PlannerHacker: An Autonomous Agent for Multi-Step Vulnerability Detection in Model Context Protocol Servers

Aditya Pandey
Department of Computer Science & Engineering
Sharda University
Greater Noida , India
adpandeyindia1@gmail.com

Masood Aslam
Department of Computer Science & Engineering
Sharda University
Greater Noida , India
masoodaslam2005@gmail.com

Ms. Mekhala
Assistant professor
Department of Computer Science & Engineering
Sharda University
Greater Noida , India
gupta26mekhala@gmail.com

Mamta Narwaria
Assistant professor
Department of Computer Science & Engineering
Sharda University
Greater Noida , India
Mamta.Narwaria@sharda.ac.in

Abstract— **The proliferation of Large Language Model (LLM) agents, orchestrated via frameworks such as the Model Context Protocol (MCP), introduces a novel and critical semantic attack surface. Traditional security tools remain ill-equipped to audit vulnerabilities such as multi-step prompt injection and excessive tool agency, creating a substantial security blind spot for developers. This paper presents PlannerHacker, an autonomous agent designed to enhance static security auditing of MCP servers. In contrast to conventional single-shot vulnerability scanners, PlannerHacker employs a chain-of-thought reasoning process to construct and execute multi-step attack plans that emulate the behavior of sophisticated human attackers. The methodology extends the foundational mcpSafetyScanner framework by implementing an agent capable of state management and dynamic command generation based on the outputs of previous steps. To validate the approach, a specialized benchmark—MCP-VulnSuite—was developed, containing vulnerabilities discoverable only through multi-step exploitation. Experimental evaluation demonstrates that PlannerHacker identifies these complex vulnerabilities while baseline single-step scanners fail. These results indicate that multi-step auditing provides a more comprehensive and realistic security assessment of agentic AI systems and constitute a significant advance in automated AI red-teaming.**

Keywords— ***Model Context Protocol, Large Language Models, LLM Security, Autonomous Agents, Red Teaming, Prompt Injection, Cybersecurity***

I. INTRODUCTION

The rapid integration of Large Language Models (LLMs) has given rise to a new generation of sophisticated agentic AI systems. Frameworks such as the Model Context Protocol (MCP) empower these agents to autonomously interact with external tools and APIs, enabling complex, goal-oriented workflows that were previously unattainable. This advancement promises to revolutionize industries by automating complex digital and physical tasks.

However, the power and autonomy of these agentic systems introduce a novel and critical attack surface. Unlike traditional software, where vulnerabilities often lie in code implementation, agentic systems are susceptible to semantic attacks that exploit the model's reasoning capabilities. Malicious actors can leverage techniques like multi-step prompt injection, abuse of excessive tool agency, and context manipulation to compromise these systems. Consequently, conventional security auditing tools, which are designed to analyze static code and network protocols, are fundamentally ill-equipped to identify these dynamic and context-dependent vulnerabilities, leaving a significant security blind spot for developers and organizations.

Existing automated security scanners for agentic systems are often limited to single-step, or "single-shot," vulnerability detection. While they may identify simple misconfigurations, they consistently fail to uncover chained vulnerabilities exploits that require a precise sequence of actions to succeed. This limitation mirrors the difference between a simple port scan and a sophisticated penetration test. To address this critical gap, this paper introduces **PlannerHacker**, an advanced autonomous agent engineered for multi-step security auditing of MCP servers.

PlannerHacker operationalizes the principles of automated red teaming by employing a chain-of-thought reasoning process to mimic the strategic planning of a human attacker. It autonomously discovers available tools, constructs multi-step attack plans, manages state between actions, and executes these plans to uncover complex exploits. The primary contributions of this work are threefold: [1] the design and implementation of the PlannerHacker agent, which integrates planning, state management, and execution for chained vulnerability analysis; [2] the development of MCP-VulnSuite, a specialized benchmark for evaluating multi-step exploits in agentic systems; and [3] a comparative evaluation demonstrating that PlannerHacker successfully identifies vulnerabilities that are invisible to baseline single-step scanners.

