AdaptiveFuzz: LLM-Based Adaptive Fuzzing for Vulnerability Discovery

Vigneshwar Sundararajan
vigneshwar_ks@ucf.edu
University of Central Florida
Orlando, Florida, USA

ABSTRACT

Traditional fuzzing techniques often struggle to uncover intricate vulnerabilities that arise under dynamic and evolving software conditions. This report presents *AdaptiveFuzz*, a Large Language Model (LLM)-powered adaptive fuzzing framework that automates enumeration during penetration testing. Developed using the Model Context Protocol (MCP) [1], the system coordinates interactions between an AI-based reasoning engine and terminal execution tools within a structured agent-based architecture. By leveraging curated exploit recipes and real-time feedback, AdaptiveFuzz intelligently formulates and refines enumeration strategies with minimal manual involvement. The LLM agent, powered by models such as Qwen 2.5 Coder and Claude, emulates red team tactics, enabling context-aware decision-making and streamlined execution flow. The project culminates in a robust PoC (Proof of Concept) that generates a comprehensive exploitability summary by analysing system responses and correlating known CVEs. This work affirms the potential of LLM-guided automation in improving the precision, adaptability, and efficiency of vulnerability discovery, laying the groundwork for further advancements in intelligent security testing.

GitHub URL: https://github.com/vickie-ks/AdaptiveFuzz

1 INTRODUCTION

The growing complexity of modern software systems has brought about a parallel rise in sophisticated security threats that demand more intelligent and context-aware vulnerability assessment techniques. Traditional penetration testing and static fuzzing approaches, while foundational, are inherently limited in their ability to identify vulnerabilities that manifest under dynamic runtime conditions or are deeply embedded within application logic. These conventional methods typically rely on hardcoded test patterns or mutation-based input generation, which often fail to simulate the nuanced strategies employed by experienced human testers.

Moreover, such methods lack adaptive reasoning and struggle to modify their behaviour based on real-time output. As a result, the discovery of vulnerabilities in real-world environments often remains incomplete, leading to security gaps that could be exploited by adversaries. With the advancement of artificial intelligence, particularly Large Language Models (LLMs), there exists a compelling opportunity to transform how security testing is conducted—shifting from static execution to intelligent, learning-based fuzzing strategies.

This work proposes AdaptiveFuzz, a Model Context Protocol (MCP)-driven LLM agent designed to automate the enumeration phase of penetration testing. The system emulates the mindset of a red teamer by dynamically generating, executing, and refining enumeration steps based on terminal outputs and curated exploit recipes. By integrating tools, prompts, and real-time feedback into a unified architecture, AdaptiveFuzz serves as a proof of concept that showcases how intelligent agents can enhance the efficiency, depth, and contextual relevance of vulnerability discovery processes.

1.1 Problem Statement

Traditional enumeration and fuzzing techniques often rely on static input patterns and fail to adapt to the dynamic behaviours of modern software systems. These methods lack contextual awareness and cannot reason through multi-step attack paths the way experienced penetration testers do. As a result, they frequently miss complex or deeply embedded vulnerabilities.

To overcome these limitations, there is a need for an intelligent, adaptive approach that can generate test inputs based on real-time feedback and security logic. AdaptiveFuzz addresses this need by employing a Large Language Model integrated with the Model Context Protocol (MCP) to automate enumeration, maintain execution context, and refine strategies iteratively—emulating human-like decision-making in vulnerability discovery.

2 RELATED WORK

The AdaptiveFuzz framework builds upon recent innovations in the Model Context Protocol (MCP) ecosystem, where LLMs and structured agents are leveraged to perform intelligent, tool-driven security analysis. Several existing MCP-based implementations have demonstrated the value of this protocol in integrating AI with cybersecurity workflows:

(1) **MCP Security:** This implementation offers a Model Context Protocol server designed for querying the ORKL API to retrieve threat intelligence [2]. It enables automated analysis of threat actors, attack campaigns, and source reports within a standardized MCP agent-server interface. AdaptiveFuzz draws inspiration from this work by adopting a similar approach to tool coordination through a lightweight MCP container.

