Università del Piemonte Orientale

Dipartimento di Scienze e Innovazione Tecnologica

Corso di Laurea Magistrale in Intelligenza Artificiale e Innovazione Digitale

Tesi di Laurea Magistrale

THESEUS: Design, Development, and Evaluation of an Agentic System for the Discovery and Execution of Exploit Chains for Penetration Testing

Tutore interno:
Prof. Cosimo Anglano

Candidato:
Leonardo Galliera

Firmato digitalmente da Cosimo Antonio Anglano
Data: 29.09.2026 10:54:05 CEST
Organizzazione: UNIVERSITA' DEGLI STUDI DEL PIEMONTE ORIENTALE/01943490027

Anno Accademico 2025/2026

Abstract

Autonomous penetration testing with Large Language Models (LLMs) offers the potential to automate complex security assessments, but deploying frontier closed-source models raises confidentiality and data-isolation concerns, while fully local open-weight models face challenges in long-horizon reasoning, context management, and reliable tool use. These limitations become particularly pronounced in multi-stage attacks, where reconnaissance, exploitation, post-exploitation, privilege escalation, and lateral movement must be coordinated as a continuous campaign.

In this work, we present **THESEUS**, an autonomous multi-agent offensive security framework designed to perform end-to-end network penetration testing using a locally deployed LLM. THESEUS addresses the challenges of long-horizon attack execution through three complementary design principles. First, **role-decoupled specialization** decomposes the penetration-testing workflow among task-specific agents, namely the *Orchestrator*, *Scout*, *Coder*, and *PostExploiter*. Second, an external **Attack Knowledge Graph (AKG)** maintains persistent campaign state, including discovered hosts, credentials, sessions, network relationships, and pivoting information, allowing individual agents to remain stateless while preserving the global attack context. Third, **deterministic grounding and closed-loop verification** constrain model decisions through concrete security tools and execution feedback, reducing reliance on unsupported exploit assumptions.

THESEUS is evaluated on established penetration-testing benchmarks as well as on multi-stage and multi-host scenarios designed to assess attack-chain execution, examining both general exploitation capability and the ability to preserve and exploit state across successive attack stages. The results show that a locally deployed LLM, when equipped with specialized agents, externalized state, and deterministic tool grounding, can execute complex penetration-testing workflows and maintain coherent multi-hop attack chains, while also revealing the conditions under which such autonomous execution remains fragile. This demonstrates the potential of structured agentic architectures as a practical alternative to monolithic LLM-based penetration testing, and points to persistent state and deterministic grounding as key ingredients for extending autonomous offensive-security agents to more realistic, long-horizon engagements.

Contents

1 Introduction **1**
- 1.1 Context and Background . . . 1
- 1.2 Problem Statement . . . 1
- 1.3 Emergence of Autonomous LLM Agents in Cybersecurity . . . 2
- 1.4 Thesis Objectives and Research Questions . . . 2
- 1.5 Summary of Main Technical Contributions . . . 3
- 1.6 Thesis Structure . . . 3

2 State of the Art and Technical Background **4**
- 2.1 Technical Background . . . 4
  - 2.1.1 Autonomous LLM Agents and State Management . . . 4
  - 2.1.2 Attack Graphs and Labeled Property Graphs . . . 5
  - 2.1.3 Local LLM Inference and Context Constraints . . . 7
  - 2.1.4 Penetration-Testing Frameworks and Operational State . . . 7
- 2.2 State of the Art in LLM-Based Penetration Testing . . . 9
  - 2.2.1 Vulnerability Analysis and Autonomous Exploitation . . . 10
  - 2.2.2 Autonomous Penetration-Testing Agents . . . 10
  - 2.2.3 Stateful and Long-Horizon Penetration Testing . . . 12
  - 2.2.4 Multi-Host Attack Chains . . . 12
- 2.3 Evaluation Methodologies and Benchmarks . . . 13
  - 2.3.1 Benchmarks and Evaluation Environments . . . 14
  - 2.3.2 The Operational-Realism Gap . . . 14
  - 2.3.3 Quantitative Comparison . . . 15
  - 2.3.4 Limits of Quantitative Comparability . . . 19
- 2.4 Research Gap and Positioning of THESEUS . . . 19
  - 2.4.1 Persistent State Under Context Constraints . . . 20
  - 2.4.2 Reliable Technical Decision-Making . . . 20
  - 2.4.3 Causal Multi-Host Attack Progression . . . 20
  - 2.4.4 Positioning of THESEUS . . . 21