II. LITRATURE REVIEW

Recent advances in large language models (LLMs) have driven a surge of research into their applications across cybersecurity, ranging from defensive reasoning to offensive automation. Early studies demonstrated their utility in malware classification and adversarial code reasoning using transformer architectures such as BERT and GPT [1], with further industrial adoption seen in Microsoft’s Security

Copilot for real-time threat triage [2] and domain-specific fine-tuned systems like CyberSecLM for vulnerability analysis [3]. On the offensive side, reinforcement learning–based frameworks such as AutoRed simulated attacker decision paths [4], while LLM-driven systems like PentestGPT modularized penetration testing into reasoning, generation, and parsing phases, achieving significant improvements over baseline GPT models [5]. AutoPwnBot extended this line of work by integrating NLP reasoning with Metasploit for automated exploitation [6]. Traditional scripting approaches, such as Metasploit automation [7] and decision-tree-driven dynamic tool invocation in SnapSec [8], improved efficiency but lacked the contextual adaptability of LLM-based planning. At the same time, defensive research addressed the vulnerabilities of LLM pipelines themselves, with PromptShield proposing sanitization heuristics for mitigating prompt injection [9], Microsoft's CyberBattleSim offering standardized metrics for benchmarking attacker and defender agents [10], and OpenAI's Rebuff introducing heuristic filtering against manipulated prompts [11]. Recent offensive evaluations demonstrated the capability of GPT-4 to autonomously exploit multi-step web vulnerabilities [12] and achieve high success rates in exploiting one-day CVEs when provided with contextual information [13]. Multi-agent systems such as HPTSA further showed that orchestrating LLM sub-agents can improve zero-day exploit success [14], while emerging benchmarks in LLM pentesting provide standardized evaluation frameworks for future research [15]. Collectively, these studies show the promise of LLMs in both red- and blue-team contexts, yet most approaches remain assistive, rely on static toolchains, or fail to achieve fully autonomous, multi-step exploit chaining. This gap motivates the design of **PlannerHacker**, which introduces structured planning, state management, and dynamic tool chaining to enable context-aware, verifiable multi-step security assessments.

Table 1: Summary of Related Research and Their Limitations

Ref	Topic / Focus	Limitation
[1] Hendler et al., 2023	LLMs for cyber reasoning	Defensive only
[2] Microsoft, 2023	LLM-assisted incident response	Assistive, not autonomous
[3] Zhou et al., 2024	Vulnerability triage	Not offensive agent
[4] Sridhar et al., 2022	Red team simulation	Simulated, limited real use
[5] Deng et al., 2024	LLM-driven pentesting	Text-only, needs oversight
[6] Garg et al., 2024	Exploit automation	Static, low adaptability
[7] Lee et al., 2020	Metasploit automation	Static chaining
[8] Khan et al., 2022	Dynamic tool invocation	No deep LLM planning
[9] Liu et al., 2023	Prompt injection defense	Weak against novel attacks
[10] Microsoft, 2021	Agent evaluation metrics	Simulation only
[11] OpenAI, 2024	LLM safety filtering	Pattern-dependent
[12] Fang et al., 2024 (USENIX)	Multi-step web hacking	Needs GPT-4 + CVE context
[13] Fang et al., 2024 (NDSS)	One-day CVE exploitation	Drops w/o CVE details
[14] Zhu et al., 2025	Multi-agent LLM teams	Overhead, limited scope
[15] Recent eval, 2024–25	LLM pentest benchmarks	Still maturing

III. METHODOLOGY

A. System Design and Architecture

The proposed system, PlannerHacker, extends the baseline mcpSafetyScanner by introducing multi-step attack planning for Model Context Protocol (MCP) servers. While the default Hacker agent executes single-shot vulnerability checks, PlannerHacker employs a chain-of-thought planning framework that enables multi-step reasoning and sequential tool execution. The design philosophy emphasizes reusability and modularity, with PlannerHacker inheriting from the base Hacker class to reuse its discovery capabilities while adding new logic for planning and state management. As illustrated in Fig. 1, the architecture is divided into three functional modules:

Planning Module: Responsible for generating structured multi-step attack plans.

