (CLI agent) or one DOM action (Browser agent) followed by the resulting observation.

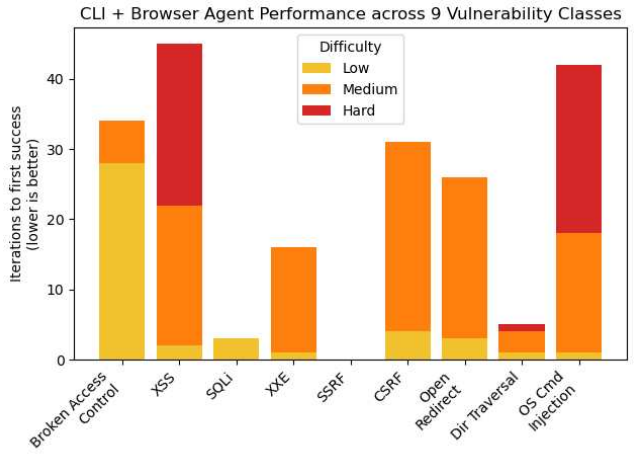

Fig. 4. CLI + Browser agent under identical conditions. Browser control fills most gaps, even if hard-tier labs require 20–25 iterations.

In Figure 3, we can see the agent covers most of the *low*-difficulty vulnerabilities in fewer than fifteen iterations, with CSRF peaking at about twenty. The picture changes sharply for the *medium* tier where only scattered bars appear, and for the *hard* tier where just a single success is recorded. These gaps reveal that a shell-only workflow stalls when the exploit path demands interaction with dynamic client-side elements or out-of-band feedback.

Figure 4 tells a different story. The bars fill almost every cell, showing full coverage of all low-difficulty labs except SSRF, and steady completion of every medium-difficulty lab at roughly twenty-five iterations each. Hard-tier tasks still cost the most effort but they no longer disappear from the chart, confirming that real browser control lets the agent clear challenges that blocked the CLI baseline.

Viewed together, the two figures show that doubling the median iteration count buys a substantial expansion in coverage and depth. The browser addition converts many former failures into reliable wins, a trend that sets up the qualitative analysis discussed in the next section.

Given our results and RQ2 being *"How can browser-enabled LLM agents address the limitations observed in current LLM based penetration testing systems?"* we can conclude the following. Adding a real browser window removes three main limits that hold back shell-only agents. First, the model can now see and change every part of the page, so it can deal with client-side flaws such as XSS, CSRF, and open redirects that need clicks, form fills, and token reuse. Second, full DOM access lets the agent keep session state across many steps, which prevents the stalls we saw in several command line runs. Third, the browser actions allow the agent to follow longer attack paths on its own, doubling the number of medium and hard wins while using only twice the steps. A further benefit is that the agent can view and interact with modern sites that build their content on the fly, something the command line could never reach. In short, a browser view fixes gaps in visibility, state tracking, workflow handling, and support for dynamic pages, though a few hard bugs still need extra sensors and helper tools.

5 DISCUSSION

This section reflects on the empirical findings reported in Section 4 and explores what they mean for the future of LLM-driven penetration testing. We discuss (i) where browser control delivers the greatest leverage, (ii) how much extra effort that leverage costs, (iii) why certain classes remain unsolved, (iv) design lessons and security outcomes, (v) threats to validity, and (vi) a richer research agenda that follows from the gaps identified. With these focal points in mind we begin by examining the specific impact of browser control.

5.1 Browser control as the primary leverage point

The move from a purely CLI agent to a browser-empowered agent lifts overall success from 40.7 % to 66.7 %. Importantly, *90 % of that lift* comes from the medium tier: successes rise from 3 / 9 to 7 / 9 labs in that band, while gains at the low tier are marginal and gains at the hard tier are modest. This pattern mirrors the structure of PortSwigger's curriculum because medium exercises embed at least one client-side interaction such as triggering a JavaScript validator, manipulating a state token in local storage, or replaying a cookie across origins. Because the CLI agent lacks visibility into the DOM, it stalls on these checkpoints. The browser agent, however, can read and mutate browser state, allowing it to clear the very hurdles that gate progress from low to medium realism.