3 THESEUS Framework System Architecture & Detailed Design **22**
- 3.1 System Overview & Core Architectural Principles . . . 22
  - 3.1.1 Closed-Loop Control Architecture . . . 23
  - 3.1.2 Operational Lifecycle . . . 24
- 3.2 Asset-Oriented Sequential Attack Knowledge Graph (AKG) . . . 26
  - 3.2.1 Formal Graph Definition & Schema . . . 26
  - 3.2.2 State Offloading Strategy & Context Isolation . . . 28

3.2.3 The Campaign Abstraction & State Persistence Architecture . . 30
3.2.4 Topological Hop Depth: Formal Definition 31
3.3 Multi-Agent Team Architecture & Least Privilege Enforcement 32
3.3.1 Least Privilege Verification & Guardrail Enforcement 37
3.3.2 Indirect Prompt Injection Resilience 38
3.4 The Deterministic Payload Selection Engine 38
3.4.1 A Two-Tier Selection Strategy: Empirical Fast-Path and Deterministic Fallback 38
3.4.2 Lexicographical Multi-Criteria Payload Ranking 39
3.4.3 Bimodal Payload Partitioning and Inherent Network Uncertainty 40
3.4.4 Agentic Feedback Loop and Failure Recovery Strategy 42
3.5 Multi-Hop Pivoting Mechanics & Network Tunnelling 42
3.5.1 Initial Compromise & Pivot Discovery 43
3.5.2 Internal Network Tunnelling & Recursive Pivoting 43
3.5.3 Topological Session & Route Restoration 44
3.5.4 Depth-Independent Pivoting Design 45
3.6 Real-Time Visual Telemetry Interactive Dashboard 45
3.6.1 Dashboard Architecture Communication Pipeline 45
3.7 End-to-End System Operational Flow & Execution Walkthrough 46

4 Experimental Evaluation and Results 51
4.1 Research Questions and Common Experimental Protocol 51
4.1.1 Research Questions 51
4.1.2 Common Experimental Protocol 52
4.1.3 Metrics 53
4.2 AutoPenBench 54
4.2.1 Results 54
4.2.2 Operational Efficiency 55
4.2.3 Comparison with Existing Systems 56
4.2.4 Exploratory Evaluation on a Smaller Model 58
4.3 Chain-Oriented Evaluation 58
4.3.1 Three-Stage Multi-Hop Attack Chain 58
4.3.2 Two-Stage Credential-Reuse Chain 59
4.3.3 Privilege-Escalation-Assisted Lateral Movement 60
4.3.4 Adaptive Target Selection 61
4.3.5 Cross-Branch Credential Dependency and Backtracking 62
4.3.6 Multi-Source Fragmented Credential Assembly (Failure Case) . 64
4.3.7 Quantitative Summary 64
4.3.8 Discussion 66
4.4 Experimental Validation of the Proposed Mechanisms 67
4.4.1 Deterministic Payload Selection Engine 67
4.4.2 AKG-Based Context Management 69
4.4.3 State Restoration and Reproducibility 72
4.5 Evaluation Limitations 73
4.6 Evaluation Summary 74

5 Conclusion and Future Work **76**
5.1 Conclusion 76
5.2 Future Work 77
5.2.1 GOAD Support 77
5.2.2 Improved Host and Target Selection 77
5.2.3 AKG-to-Shell Attack Reproduction 78
5.2.4 Enhanced Payload Determinism for Deep Pivot Chains 78

Bibliography **82**

Acknowledgements **83**

Chapter 1

Introduction

1.1 Context and Background

Modern enterprise infrastructures are increasingly composed of heterogeneous and interconnected systems, including cloud services, containerized applications, internal networks, and distributed services. As a result, security assessments can no longer be limited to the identification of isolated vulnerabilities on individual hosts. Realistic attack scenarios often involve multiple stages, where information and access obtained from one system enable subsequent actions against previously unreachable targets.

This growing complexity has increased the need for automated approaches capable of reasoning across long and dynamic attack sequences. While traditional penetration testing remains highly dependent on human expertise, recent advances in Large Language Models (LLMs) have created new opportunities for automating parts of the security assessment process. In particular, LLM-based agents can interpret heterogeneous information, select actions dynamically, and adapt their strategy according to the results obtained during an assessment.

However, transforming these capabilities into reliable autonomous penetration testing systems remains challenging, particularly when attacks require maintaining state across multiple hosts and network segments.