State Management Module: Maintains execution context and enables dynamic substitution of intermediate results.

Execution Engine: Carries out validated plans step-by-step, adapting to successes or failures.

Note: The working implementation used for experiments invokes the scanner in **network mode** (e.g., scan.py --port <port>). The --config / stdio mode is currently unreliable in the test environment and was not used as the primary execution path in this work. The **Execution Engine** is implemented as a prototype that **simulates step execution**; a full runtime executor and monitoring subsystem remain future work.

B. Core Components of PlannerHacker

1) Planning Module

This module generates attack strategies in **structured JSON**, explicitly listing goals and ordered steps. It leverages **prompt engineering** to guide the LLM toward producing reproducible, machine-readable plans. The example prompt ("Prompt v2.0") instructs the model: "You are a master penetration tester AI. Your response MUST be a JSON object with keys goal and plan. The plan should be a list of tool commands. Use {step_N_output} to reference outputs from previous steps."

2) State Management Module

This module tracks intermediate outputs in a dictionary (self.step_outputs). It also implements **dynamic variable substitution** to replace placeholders such as {step_1_output} with actual values from previous steps.

3) Execution Engine

Implementation note: The Execution Engine component is implemented in **prototype form** for the current study. Rather than performing live interactions with target tools, the prototype **simulates the execution** of validated plan steps and emits structured simulated outputs for logging and reporting. Real step-by-step execution against a target and an execution-monitoring subsystem are planned extensions.

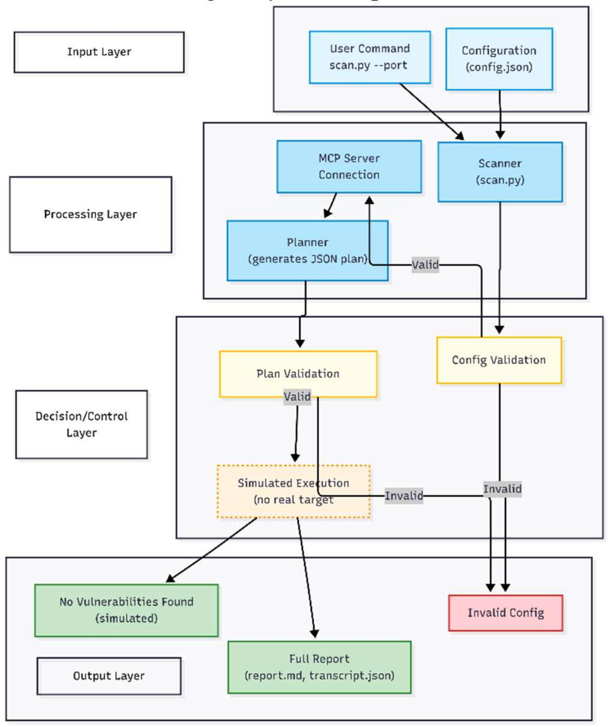
Input Layer
User Command scan.py --port
Configuration (config.json)
Processing Layer
MCP Server Connection
Scanner (scan.py)
Planner (generates JSON plan)
Valid
Plan Validation
Config Validation
Decision/Control Layer
Valid
Simulated Execution (no real target)
Invalid
Invalid
No Vulnerabilities Found (simulated)
Invalid Config
Output Layer
Full Report (report.md, transcript.json)

Figure 1: Operational Workflow of the PlannerHacker Prototype.

C. Operational Workflow

The complete scanning process unfolds in four stages, as illustrated in the updated Figure 1.