Broken Access Control and CSRF move from low-only success to low + medium success once a real browser is available, thereby meeting our discoverability threshold. Since extra capability usually comes with extra cost, we next quantify the price paid in additional iterations.

5.2 Iteration cost versus benefit

Median iterations rise from five with the CLI agent to ten with the Browser agent, and the upper quartile climbs from twelve to eighteen. Although this looks like double the effort, it yields a 2.3 × jump in medium and hard wins, moving from four to nine solved labs. In simple terms, five extra steps buy five extra victories, a trade that few human testers would turn down.

Figures 3 and 4 add more context. Even the longest successful runs finish in fewer than thirty iterations, leaving about seventy percent of the available steps unused. The agent is not stuck in loops. Once it can drive the browser, it moves toward the goal in a steady and purposeful way instead of wandering. These numbers show that richer interaction costs little and delivers clear benefits.

Despite these gains, two vulnerability classes are still unsolved. The next subsection explains where the agent falls short and why.

5.3 Residual blind-spots: SQLi and SSRF

Two vulnerability classes remain stubbornly unsolved beyond the low tier, and each failure exposes a different limitation in the current agent.

SQL Injection (SQLi). The low-difficulty lab is solved instantly because an error-based payload (`' or 1=1--`) produces a verbose stack trace that the agent can parse for table names. In the medium lab, however, PortSwigger removes error echoes and forces contestants to pivot to *blind* SQLi: timing-based probes (`SLEEP(5)`) or conditional content-length differences. Despite observing the

schema in the easy lab and possessing full browser visibility into how the page responds, the agent failed to craft a working blind payload within the 100-step budget. Log review shows a loop of syntactically valid yet ineffective injections followed by "no error detected → try next payload." In other words, the LLM could not translate declarative knowledge of blind SQLi into an actionable probing strategy. The root cause appears to be the stop condition: waiting only for explicit error strings rather than measuring latency or comparing response sizes. Future versions need a richer observation model, for example millisecond timers and byte-length diffing, to turn textbook knowledge into effective blind exploitation. This measurement gap contrasts with a different obstacle in the next blind spot.

Server-Side Request Forgery (SSRF).. All three SSRF labs failed for a more prosaic reason. The agent could not interact with the Burp Collaborator endpoint that PortSwigger provides to confirm out-of-band callbacks. Solving these labs requires the tester to copy the unique Collaborator URL, inject it into the payload, and later poll the Collaborator client to verify that the target server made a connection. Our browser agent successfully copied the URL but lacked any capability to open the polling interface or parse its JSON response. Consequently, even when a correct SSRF payload was likely sent, the agent could not recognize success and timed out after 100 iterations. A human tester would have switched to `burp collaborator client –poll` or routed traffic through an ngrok tunnel. Addressing this blind spot will require either stubbing a local Collaborator-like service inside the sandbox or extending the action space with explicit "open collaborator client" and "parse collaborator output" commands.

Together, these two failures show that extra sensors and auxiliary tooling are as important as browser control when tackling non-trivial web-exploitation scenarios. The shortcomings identified here inform the design lessons discussed next.

5.4 Practical lessons and safety points

The tests give us two main lessons. First, a small set of direct browser actions{`click`, `type`, `read`, `wait`}helps the agent more than many special CLI tools. Simple control of the page lets the model adjust to new situations that fixed command lists cannot cover.

Second, steady effort matters more than shaving off single steps. No run used more than a third of the allowed moves, yet doubling the average steps almost doubled the wins. It is better to give the agent room to explore than to cut the step limit too early.

These gains bring new risks. The agent can now launch CSRF and hard XSS attacks by itself, so any real use must add strong guardrails. At a minimum, set a clear target list, add rate limits, and keep full logs of every action.