1.2 Problem Statement

Despite the critical need for continuous security validation, existing offensive assessment approaches present severe limitations:

- **Manual Penetration Testing Constraints:** Skilled security professionals provide high-fidelity assessments but are expensive, resource-intensive, and unscalable. As a result, penetration tests are conducted infrequently (e.g., annually), creating large temporal windows of risk.
- **Static Scanner Limitations:** Automated scanners (such as Nessus [27] or OpenVAS [2]) identify vulnerabilities based on static signatures but lack the ability to validate exploitability, execute post-exploitation commands, or pivot across multi-subnet topologies.

1.3 Emergence of Autonomous LLM Agents in Cybersecurity

The integration of Large Language Models (LLMs) into cybersecurity has enabled autonomous agents capable of high-level reasoning and command generation. However, applying raw LLM architectures to offensive operations introduces major failure modes:

1. **LLM Non-Determinism and Hallucinations:** LLMs may generate invalid actions, incorrect commands, or technically incompatible exploitation strategies. These errors can lead to repeated execution attempts, inefficient reasoning loops, and ultimately unsuccessful attack progression.

2. **Context Window Limits and Token Explosion:** Standard agent implementations often append raw command outputs, network scan results, and service responses directly to conversation histories. On local inference setups with strict token bounds, this causes rapid context saturation and degraded reasoning.

3. **Lack of Multi-Hop Pivoting Mechanics:** Existing agent frameworks often focus primarily on single-host perimeter targets and fail to maintain the dynamic network state, routing information, and compromised-host knowledge required for systematic multi-hop pivoting into internal subnets.

1.4 Thesis Objectives and Research Questions

This thesis investigates the design and implementation of **THESEUS**, an autonomous multi-agent framework designed to perform end-to-end, multi-hop penetration testing under local inference constraints. THESEUS integrates Large Language Model (LLM)-based agents with established offensive security tools, including *Metasploit*, to autonomously execute and coordinate complex attack workflows across multiple hosts and network segments.

The research is guided by three central questions:

1. **RQ1:** Can the framework autonomously execute penetration-testing tasks from reconnaissance to exploitation?

2. **RQ2:** How does the framework perform compared with existing LLM-based penetration-testing systems on a common benchmark?

3. **RQ3:** Can the explicit representation of attack state be successfully used to execute dependent attack stages?

4. **RQ4:** Can the framework maintain and exploit information obtained from previous stages when constructing longer, and non-linear, attack chains?

5. **RQ5:** Do the individual architectural mechanisms introduced by the framework (deterministic payload selection, AKG-based context management, and state restoration) produce the effects they are designed to produce?

1.5 Summary of Main Technical Contributions

The primary technical contributions of this thesis are:

1. **Stateless Worker Multi-Agent Architecture:** Design and implementation of an Agno-based multi-agent architecture composed of a stateful *Orchestrator* and specialized stateless worker agents, including the *Scout*, *Coder*, and *PostExploiter*. The architecture separates strategic coordination from specialized operational tasks and restricts tool access according to agent responsibilities.

2. **Asset-Oriented Sequential Attack Knowledge Graph (AKG):** A dynamic graph-based memory structure that externalizes attack state from the LLM context and represents discovered assets, compromised hosts, credentials, network relationships, and pivoting information as structured entities and relationships.

3. **Deterministic 5-Axis Payload Selection Engine:** A programmatic scoring mechanism that evaluates compatible *Metasploit* payloads across five dimensions, replacing LLM-based payload selection with a deterministic decision process.

4. **Automated Multi-Hop Pivoting Mechanics:** Native integration with *Metasploit* routing and session management mechanisms, enabling the framework to maintain network reachability and progressively extend an attack chain from perimeter systems to previously inaccessible internal networks.

1.6 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2:** Reviews literature on attack graph modeling, autonomous LLM agents in security, and local inference constraints.

- **Chapter 3:** Details the system design of THESEUS, including agent specialization, AKG schema, and the payload selection engine.

- **Chapter 4:** Presents the experimental evaluation, hardware setup, and empirical results across multi-subnet testbeds.

- **Chapter 5:** Summarizes technical achievements, discusses system limitations, and outlines future research directions.

Chapter 2

State of the Art and Technical Background

The development of autonomous penetration-testing systems based on Large Language Models (LLMs) has progressively moved from vulnerability analysis and exploit generation toward agents capable of interacting directly with target environments. Modern systems can invoke security tools, interpret their outputs, maintain information about the current engagement, and adapt their actions according to observations obtained during execution.