(2) **MCP Shodan:** This MCP server provides integrated access to Shodan and VirusTotal APIs, allowing for advanced network intelligence operations such as DNS resolution, vulnerability scanning, and host information lookup [3]. It demonstrates how MCP servers can act as comprehensive security assistants, a concept extended in AdaptiveFuzz to internal pentesting enumeration and local tool execution.

(3) **Illumio MCP Server:** The Illumio MCP server integrates with Illumio's Policy Compute Engine (PCE) to manage workload segmentation, label operations, and traffic flow insights [4]. While domain-specific to Illumio, it reinforces the generality and modularity of MCP-based agents, which AdaptiveFuzz uses

to coordinate terminal tools and enumeration logic in penetration testing.

(4) **MCP Security Checklist:** This project provides a security guide tailored for AI tools and MCP-based systems [5]. It offers best practices and risk assessment strategies for integrating AI-driven agents with system-level operations. AdaptiveFuzz aligns with these recommendations by maintaining process isolation and secure data handling in its proof-of-concept implementation.

(5) **Semgrep MCP Server:** Still in its early stages, this MCP server enables Semgrep to be used via agent-based orchestration for scanning codebases and identifying potential vulnerabilities [6]. Though focused on static code analysis, it demonstrates the feasibility of integrating powerful security tools under a common agent interface, a philosophy similarly applied in AdaptiveFuzz's command execution and response analysis.

3 METHODOLOGY

The proposed system, AdaptiveFuzz, is a Model Context Protocol (MCP)-driven agent designed to perform intelligent enumeration in penetration testing by leveraging the reasoning capabilities of Large Language Models (LLMs). The architecture adopts a modular container-based design that enables isolated orchestration of commands, prompt-based decision making, and dynamic result processing.

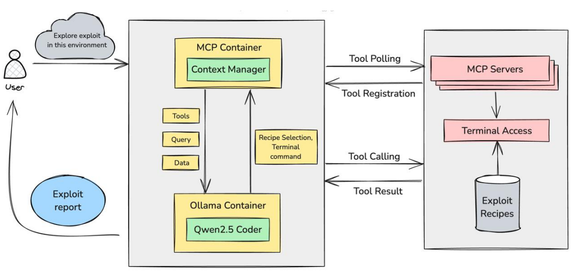

Figure 1: AdaptiveFuzz Architecture

Figure 1 illustrates the architecture of AdaptiveFuzz, which comprises four key entities: the User Interface, MCP Container, LLM (Ollama) Container, and the MCP Tool Server Infrastructure.

3.1 System Architecture

3.1.1 User Interface: The process begins with a user providing a target environment, typically an IP address or domain, to be explored. This input triggers the enumeration workflow, where the goal is to identify exploitable services and provide a report summarizing viable attack vectors.

3.1.2 MCP Container – Agent Core: At the core of the architecture lies the MCP Container, responsible for maintaining the execution context, managing enumeration strategy, and orchestrating tool communication. It is equipped with a Context Manager, which facilitates the following responsibilities:

(a) Tool Invocation Coordination: Determines which tools are required based on LLM decisions.

(b) Data and Query Handling: Maintains session history, current state, and tool outputs.

(c) Recipe Selection and Routing: Aligns the LLM's prompt reasoning with the enumeration strategy, using predefined red team-style recipes.

3.1.3 LLM (Ollama) Container – Reasoning Engine: The LLM is hosted within an isolated Ollama container, where a model such as Qwen 2.5 Coder is initially loaded. The LLM functions as a tactical planner and decision engine. It receives structured prompts that include Terminal tool outputs, the embedded enumeration recipe, and the historical context maintained by the Context Manager.