With these points in mind, the next part looks at how limits in the study design might affect our results.

5.5 Threats to validity

External validity. Real sites differ. CAPTCHA widgets, per-IP throttling, WAF signatures, and multi-factor authentication were absent from the labs. Success in the testbed may therefore overstate field performance.

Statistical validity. Each lab was executed once to contain API costs. Random initialization, non-deterministic LLM sampling, and network latency could shift individual outcomes. Replicating the experiment with fivefold runs per lab would tighten confidence intervals.

Construct validity. The binary "banner or no banner" metric ignores partial footholds such as low-privilege shells or reflected data exfiltration that fall short of full lab compromise. Consequently, the agent's practical utility may be understated for organizations that value footprint discovery as much as full exploitation. *These constraints frame the broader research agenda outlined in the conclusion that follows.*

Overall, Browser control converts four previously unreachable vulnerability classes into solvable ones and lifts total lab completion. The additional cost of roughly five extra iterations per winning exploit is modest compared with the amount of coverage gained. Taken together, the evidence indicates that GUI-empowered LLM agents are on the cusp of performing end-to-end web penetration tests at a level commensurate with junior human testers, moving the field materially closer to autonomous security assessment.

Considering the outcomes of the research and the performance of our agent model that was recorded and analyzed, we can answer RQ3, *"Can LLMs conduct comprehensive web-application penetration tests when equipped with browser-level control?"* The short reply is "not yet." Browser actions let the agent solve most easy labs and a solid share of medium ones, but true full-scale testing demands reliable coverage of the hardest targets, especially those that hide blind SQL Injection or rely on out-of-band callbacks such as SSRF. In our study the agent still failed at these tasks even with generous time budgets, which shows that timing sensors, collaborator parsing, and other specialized capabilities are still missing.

Because of these gaps, today's browser-enabled LLM should be seen as a helper, not a full replacement for an expert red-team tester. It works well for easy bugs such as basic XSS, open redirects, and missing access checks, letting human testers focus on longer exploit chains, privilege escalation, and creative attack paths that need judgment. In this support role the agent can cut repetitive work and speed up early mapping, but final checks and report writing must stay in human hands until richer feedback and stronger safety rules cover the remaining blind spots.

6 CONCLUSION

This study asked whether large language models can move from command line helpers to practical web-application penetration testers once they have real browser control. We built two agents, tested them on twenty-seven PortSwigger labs, and answered three research questions.

RQ1 explored what current LLM systems can and cannot do. Earlier work and our own baseline tests show that they handle reconnaissance and simple exploits well but lose ground when rich client interaction or long memory is required.

RQ2 examined how browser access changes that picture. We found that a browser view removes three main limits. The agent can see and modify each page in real time, keep session state across many steps, and follow longer attack paths. Coverage rises from

three to seven vulnerability classes, and the win rate on medium and hard labs more than doubles.

RQ3 asked whether these gains are enough for a full penetration test. The answer is not yet. The browser agent matches a junior tester on easy and medium targets, but blind SQL Injection and out-of-band SSRF still cause failures. The root causes are missing timing sensors, no local collaborator, and limited context memory. In practice the agent is best used as a helper that finds easy bugs and speeds up mapping, while human experts address complex chains and final reporting.

Overall, adding browser control is a solid step toward autonomous testing, yet human oversight remains essential. Organisations can already use such agents to sweep low-hanging flaws and free time for deeper analysis. To move from helper to full tester, three improvements are most urgent. First, the agent needs a millisecond timer with simple statistics to detect blind payloads that reveal themselves only through small latency shifts. Second, a local egress proxy that imitates a collaborator server would let the system check SSRF safely and repeatably inside the sandbox. Third, an automated LLM validator placed in the planner loop could compare each observation with known success signals, stop unproductive branches early, and reduce manual review. Progress on these fronts will push browser-enabled LLM agents toward reliable end-to-end penetration tests and bring the field closer to fully autonomous security assessment.