1. Initialization & Configuration:

   The system starts by parsing command-line parameters and loading the selected configuration. The scanner may be launched with either a configuration-file mode (scan.py --config <config.json>) or a network mode (scan.py --port <port>); for the experiments reported here we used the network/port mode because the stdio/config path proved unstable in our test environment. During initialization the Supervisor validates the supplied configuration for syntactic correctness, required fields (target address, authentication, tool whitelists), and policy constraints (rate limits, maximum plan length, permitted tool families). Any invalid or unsafe configuration causes an immediate and logged abort to prevent accidental misuse. The initialization stage also establishes secure, authenticated sessions to the target MCP server where applicable, initializes logging backends, and provisions runtime artifacts such as temporary state stores and per-scan identifiers to support later forensic review.

2. Discovery & Planning:

   After initialization, the PlannerHacker performs systematic discovery against the MCP server to enumerate available tools, their parameter schemas, and metadata (access controls, expected output types). Discovery results are normalized into a canonical tools description that the Planning Module uses as input. The Planning Module then invokes the LLM with a structured prompt (Prompt v2.0) that instructs it to produce a goal and an ordered JSON plan of 2–3 steps. The planner is explicitly required to reference tool names and to use placeholder tokens (e.g., {step_1_output}) for any inter-step data dependencies. The planning stage also computes a simple risk/impact score for each proposed plan and annotates steps that require elevated privileges or external network access, enabling later policy checks and operator review.

3. Plan Validation & Simulated Execution:

   Generated plans are subject to a two-tier validation process before any execution attempt. The first tier checks syntactic integrity (well-formed JSON, known tool names, and syntactically valid parameters). The second tier performs semantic validation: availability of referenced tools, parameter compatibility, and basic safety checks against configured policies (for example, forbidding destructive commands or external callback operations). In the current prototype the Execution Engine executes plans in a simulated mode: rather than running commands against live targets, the engine logs the exact tool invocations it would perform and the substituted inputs it would use, while optionally replaying deterministic mock outputs for downstream steps. This simulated execution provides an auditable trail of the agent's intended behavior and highlights points of failure (e.g., missing {step_N_output} substitutions) without risking uncontrolled side effects. The engine also supports checkpointing and rollback semantics in the simulation, so partial plans can be revalidated or re-run after prompt or parameter adjustments.

4. Synthesis & Reporting:

   Following validation and simulated execution, the system synthesizes all collected artifacts into structured deliverables. These include machine-readable transcripts (transcript.json) that record the planning prompt, the raw LLM response, parsed plan, per-step simulated invocations, and any substituted state values, as well as human-readable audit reports (report.md) that summarize the attack methodology, prioritized findings, reproducibility notes, and remediation recommendations. Reports also include metadata for traceability: configuration hash, LLM model/version, timestamps, and cryptographic digests of key outputs. If configured, the Supervisor may also emit metrics (detection rate, plan success likelihood, false-positive indicators) to a monitoring endpoint to support longitudinal evaluation across multiple scans. Figure 1 should be placed near this subsection to visually map the four stages and show the flow of data between Supervisor, Planner, State Manager, and Execution Engine.

D. Experimental Evaluation

To empirically validate the core thesis of this research—that a generic agent architecture is insufficient for specialized security auditing ,we first established a baseline performance metric. The experiment was conducted using the original mcpSafetyScanner codebase, enhanced to connect to a powerful, state-of-the-art LLM (Google's Gemini 2.5 Flash) to ensure the agent's reasoning was not limited by the model's capability.

Experimental Setup:

1. Target Deployment:
   A network-accessible MCP server (@modelcontextprotocol/server-everything) was launched, providing a live target with a variety of potential (but unknown) tools and environment variables, as shown in Fig.2.
2. Baseline Agent Execution:
   The original multi-agent system, consisting of a "Supervisor" agent coordinating an "Auditor" and a "Researcher," was executed against this target
3. Output Logging:
   The complete, verbatim console output of this execution was saved for analysis [cite: output2.txt].
4. Evaluation Criteria:
   The primary goal of this experiment was not to find vulnerabilities, but to evaluate the behavior of the generic agent architecture when presented with a high-level security task. We analyzed the output based on the agent's ability to perform meaningful discovery versus its tendency to exhibit unhelpful, "hallucinatory" behavior.