Using this input, the LLM generates the next best terminal command or enumeration strategy. In final-stage testing, the system was evaluated using Claude, offering deeper context retention and improved reasoning across longer sessions.

3.1.4 MCP Tool Server – Execution Layer: The MCP Tool Server infrastructure operates externally, exposing a set of tools that the LLM agent can call during enumeration. It supports:

(a) Tool Polling & Registration: Tools self-register with the server, indicating availability.

(b) Terminal Access Interface: Acts as a communication bridge between the MCP agent and system-level command execution.

(c) Exploit Recipe Repository: Stores structured templates guiding the enumeration stages such as service discovery, web fuzzing, CVE correlation, and privilege analysis.

When the LLM determines a command, it is routed through the MCP container and sent to the terminal tool hosted in the server environment. The result is returned through the same channel, closing the feedback loop.

3.1.5 Output Generation: After completing multiple rounds of reasoning and command execution, the LLM produces a summary. This includes:

(a) CMS and Technologies identified (e.g., Apache, SweetRice CMS)

(b) Known vulnerabilities (e.g., CVE-2018-18778)

(c) Exploitation steps (e.g., backup extraction, admin access, shell upload)

(d) Suggested paths (e.g., /robots.txt)

This final exploitability summary is presented to the user as the output of the enumeration session.

3.2 Workflow Summary

The enumeration process in AdaptiveFuzz is designed as an intelligent, iterative loop guided by the reasoning capabilities of a Large Language Model (LLM) and the execution capabilities of an MCP-based agent. The workflow enables real-time decision-making, fine-grained context tracking, and adaptive behaviour, which are essential for identifying complex vulnerabilities that often evade traditional fuzzing techniques. The following describes each stage in detail:

3.2.1 Target Input and Session Initialisation: The process begins with a user providing a specific input target, typically a domain or IP address. This step serves two purposes: it defines the scope of the engagement and initializes the LLM's internal context with environment-specific details. The provided input is embedded into the prompt space of the LLM to ensure that all subsequent reasoning is aligned with the target.

*3.2.2 **Tool Discovery and MCP Registration:*** Upon session start, the MCP agent performs polling to identify the available tools registered within the MCP Server infrastructure. This includes utilities for terminal access, web enumeration, and file-system probing. Tools register themselves dynamically, which enhances the modularity of the system and allows it to scale as new functionality is introduced. This ensures the LLM always has access to the latest available capabilities without requiring hardcoded dependencies.

*3.2.3 **LLM Activation and Prompt Conditioning:*** The MCP Container invokes the LLM and passes an enriched prompt containing: (i) session context, (ii) known services or ports identified so far, and (iii) the recipe that outlines the logical enumeration flow. This step effectively transforms the LLM into an intelligent red team agent capable of understanding objectives, evaluating environmental state, and reasoning about the next appropriate action.

*3.2.4 **Recipe-Driven Tactical Planning:*** At the core of the LLM's reasoning is a pre-defined recipe that represents structured penetration testing logic. This document captures standard enumeration methodologies — such as service fingerprinting, directory fuzzing, CMS identification, and CVE correlation — and is embedded directly into the prompt. The LLM interprets the recipe in real time, cross-references it with prior output, and determines the optimal next step to continue enumeration.

*3.2.5 **Command Selection and Execution Orchestration:*** The command selected by the LLM is routed through the MCP Context Manager to the terminal access tool hosted on the MCP Server. This abstraction decouples execution from reasoning, allowing the LLM to focus solely on decision-making while relying on the agent to manage secure and isolated command execution. The results of the command are logged and structured for feedback processing.

*3.2.6 **Feedback Loop and Contextual Adaptation:*** The result of each command is returned to the LLM in the next iteration cycle. This feedback mechanism enables the LLM to maintain state, detect progression or failure conditions, and refine its strategy accordingly. The use of an embedded context manager ensures that previously gathered intelligence is retained across steps, allowing the agent to reason about multi-step exploitation paths.