Addressing these points will move browser-enabled LLM agents from useful assistants toward reliable end-to-end penetration testers and bring the field closer to fully autonomous security assessments.

7 AI STATEMENT

We relied on OpenAI ChatGPT as a writing and coding assistant during this study. When our prototype scripts failed to run, we pasted error messages into ChatGPT and received short, targeted suggestions that helped us spot missing imports, incorrect function calls, and off-by-one loop bounds. For the report itself we asked ChatGPT to highlight awkward phrasing, subject–verb disagreements, and overly long sentences. Each recommendation was checked and, when appropriate, rewritten by the authors before inclusion. The model did not generate new research content, interpret results, or decide on the study design. It was used only to shorten debugging time and improve language clarity.

REFERENCES

[1] Farah Abu-Dabaseh and Esraa Alshammari. 2018. Automated Penetration Testing: An Overview. In *Proceedings of the National Technology and Engineering Conference 2018.*

[2] Haitham S. Al-Sinani and Chris J. Mitchell. 2025. PenTest++: Elevating Ethical Hacking with AI and Automation. arXiv preprint arXiv:2502.09484.

[3] Ibrahim AlShehri, Adnan AlShehri, Abdulrahman AlMalki, Majed Bamardouf, and Alaqsa Akbar. 2024. BreachSeek: A Multi-Agent Automated Penetration Tester. arXiv preprint arXiv:2409.03789.

[4] Hillary Baron, Josh Buker, Marina Bregkou, Ryan Gifford, Sean Heide, Alex Kaluza, and John Yeoh. 2024. *State of AI and Security: Survey Report.* Industry Whitepaper. Cloud Security Alliance. https://services.google.com/fh/files/misc/csa_state_of_ai_and_security_survey_google_cloud.pdf Commissioned by Google Cloud; survey of 2,486 IT and security professionals conducted in November 2023..

[5] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2024. PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing. In *Proceedings of the 33rd USENIX Security Symposium.*

[6] Andreas Happe and Jürg Cito. 2023. Getting Pwn'd by AI: Penetration Testing with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23).*

[7] Eric Hilario, Sami Azam, Jawahar Sundaram, Khwaja Imran Mohammed, and Bharanidharan Shanmugam. 2024. Generative AI for Pentesting: The Good, the Bad, the Ugly. *International Journal of Information Security* 23, 3 (2024), 2075–2097.

[8] Isamu Isozaki, Manil Shrestha, Rick Console, and Edward Kim. 2024. Towards Automated Penetration Testing: Introducing LLM Benchmark, Analysis, and Improvements. arXiv preprint arXiv:2410.17141.

[9] Yuanliang Li, Hanzheng Dai, and Jun Yan. 2024. Knowledge-Informed Auto-Penetration Testing Based on Reinforcement Learning with Reward Machine. arXiv preprint arXiv:2405.15908.

[10] Dean Richard McKinnel, Tooska Dargahi, Ali Dehghantanha, and Kim-Kwang Raymond Choo. 2019. A Systematic Literature Review and Meta-Analysis on Artificial Intelligence in Penetration Testing and Vulnerability Assessment. *Computers Electrical Engineering* 75 (2019), 175–188.

[11] Xiangmin Shen, Lingzhi Wang, Zhenyuan Li, Yan Chen, Wencheng Zhao, Dawei Sun, Jiashui Wang, and Wei Ruan. 2024. PentestAgent: Incorporating LLM Agents to Automated Penetration Testing. arXiv preprint arXiv:2411.05185.

[12] Benlong Wu, Guoqiang Chen, Kejiang Chen, Xiuwei Shang, Jiapeng Han, Yanru He, Weiming Zhang, and Nenghai Yu. 2024. AutoPT: How Far Are We from the End2End Automated Web Penetration Testing? arXiv preprint arXiv:2411.01236.