This evolution introduces requirements that are substantially different from those of conventional vulnerability assessment. In an autonomous penetration test, reconnaissance, exploitation, post-exploitation, privilege escalation, credential collection, and lateral movement are not independent operations. Instead, the outcome of an earlier action can determine which actions become possible later in the engagement. In particular, multi-host attack chains require the system to preserve information about compromised assets, credentials, network connectivity, privileges, and previously discovered resources over potentially long execution horizons.

This chapter first introduces the technical concepts required to understand these challenges and the architecture of THESEUS. It then reviews the state of the art in LLM-based autonomous penetration testing, followed by the principal benchmarks and evaluation methodologies used in the literature. Finally, the chapter identifies the remaining research gap and positions THESEUS with respect to existing approaches.

2.1 Technical Background

2.1.1 Autonomous LLM Agents and State Management

LLM-based penetration-testing agents can be viewed as closed-loop systems that repeatedly alternate between reasoning and interaction with an external environment. Rather than producing a complete attack procedure before execution, the agent observes the current state, selects an action, invokes an external tool, interprets the resulting observation, and uses the new information to determine the next action.

A simplified interaction loop can be represented as in Figure 2.1.

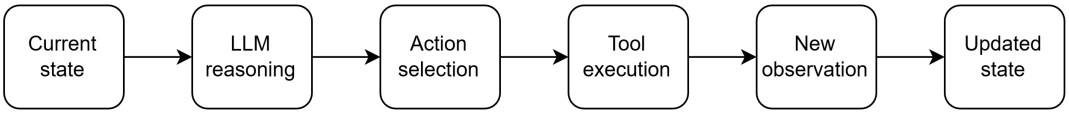

Figure 2.1: Simplified closed-loop interaction model of an autonomous LLM-based agent, from the current state through reasoning, action selection, tool execution, and observation to the updated state.

This interaction model is particularly important in penetration testing because observations obtained during execution may modify the set of subsequent actions. For example, reconnaissance may reveal a vulnerable service, exploitation may establish a session, post-exploitation may reveal a new network interface, and the newly discovered connectivity may make additional hosts reachable.

A fundamental architectural problem therefore concerns the representation of state. If all observations are retained directly in the LLM conversation, long execution sequences can produce rapidly growing contexts containing large quantities of command output, intermediate reasoning, and redundant information. This increases inference cost and can degrade the model's ability to effectively retrieve and use operationally relevant information from long contexts, as observed in the "Lost in the Middle" phenomenon [19].

State can consequently be externalized from the language-model context. Structured representations, summaries, task trees, databases, knowledge bases, and other forms of persistent memory allow an agent to retain information that must remain available across multiple reasoning steps without continuously appending the complete execution history to the prompt.

This distinction between *reasoning context* and *persistent engagement state* is central to long-horizon autonomous penetration testing. The former contains the information required by the model to make an immediate decision, whereas the latter represents information that remains relevant throughout the campaign.

2.1.2 Attack Graphs and Labeled Property Graphs

Graph-based representations have long been used to model relationships between vulnerabilities, system states, privileges, and possible attack paths. As surveyed by Wachter [30], security graph formalisms range from static Attack Trees and Attack Graphs to state enumeration approaches and Labeled Property Graphs (LPGs). Wachter's analysis of a broad corpus of existing formalisms proposes a taxonomy based on what their nodes represent: distinguishing, for instance, models centered on system assets from those centered on vulnerabilities or on logical conditions as well as on the agents involved and on whether the formalism supports automatic generation from data. This diversity reflects the lack of a single, agreed-upon notion of "attack graph" in the literature, and situates the choice of an LPG-based representation as one point among several possible design options.

Classical Attack Trees and Attack Graphs illustrate two ends of this spectrum. An Attack Tree is rooted at a single attacker goal, refined hierarchically through

AND/OR operators down to atomic actions, and is acyclic by construction. An Attack Graph is instead defined over a set of system states connected by transitions, may contain multiple goal states, and can in principle admit cycles, e.g., when an attacker repeatedly retries an unsuccessful action. Both formalisms, however, commit early to a single node semantics (a sub-goal, or a state), which makes them less convenient for representing the variety of concrete entities that a penetration-testing engagement actually produces.

An LPG represents entities as typed nodes associated with properties and relationships as typed directed edges. This representation is particularly suitable for penetration testing because an engagement naturally contains heterogeneous entities such as hosts, interfaces, services, credentials, sessions, privileges, and network segments, each with its own attributes and relations more rigid formalisms do not easily accommodate. For example, a simplified penetration-testing state can be represented as shown in Figure 2.2.