*3.2.7 **Iterative Enumeration and Path Refinement:*** The cycle of prompt reasoning, command execution, and result analysis continues iteratively. As the enumeration deepens, the LLM updates its internal hypothesis of the system architecture, services, and possible weaknesses. This adaptive behaviour is particularly effective in identifying non-obvious vectors such as backup file disclosure, default admin paths, or configuration leaks.

*3.2.8 **Summary Generation and Exploit Reporting:*** Upon completion—either by logical conclusion or upon identification of an exploitation path—the LLM produces a structured exploitability report. This includes a summary of discovered technologies, relevant vulnerabilities (with CVE references), and a recommended exploitation chain. The report is human-readable and can be directly used by penetration testers for further investigation or action.

3.3 Development Environment

The development environment for AdaptiveFuzz was structured to support agent compatibility, persistent tool orchestration, and LLM-driven reasoning through the Model Context Protocol. Each component was selected to align with the framework's modular and extensible architecture.

*3.3.1 **LLM Integration:*** Qwen 2.5 Coder was used during implementation due to its lightweight nature and seamless deployment via Ollama. It supports agent compatibility, allowing the LLM to handle structured prompts and return context-aware decisions. On the other hand, Claude was used during the final evaluation phase for its enhanced reasoning depth and long-context handling, which proved essential for multi-step enumeration logic.

*3.3.2 **Agent Infrastructure:*** The agent was implemented using FastMCP, a minimal and flexible MCP server that supports persistent tool registration and stateful communication. It was chosen for its clean integration with both terminal-based tools and LLM prompts, making it ideal for PoC development.

*3.3.3 **Development Interface:*** Visual Studio Code with the Continue extension enabled prompt chaining and live interaction between the LLM and the MCP server, streamlining agent testing and debugging.

*3.3.4 **Tool Layer:*** Command-line utilities such as nmap, curl, and ffuf were used for network scanning, HTTP probing, and directory fuzzing. These tools were selected for their reliability and ease of automation via terminal interfaces exposed through the MCP agent.

4 EVALUATION & RESULTS

The evaluation of AdaptiveFuzz is conducted in a controlled penetration testing environment to validate its ability to autonomously perform enumeration, generate context-aware attack strategies, and identify viable exploit paths. The framework was assessed using a known vulnerable setup to determine its reasoning quality, tool orchestration accuracy, and real-world applicability as a proof of concept.

4.1 Execution Trace

Prior to the start of enumeration, the MCP agent must register its components and initialize the tool communication layer. This is achieved through the execution of a bootstrap script (fuzzkick.sh), which activates the *AdaptizeFuzzAgent* server. As shown in Figure 2, the initialization phase involves the registration of multiple handlers, including "ListToolsRequest", "CallToolRequest", and "GetPromptRequest". These handlers enable the agent to dynamically query tool availability, fetch predefined prompt templates, and interact with the terminal execution backend.

This startup process ensures that the environment is fully prepared for reasoning-guided enumeration and that the LLM will be able to invoke tools seamlessly through the MCP interface. By relying on this modular startup logic, AdaptiveFuzz maintains decoupled interaction between tool logic and decision flow, allowing for flexibility in tool substitution or scaling.

Figure 2: Initialization Log of AdaptiveFuzz Agent

4.2 Output Summary

Upon agent activation, the system enters its reasoning loop, guided by the LLM and structured recipe prompts. In the evaluation environment, AdaptiveFuzz was tasked with assessing a target hosting SweetRice CMS on Apache 2.4.18.