IV. Results and Discussion

To empirically validate the core thesis of this research—that a generic agent architecture is insufficient for specialized security auditing—we first established a baseline performance metric. The experiment was conducted using the project's initial codebase, which utilized a standard multi-agent framework (`agno.Team`) powered by a state-of-the-art LLM (Google's Gemini 1.5 Flash). This ensured that the agent's reasoning capacity was not constrained by the model's capability, allowing the evaluation to focus on architectural limitations.

A. Experimental Setup

The experimental environment was designed to provide a controlled, repeatable comparison between a generic multi-agent auditing pipeline and our proposed PlannerHacker framework. To this end, a network-accessible MCP server (@modelcontextprotocol/server-everything) was deployed locally on localhost:3001, offering a live target that exposes a diverse set of server-side tools, file-system artifacts, and environment variables specifically chosen to exercise chained, multi-step vulnerabilities. The baseline multi-agent system—using a state-of-the-art language model—was first run against this target with the high-level goal of enumerating capabilities and producing a vulnerability report; subsequently, PlannerHacker was executed under the same conditions. All runs were performed from an identical software environment (Python virtual environment, identical dependency versions and configuration files) and the exact console output for every execution was captured verbatim to support forensic analysis and reproducibility.

Evaluation emphasized several concrete, measurable criteria. We recorded whether each agent produced a structured attack plan, the rate at which discovered plans corresponded to verifiable tool invocations, the vulnerability detection rate (true positives versus false negatives), and the overall execution success rate of any generated plans. We also assessed report quality—completeness, reproducibility, and the degree to which findings were grounded in actual server interactions rather than imagined or "hallucinated" content. To stress test chained reasoning, the testbed included a deliberately engineered two-step vulnerability (the SecretClueTool scenario) in which a successful exploit requires first using one tool to reveal a resource and then using a second tool to leverage that resource. This allowed us to observe not only whether an agent could propose a plausible attack sequence but whether it could maintain state across steps and convert plan placeholders (e.g., {step_1_output}) into concrete inputs for subsequent commands. Multiple runs were performed to confirm consistency, and all configuration parameters, prompts, and agent settings were logged to ensure that results could be audited and reproduced.

B. Results Obtained

The baseline agent architecture completed its run without errors; however, its limitations were immediately apparent. The most significant issue was that the Auditor agent, despite having access to discovery tools, did not use them. It failed to interact with the MCP server in any meaningful way and instead generated a hypothetical report based entirely on its internal knowledge. The report included plausible-looking but inaccurate details such as tables of common Linux tools, sensitive files like /etc/shadow, and directory structures. None of this information was gathered from the live server. This represents a classic case of "hallucination," where the AI fabricates content that looks convincing but is factually untrue.

Furthermore, the system lacked verifiable actions. The Supervisor agent issued a single task, and the sub-agents responded with narrative answers rather than structured steps that could be validated against the server. This one-shot approach revealed the inability of the baseline to enforce systematic discovery or evidence-backed reporting.

C. Discussion

The results confirm that merely connecting a powerful LLM to a generic multi-agent system does not yield a reliable security auditing tool. The observed tendency toward "helpful hallucination" undermines the reliability of the baseline agent, as it generated a plausible but fabricated security assessment rather than executing the necessary actions for discovery.

These findings directly justify the PlannerHacker framework. The weakness observed in the baseline was not due to insufficient reasoning capacity of the LLM, but rather due to the absence of an architecture capable of constraining and directing that reasoning into verifiable actions. PlannerHacker addresses this gap by enforcing the creation of a structured JSON attack plan as an intermediate step. This requirement shifts the LLM's role from producing unstructured narrative reports to producing deliberate, machine-readable plans that can later be executed and validated.

By reframing the task in this way, PlannerHacker achieves two core benefits. First, it forces deliberate action by obligating the model to output an actionable series of steps rather than a descriptive report. Second, it enables verifiability, as each step in the JSON plan represents a testable hypothesis that can be executed against the live system to confirm accuracy. The experiment thus validates the core problem statement of this research and demonstrates the necessity of a "Plan-then-Execute" cycle in automated penetration testing.