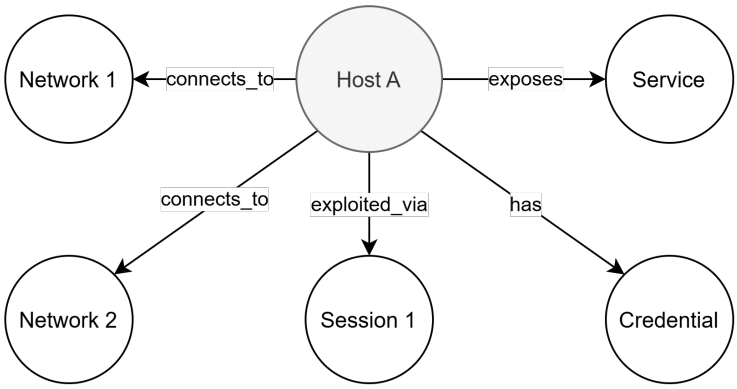

Figure 2.2: Simplified property graph representing a penetration-testing state through typed entities and directed relationships.

The graph can therefore represent not only the assets discovered during reconnaissance but also the dependencies that determine how the attack can progress.

Konsta et al. [15] survey existing techniques for the automatic generation of attack graphs and attack trees, classifying them into approaches driven by a system model, by a security analysis process, or by libraries of known vulnerability patterns. Across all three categories, most proposed methods are designed to produce a single attack model from a fixed input, and only rarely support updating that model as the underlying system changes – a limitation the authors explicitly flag as an open challenge. This distinction is particularly relevant to autonomous penetration testing. During an active engagement, the graph cannot be considered a fixed description of the environment: reconnaissance may reveal new services, successful exploitation may create new access paths, and post-exploitation may expose additional interfaces, credentials, or network segments. Consequently, an operational attack graph must support incremental updates as new evidence becomes available. This provides a natural conceptual foundation for the Asset-Oriented Sequential Attack Knowledge Graph (AKG) introduced in Chapter 3.

2.1.3 Local LLM Inference and Context Constraints

The deployment environment of an autonomous agent directly affects its architecture. Cloud-based LLM APIs provide access to increasingly large models and context windows, but they require the transmission of potentially sensitive operational information to an external service. Such a dependency can be undesirable in enterprise environments, controlled laboratories, or air-gapped penetration-testing infrastructures.

Local inference provides an alternative in which model execution remains inside the controlled environment. Modern inference engines such as vLLM [16] and llama.cpp [9] provide different mechanisms for efficient local LLM inference. vLLM introduces PagedAttention for efficient KV-cache management, while llama.cpp provides lightweight local inference with configurable KV-cache and memory-management mechanisms.

For autonomous penetration testing, however, local deployment also introduces practical constraints. The available computational resources may limit model size, inference throughput, and effective context length. When the context window is relatively small, continuously appending raw terminal output becomes particularly expensive.

These constraints motivate an architectural separation between the LLM and the persistent state of the engagement. Rather than requiring the model to retain the complete execution history, the system can maintain a compact structured representation of relevant information and provide the model only with the state required for the current decision.

This design principle is particularly important for long multi-stage attack chains, where the amount of information generated by the environment can grow much faster than the information actually required for each individual reasoning step.

2.1.4 Penetration-Testing Frameworks and Operational State

A penetration-testing framework provides the execution layer through which security operations can be performed against target systems. In this thesis, the framework used to operationalize exploitation and post-exploitation is Metasploit [26].

The relevance of such a framework to THESEUS is not limited to individual exploit execution. It provides a persistent operational environment in which successful exploitation can create sessions, post-exploitation can reveal additional information, and compromised hosts can provide access to otherwise unreachable network segments.

Modules and Execution

Metasploit organizes its functionality into reusable modules. Among these, exploit modules perform vulnerability exploitation, payloads define the functionality delivered after successful exploitation, and auxiliary modules support operations such as reconnaissance, enumeration, and interaction with network services.

The framework can be accessed interactively through `msfconsole` or programmatically through interfaces that allow external software to configure and execute operations. This programmatic access is particularly relevant to autonomous agents because the LLM does not need to implement exploitation functionality itself. Instead,

the agent can reason about the operation while the penetration-testing framework provides the underlying execution mechanisms.

Sessions as Operational State

A successful exploitation may establish an active session on the target. Unlike static information such as a discovered service or credential, a session represents an active communication channel through which subsequent operations can be performed.

The distinction between persistent engagement information and active sessions is important for autonomous penetration testing. Information about a host, such as its operating system, services, credentials, or interfaces, describes the environment. A session, in contrast, represents currently available operational access.