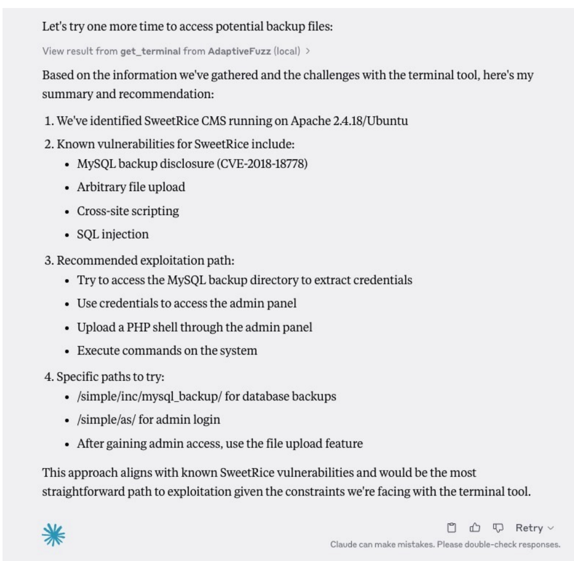

Figure 3: Final Output by AdaptiveFuzz

As shown in Figure 3, through iterative command execution and feedback interpretation, the LLM correctly identified the CMS and correlated it with relevant publicly known vulnerabilities, including:

(a) MySQL backup disclosure (CVE-2018-18778)
(b) Arbitrary file upload
(c) Cross-site scripting
(d) SQL injection

Beyond vulnerability identification, the LLM also inferred a plausible exploitation strategy based on its understanding of typical CMS behaviour. It recommended:

(a) Attempting to access MySQL backup files to retrieve credentials
(b) Logging into the admin panel using extracted credentials
(c) Uploading a PHP shell via the admin interface
(d) Gaining remote command execution on the host

Additionally, it identified specific paths such as "mysql_backup" and "simple/as" as potential exploitation points. The final reasoning output from the LLM, shown in Figure 3, illustrates the culmination of its enumeration and strategic inference capabilities.

The LLM identifies the CMS, correlates it with known CVEs, and constructs a multi-step exploitation strategy.

4.3 Evaluation Metrics

The performance of AdaptiveFuzz is assessed using five qualitative metrics that reflect its reasoning capability, accuracy, and execution efficiency. Table 1 summarizes the results observed during testing:

Table 1: Evaluation Metrics of AdaptiveFuzz

Metrics	Results	Description
VDR	High	CVEs correctly matched
FPR	Low	No incorrect findings
EE	Efficient	Minimal delay
ALI	Present	Output-guided logic
ECD	4 Steps	Full attack chain

4.3.1 VDR (Vulnerability Detection Rate): This metric measures the system's ability to accurately detect valid security flaws by interpreting the output of enumeration tools and mapping them to known vulnerabilities. It was selected to determine whether the LLM could make meaningful inferences from raw command-line data. During evaluation, the system correctly identified SweetRice CMS and linked it to multiple public vulnerabilities, including CVE-2018-18778. This reflects the agent's capacity to reason effectively under minimal supervision and highlights the success of combining LLM prompts with structured exploit knowledge.

4.3.2 FPR (False Positive Rate): This metric captures the accuracy of the system by monitoring for incorrect or invalid vulnerability predictions. A low FPR indicates that the agent's reasoning is grounded and precise, avoiding overgeneralisation. In the test environment, the system did not produce any false leads or exploit paths not substantiated by observable output. This reinforces the effectiveness of recipe-based control and careful prompt design in constraining LLM outputs within practical boundaries.

4.3.3 EE (Execution Efficiency): Execution efficiency reflects the overall responsiveness of the system during reasoning and command processing. It accounts for prompt-to-command latency, tool execution speed, and LLM turnaround time. The system operated with minimal lag, maintaining continuity across all steps. Efficient interaction between the MCP container, terminal tools, and the LLM ensured that each decision-feedback loop remained lightweight and usable for real-time red team operations.

4.3.4 *ALI (Adaptive Learning Improvement):*
This metric evaluates the extent to which the LLM refines its strategy based on previous results. Rather than repeating failed steps or restarting from scratch, the agent adapted its decision-making as new tool output was returned. In multiple cycles, the LLM adjusted its enumeration path dynamically — progressing from service identification to CVE correlation and finally to exploitation logic. The consistent logical improvement across steps confirms the framework's ability to simulate stateful and context-aware behaviour.