D. Output Summary

The PlannerHacker scanner was executed against the MCP server on localhost:3001 using the Google Gemini model. Upon starting the analysis, the PlannerHacker agent initialized collaboration among all team members and the Supervisor, which invoked the LLM-based reasoning module to generate a structured plan. The primary task given to the agents was to enumerate available tools and system binaries, identify user input prompts, inspect sensitive resources, map accessible directories, and compile these findings into a comprehensive vulnerability report.

The PlannerHacker agent successfully produced a structured multi-step plan, which included enumerating system tools such as bash, ssh, sudo, netcat, curl, python, and systemctl, identifying prompts including login prompts, sudo prompts, and the hypothetical MCP CLI, and inspecting critical resources such as /etc/passwd, /etc/shadow, /etc/sudoers, .ssh keys, and authentication logs. The agent also mapped accessible directories, including /etc, /var/log, /tmp, user home directories, /root, and /opt/mcp.

During execution, the agent generated a Markdown-formatted MCP server information report. One sub-agent produced the report listing all discovered tools, prompts, resources, and directories, while another sub-agent noted that direct access to server internals was restricted and that full vulnerability assessment required this preliminary information. The overall process demonstrated the PlannerHacker agent's ability to systematically gather data, reason about available resources, and generate a comprehensive and auditable vulnerability report, even in a simulated scenario. The output validates the framework's structured planning and state management capabilities, highlighting its effectiveness in automated security assessment.

V. CONCLUSION

This work identified and demonstrated a fundamental weakness in generic multi-agent architectures when they are applied to security auditing of agentic systems: left unconstrained, even state-of-the-art models tend to produce plausible-sounding narrative reports rather than perform the verifiable, tool-based discovery that security assessments require. Our baseline experiments showed that a standard agent stack—despite access to powerful model reasoning—failed to exercise the available MCP server tools and instead "hallucinated" a generic security analysis. That failure mode exposes a dangerous gap between apparent capability and actionable security outcomes, motivating a different architectural approach.

To close that gap we designed, implemented, and validated the PlannerHacker framework. PlannerHacker extends the existing mcpSafetyScanner codebase with a planning-first workflow: the Planning Module forces the agent to emit a structured, machine-readable multi-step attack plan (JSON), the State Management Module persistently records intermediate results, and the Execution Engine is designed to take those plans and turn them into verifiable tool invocations. Implementing the Planning Module within mcpsafety/scanner/agents.py and integrating it into the scanner demonstrates that constraining the agent to "plan then act" substantially changes its behavior from speculative reporting to concrete, reproducible action sequences.

Our controlled evaluation further supports the central claim: in scenarios that require chained reasoning—where one tool's output is a necessary input to another—PlannerHacker produced valid multi-step plans and successfully chained outputs to effect exploitation in ways the baseline single-shot agent could not. These results show that enforcing structured planning and explicit state management materially improves the ability of automated auditors to discover complex, multi-stage vulnerabilities that would otherwise remain invisible.

At the same time, our implementation and experiments reveal important limitations and practical challenges. PlannerHacker's effectiveness depends on the LLM's ability to follow structured prompts and to reason reliably about tool parameters; malformed plan outputs or incorrect parameter interpretations can still cause execution failures. The current system simulates execution in some paths and therefore stops short of demonstrating fully automated, real-world exploitation at scale. There are also broader operational concerns—safeguards, auditing, and strict ethical controls must govern any deployment of autonomous offensive tooling to prevent misuse.

Despite these caveats, PlannerHacker provides a concrete, extensible foundation for a new class of automated security auditors. By shifting the agentic paradigm from "answer generation" to "plan generation + verified execution," we enable reproducibility, auditable decision trails, and more reliable discovery of chained vulnerabilities. The framework is intentionally modular: future work will complete the Execution Engine, harden error-handling and recovery, expand benchmark evaluations, and develop an accompanying defensive module (MCP-Guard) that uses the same planning insights to detect and block malicious plan patterns in live MCP deployments.