The simplified lifecycle is illustrated in Figure 2.3.

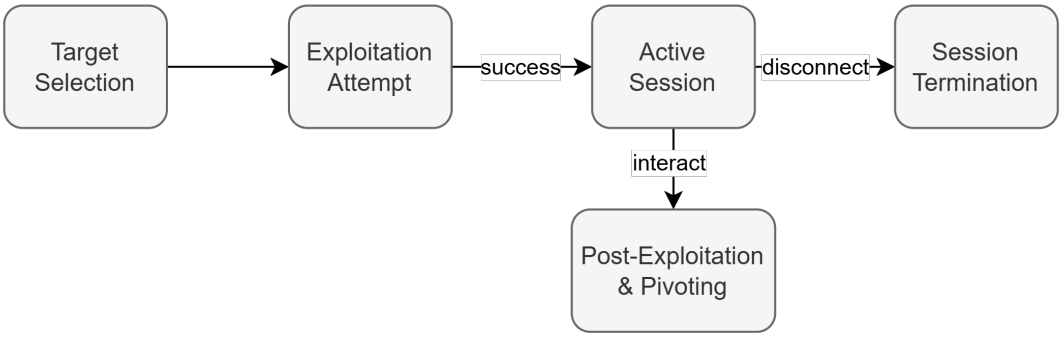

Figure 2.3: Simplified lifecycle of an active session, from target selection and exploitation to post-exploitation or session termination.

The resulting session can therefore become the starting point of subsequent operations rather than merely representing the final outcome of exploitation.

Post-Exploitation and State Expansion

Post-exploitation extends the engagement beyond the initial compromise. An active session can be used to collect information about the compromised system and its surrounding environment, including system properties, privileges, network interfaces, reachable networks, and credentials.

From the perspective of an autonomous agent, these operations are important because they can change the effective attack surface. A credential recovered from one host may enable authentication against another host, while a newly discovered interface may reveal a network segment that was not accessible from the original attacker position.

The compromise of a host should therefore be understood as a state transition, as shown in Figure 2.4.

This perspective is fundamental to the multi-host attack chains investigated later in this thesis.

Figure 2.4: Attack surface expansion flow

Routing and Multi-Hop Pivoting

A compromised host may possess network connectivity that is unavailable from the original attacking environment. For example, a host exposed to an external network may simultaneously possess an interface connected to an internal subnet.

In such a situation, the compromised host can act as an intermediate access point. A route associated with an active session allows traffic directed toward a previously unreachable network to traverse the compromised host, enabling what is commonly referred to as *pivoting*.

The resulting attack progression can be represented as shown in figure 2.5.

Figure 2.5: Conceptual progression of a multi-hop attack, illustrating the recursive nature of internal network discovery and pivot establishment.

Multi-hop penetration testing consequently requires the agent to preserve the relationship between compromised hosts, active sessions, network segments, and routes. A session is therefore not simply evidence that an exploit succeeded: it is an operational resource that can enable subsequent stages of the campaign.

2.2 State of the Art in LLM-Based Penetration Testing

Research on LLM-based penetration testing has evolved from vulnerability analysis and exploit generation toward autonomous agents capable of performing complete, iterative security workflows. The systems reviewed in this section can be distinguished according to three progressively more demanding capabilities:

1. **Multi-step exploitation**: the system performs several causally dependent actions against a target, where the outcome of one action influences subsequent actions.

2. **Stateful penetration testing**: the system preserves and reuses information across multiple phases of an engagement instead of relying exclusively on the immediate LLM context.

3. **Multi-host attack chains**: information or access obtained from one compromised host enables subsequent actions against previously inaccessible hosts or network segments.

This progression provides a useful framework for interpreting the systems discussed below.

2.2.1 Vulnerability Analysis and Autonomous Exploitation

Early LLM-based security research primarily investigated vulnerability analysis, exploit generation, and the use of publicly available vulnerability knowledge. Although these approaches demonstrate that language models can reason about technical weaknesses, they do not necessarily provide the closed-loop interaction required for autonomous penetration testing.

Fang et al. [8], for example, demonstrate that an LLM agent can autonomously identify and exploit web application vulnerabilities through iterative interaction with security tools. The resulting workflows include multiple dependent actions rather than a single generated exploit.

Related work investigates autonomous exploitation of previously disclosed vulnerabilities when the relevant technical information is available to the model [7]. LLM-CVX [6] provides a complementary evaluation perspective by testing LLMs against real vulnerabilities through executable security workflows.