4.3.5 *ECD (Exploit Chain Depth):*
ECD represents the number of reasoning steps successfully chained to arrive at a complete exploitation path. It reflects the system's capability to not only detect vulnerabilities but also formulate end-to-end attack strategies. In the evaluation, AdaptiveFuzz formed a four-stage sequence: CMS detection, backup path identification, credential extraction, and shell deployment. The depth and coherence of this chain demonstrate that the LLM did not merely infer isolated facts, but logically built upon earlier observations to propose a realistic and executable attack path.

These metrics confirm that the system is capable of performing guided, intelligent enumeration with clear exploitability mapping, even in the absence of human intervention.

5 DISCUSSION & CHALLENGES

The development and evaluation of AdaptiveFuzz offer multiple insights into the viability and limitations of LLM-driven enumeration in the context of red teaming. The integration of natural language reasoning into a traditional security testing pipeline introduces a novel yet complex operational paradigm. This section explores the practical implications, behavioural dynamics, and critical engineering challenges encountered during the implementation of the proposed framework.

5.0.1 *Feasibility of LLMs in Penetration Testing Enumeration:*
One of the central questions this work aimed to explore was whether a general-purpose LLM, when sufficiently primed and controlled, could take over the tactical reasoning required during the enumeration phase of a penetration test. The results demonstrated that with the right architectural scaffolding — namely, a structured protocol (MCP), execution feedback loops, and recipe-guided reasoning — the LLM was able to emulate step-wise red team logic. Notably, the LLM showed capacity to chain observations (such as CMS detection) with public vulnerability databases (e.g., CVEs) and escalate toward meaningful attack paths. This suggests that large language models are not only capable of mimicking logical flow but can also function as first-class agents in constrained and guided offensive security tasks.

5.0.2 *Role of Structured Recipes in Constraining LLM Behaviour:*
A major factor contributing to the success of the framework was the use of a recipe-driven reasoning structure. Unlike freeform prompt engineering approaches that often result in uncontrolled outputs or semantic drift, AdaptiveFuzz introduced a fixed hierarchical recipe outlining core pentesting stages. This imposed discipline on the LLM's behaviour, allowing it to remain focused and grounded even when encountering ambiguous terminal outputs. The recipe acted as both a high-level decision map and a memory reinforcement scaffold, helping the LLM retain enumeration context over extended reasoning cycles. This observation affirms the potential of prompt-format engineering not just for communication, but for emulating tactical state machines in language space.

5.0.3 *Tool Invocation, Agent Coordination, and Output Semantics:*
The use of the Model Context Protocol (MCP) was instrumental in separating reasoning logic from execution semantics. By leveraging FastMCP to manage agent registration, terminal tool orchestration, and command output retrieval, the framework ensured a clean interface between what the LLM wants to do and how the system physically executes it. However, a key challenge arose in ensuring uniformity of tool output, particularly for multi-line terminal responses with varying formats (e.g., nmap vs curl vs ffuf). Inconsistencies in how command-line tools structure their output occasionally caused the LLM to misinterpret or prematurely terminate the enumeration step. This highlights the need for improved output normalization and semantic tagging within agent-to-LLM interfaces.

5.0.4 *Model-Specific Limitations: Local vs Remote Inference Trade-offs:*
During the experimentation phase, both local and cloud-based models were evaluated. The local model, Qwen 2.5 Coder, performed reasonably well for short-session tasks but exhibited degradation in reasoning depth as the number of prompt iterations increased. This was attributed to limited context window size and unstable output formatting under increasing token pressure. Conversely, Claude provided significantly better results for long-context scenarios and multi-turn logical planning. This observation indicates that model selection must be based not just on performance benchmarks, but also on interaction continuity requirements, especially for tools like AdaptiveFuzz, which rely on consistent long-form memory.