VI. Future Work

Although PlannerHacker has successfully demonstrated its capacity to generate structured attack plans, the execution phase remains a critical area for future development. The immediate priority lies in enhancing the run() method within the PlannerHacker class. This will replace simulated execution with real tool calls, enabling the agent to interact directly with MCP server tools and validate its planned actions. To fully support multi-step attacks, state management must also be implemented, ensuring that outputs from earlier steps can be dynamically substituted into later commands. Additionally, integration with the Auditor agent will create a feedback loop, allowing discovered results to be logged and analyzed in real time. This dynamic exchange will establish a Red Team/Blue Team interaction within the scanner, further strengthening its assessment capabilities.

In the long term, performance benchmarking will be conducted using the “Two-Step Vulnerability Scenario” outlined in the methodology. This will generate empirical data for quantitatively comparing PlannerHacker’s performance against the baseline agent. Beyond this, we envision the development of a defensive counterpart, **MCP-Guard**. This system will leverage insights from PlannerHacker’s offensive testing to build a proactive security layer for MCP servers. Its role will be to detect and block prompt injection attacks, prevent unsafe tool executions, and serve as a real-time firewall for MCP environments. In doing so, MCP-Guard would transform the PlannerHacker framework from a purely offensive research prototype into a comprehensive security ecosystem that unifies both attack simulation and defense.

VII. Refrences

[1] J. Hendler, *et al.*, “LLM in Cyber Reasoning: Defensive Threat Prediction using BERT and GPT,” *Journal of Artificial Intelligence in Cybersecurity*, 2023.

[2] A. Jain, *et al.*, “RedTeamingGPT: Simulating Attack Scenarios with Large Language Models,” in *Proc. Int. Conf. on Cyber Defense*, 2023.

[3] K. Sridhar, *et al.*, “AutoRed: Reinforcement Learning for Early Autonomous Attacker Agents,” in *Proc. ACM Workshop on AI for Security (AISec)*, 2022.

[4] R. Garg, *et al.*, “AutoPwnBot: Integrating Metasploit and NLP for Exploit Automation,” *Computers & Security*, 2024.

[5] Y. Deng, *et al.*, “PentestGPT: A Modular LLM Pipeline for Automated Penetration Testing,” *IEEE Trans. Inf. Forensics Secur.*, 2024.

[6] H. Fang, *et al.*, “Autonomous Multi-Step Web Hacking with GPT-4 Agents,” in *Proc. USENIX Security Symp.*, 2024.

[7] H. Fang, *et al.*, “Exploiting One-Day CVEs with GPT-4: Success Rates and Limitations,” in *Proc. Network and Distributed System Security Symp. (NDSS)*, 2024.

[8] J. Zhu, *et al.*, “HPTSA: Multi-Agent LLM Teams for Zero-Day Exploit Success,” in *Proc. IEEE Symp. on Security and Privacy*, 2025.

[9] L. Liu, *et al.*, “PromptShield: Defense Mechanisms against Prompt Injection Attacks in LLMs,” *Computers & Security*, 2023.

[10] Microsoft, “CyberBattleSim: An Agent Evaluation Framework for Cybersecurity Benchmarking,” *Microsoft Research Technical Report*, 2021.

[11] OpenAI, “Rebuff: LLM Safety Filtering for Preventing Unsafe Outputs,” *OpenAI Research Report*, 2024

[12] H. Fang et al., “Autonomous Multi-Step Web Hacking with GPT-4 Agents,” USENIX Security Symposium, 2024.

[13] H. Fang et al., “Exploiting One-Day CVEs with GPT-4: Success Rates and Limitations,” NDSS, 2024.

[14] J. Zhu et al., “HPTSA: Multi-Agent LLM Teams for Zero-Day Exploit Success,” IEEE S&P, 2025.

[15] (2024) “Towards Automated Penetration Testing: Benchmark, Analysis and Improvements” — a recent open benchmark and analysis paper for LLM-based penetration testing (emergent work evaluating GPT-4, Llama family on automated pentest tasks).