These systems establish the feasibility of moving from vulnerability knowledge toward executable interaction. However, their primary evaluation unit remains the individual vulnerability or target. They provide limited evidence regarding persistent campaigns in which information obtained from an earlier compromise is required to complete later stages.

2.2.2 Autonomous Penetration-Testing Agents

A subsequent research direction treats penetration testing as an iterative agentic workflow rather than as isolated exploit generation. The agent repeatedly observes the environment, selects an action, invokes a security tool, and incorporates the result into subsequent decisions.

PentestGPT

PentestGPT [5] is one of the earliest frameworks specifically designed around LLM-driven penetration testing. Its modular architecture separates reasoning, generation, and parsing and introduces task-tree representations to mitigate the difficulty of maintaining a coherent penetration-testing context as tool outputs accumulate.

The framework therefore represents an important transition from single-action generation toward iterative penetration testing. The agent can interpret execution results, update its task representation, and select subsequent actions.

However, its state representation is primarily centered on task progression and context management. It does not provide an explicit persistent representation of cross-host relationships involving credentials, routes, sessions, and network segments.

AutoPT

AutoPT [31] addresses the problem of maintaining coherent execution state over extended interactions through a Penetration Testing State Machine.

By representing the current testing stage outside the unconstrained language model reasoning process, AutoPT makes workflow state an explicit architectural component. This reduces the dependence on the LLM context window for retaining basic information about the current phase.

The evaluation, however, remains centered on web penetration-testing scenarios. Consequently, its state representation primarily models workflow progression rather than evolving relationships between multiple hosts, network segments, credentials, and access paths.

Structured and Multi-Agent Approaches

Several subsequent systems investigate agent specialization, structured reasoning, external knowledge, and reusable execution experience. PentestAgent [29], VulnBot [14], and xOffense [21] employ specialized agents or orchestration mechanisms to decompose reconnaissance, vulnerability analysis, and exploitation into distinct responsibilities.

Other approaches focus on the reliability of long execution sequences. CurriculumPT [32] introduces an Experience Knowledge Base that stores successful and failed execution experiences for later reuse. RedTeamLLM [3] addresses memory management, context limitations, and plan correction. Structured Task Tree (STT) [25] constrains the action space using an explicit task representation derived from the MITRE ATT&CK framework.

These systems demonstrate a common architectural trend: information and decision processes that would otherwise reside entirely inside the LLM are progressively externalized into specialized agents, workflows, knowledge bases, memory systems, or structured reasoning representations.

Nevertheless, many earlier evaluations remain centered on individual hosts, isolated vulnerable services, or bounded scenarios. Consequently, they provide limited evidence regarding persistent attack chains in which the compromise of one asset dynamically changes the set of reachable targets.

APT-Agent

APT-Agent [17] explicitly addresses two failure modes that become increasingly important during long execution sequences: hallucination of technical entities and loss of operational context.

The framework introduces a hybrid rectification mechanism for recovering from incorrect commands and technical identifiers, together with command-specific memory that preserves relevant execution context across stages.

APT-Agent therefore represents a significant step toward reliable stateful penetration testing. However, its evaluation is performed on multiple vulnerable services within a single target environment and does not establish causal attack chains spanning multiple hosts or network segments.

2.2.3 Stateful and Long-Horizon Penetration Testing

As autonomous agents become capable of executing longer sequences, state management becomes increasingly important. The central problem is no longer only whether an agent can produce a technically valid action, but whether it can maintain a coherent representation of the engagement throughout a long sequence of dependent decisions.

This motivates architectures that separate short-term reasoning from persistent operational information. External memory, state machines, structured task trees, execution summaries, and persistent knowledge bases all represent different solutions to this problem.

The common limitation is that statefulness alone does not imply multi-host reasoning. An agent can maintain detailed state about a single target while remaining unable to reason about how a compromise changes the reachable network.

The distinction becomes particularly important when the attack progresses through credentials, network routes, privileges, or sessions. In such settings, the state is not simply a record of what has already happened. It also determines which actions are feasible next.

2.2.4 Multi-Host Attack Chains

Multi-host attack chains introduce an additional level of complexity. A successful compromise does not necessarily terminate a task; instead, it changes the environment in which subsequent actions are performed.

For example, a compromised host can provide:

- credentials reusable against other systems;
- additional privileges;
- network interfaces connected to previously inaccessible segments;
- routing opportunities for pivoting;
- information about additional hosts and services.

An autonomous system must therefore reason not only about the current target but about the evolving attack surface produced by previous compromises.