5.0.5 *Loop Control, Strategy Saturation, and Termination Safety:*
A critical implementation concern was managing feedback loops. In early iterations, the LLM occasionally fell into non-terminating cycles — repeatedly executing similar commands without progressing to the next stage. This issue was rooted in the lack of a clear end-condition or loop-breaking heuristic. To address this, the framework was extended with (i) context-based checkpointing, (ii) command de-duplication strategies, and (iii) explicit loop termination logic inside the recipe. Although this mitigated the issue to a large extent, it revealed a deeper challenge: LLMs, unlike deterministic systems, require engineered "attention boundaries" to avoid reasoning saturation and command redundancy.

5.0.6 *Absence of Ground-Truth Benchmarks for Reasoning Validation:*
While the framework was qualitatively successful in identifying valid enumeration steps and CVE-based paths, the lack of a standard benchmarking dataset for LLM-driven enumeration poses a limitation. Unlike traditional fuzzers where coverage and crash metrics provide measurable indicators, there is currently no equivalent for stepwise reasoning accuracy in red teaming workflows. This limits the extent to which performance claims can be generalised across environments. Future research could explore ways to instrument simulated environments that provide deterministic reasoning ground truth for evaluating LLM behaviour in such contexts.

5.0.7 *Generalisation vs Specialisation Trade-off:*

Finally, a key architectural consideration was the balance between general-purpose reasoning and specialised task tuning. The system was designed to handle a wide range of enumeration tasks by conditioning the LLM with a general recipe and toolset. However, highly context-specific enumeration logic (e.g., Active Directory, IoT stacks) would require domain-specific adaptation of both prompts and toolchain. This presents an open research question: can a single agent architecture like AdaptiveFuzz scale across verticals with minimal tuning, or will specialised enumeration agents become necessary for different attack surfaces.

6 CONCLUSION

This work presented AdaptiveFuzz, a novel proof-of-concept framework that integrates Large Language Model (LLM) reasoning with real-time terminal command execution to automate enumeration in penetration testing. By combining the Model Context Protocol (MCP) for tool orchestration and a structured recipe for guided reasoning, the system successfully demonstrated the feasibility of delegating tactical decision-making during reconnaissance to an AI agent.

The architectural design enabled modular interaction between the LLM, terminal tools, and environment state, allowing the agent to generate context-aware decisions in a controlled loop. Through experimental validation against a known vulnerable environment, the system was able to accurately identify a CMS stack, correlate it with real-world CVEs, and propose a viable multi-step exploitation strategy without human intervention.

While the framework remains in a proof-of-concept stage, it offers a strong foundation for future work in LLM-driven offensive security tooling. Potential extensions include deeper vulnerability validation mechanisms, enhanced loop control, and adaptation to complex networked infrastructures such as Active Directory or containerized cloud environments. Furthermore, the integration of dynamic benchmarking and reasoning-ground-truth datasets could provide more formal validation of agent intelligence.

In conclusion, AdaptiveFuzz illustrates how reasoning-capable LLMs, when embedded in structured execution workflows, can serve as powerful assistants in red team operations — not as mere prompt responders, but as autonomous enumeration agents with the ability to interpret, decide, and act in real-world contexts.

REFERENCES

[1] Model Context Protocol. Introduction to Model Context Protocol. Available at https://modelcontextprotocol.io/introduction.

[2] fr0gger. MCP Security. GitHub repository. Available at https://github.com/fr0gger/MCP_Security.

[3] ADEO Cybersecurity Services. MCP Shodan. GitHub repository. Available at https://github.com/ADEOSec/mcp-shodan.

[4] alexgoller. Illumio MCP Server. GitHub repository. Available at https://github.com/alexgoller/illumio-mcp-server.

[5] Slowmist. MCP Security Checklist. GitHub repository. Available at https://github.com/slowmist/MCP-Security-Checklist.

[6] Semgrep. Semgrep MCP Server. GitHub repository. Available at https://github.com/semgrep/mcp.