PenExpert

PenExpert [18] explicitly targets multi-stage and multi-host penetration testing. Its architecture combines multiple LLM agents with a MITRE ATT&CK-derived capability matrix, retrieval-backed technical knowledge, an Attack Surface Dictionary, and structured state management.

The state representation records information relevant to subsequent actions, including host and network-segment status, credentials, routes, privileges, and failure trajectories. This allows lateral movement and pivoting to be treated as consequences of previous compromises rather than isolated tasks.

PenExpert evaluates individual vulnerabilities, single-host penetration testing, and a multi-host cross-subnet environment. It therefore represents one of the closest existing approaches to explicit causal dependencies between hosts.

Cochise

Cochise [13] provides a complementary approach to autonomous multi-host penetration testing. Rather than relying on a large number of specialized agents, it adopts a lightweight Planner–Executor architecture in which long-term state is maintained outside the LLM context while a ReAct-style executor interacts with the environment.

The framework is evaluated on the Game of Active Directory (GOAD), a vulnerable enterprise environment in which the agent is not provided with a complete network description and must progressively discover the environment through execution.

Cochise provides evidence that persistent multi-host attack progression can be implemented without a highly complex multi-agent architecture. Explicit separation between LLM reasoning and persistent environment state can itself provide a mechanism for supporting long-horizon attack sequences.

PentestGPT V2

PentestGPT V2 [4] extends the original architecture toward long-horizon autonomous penetration testing. The framework introduces a Tool and Skill Layer, retrieval-augmented security knowledge, Task Difficulty Assessment (TDA), persistent state, and Evidence-Guided Attack Tree Search (EGATS).

The architecture explicitly addresses unreliable tool use, missing technical knowledge, increasing context load, and the complexity of long attack horizons. EGATS further allows the system to explore alternative attack paths instead of indefinitely committing to an unsuccessful branch.

The framework is evaluated on CTF-style tasks, individual vulnerable machines, and the GOAD Active Directory environment. It therefore represents one of the strongest examples of an LLM-based penetration-testing system evaluated across both single-target and multi-host environments.

2.3 Evaluation Methodologies and Benchmarks

The evaluation methodology strongly influences the conclusions that can be drawn from autonomous penetration-testing systems. Existing benchmarks differ in task structure, amount of prior information, execution budget, success criteria, environmental realism, and number of repetitions.

For this reason, benchmark results should not be interpreted solely through their final success rate. The ability to reach intermediate stages, recover from failures, maintain state, and progress across network boundaries can be equally important when evaluating autonomous attack chains.

2.3.1 Benchmarks and Evaluation Environments

CyBench [36] evaluates professional CTF tasks in instrumented Kali Linux environments and provides intermediate subtasks as well as human first-solve time as a difficulty proxy. It is useful for evaluating cyber reasoning and tool interaction across a broad range of security tasks, but it does not explicitly model persistent enterprise attack campaigns.

AutoPenBench [10] focuses more directly on automated vulnerability testing through reproducible tasks with binary objectives and stage-level milestones. It provides one of the most useful common reference points for comparing autonomous penetration-testing agents, although its targets remain independent systems.

PentestEval [34] addresses diagnostic granularity by decomposing penetration testing into multiple stages and providing expert annotations for hundreds of tasks across vulnerable scenarios. Its Sequential Modular Pipeline (SMP) additionally illustrates how errors introduced during early stages can propagate into later exploitation decisions.

TermiAgent and TermiBench [22] adopt a different evaluation philosophy by focusing on realistic system compromise rather than simple flag discovery. The benchmark includes a large collection of hosts and services and requires agents to perform reconnaissance, distinguish relevant targets from benign services, and execute successful attack sequences under incomplete information.

The Game of Active Directory (GOAD) [23] provides a complementary environment for multi-host attack progression. Its interconnected enterprise environment requires reasoning about multiple Windows systems, accounts, credentials, and privileges whose relationships evolve during the engagement.

2.3.2 The Operational-Realism Gap

A central limitation of many existing benchmarks is the difference between *solving a vulnerability* and *conducting an autonomous attack campaign*.

An isolated vulnerability benchmark typically defines a target and an objective. The agent succeeds if it can identify the relevant weakness, execute an appropriate attack, and achieve the expected result.

A multi-host campaign introduces additional dependencies. Credentials obtained on one host may be necessary to access another, a compromised host may expose a new subnet, and privilege escalation may enable operations that were previously impossible.

Consequently, successful exploit-chain execution requires a combination of:

- effective reconnaissance;
- appropriate exploit selection;
- reliable tool execution;
- persistent state management;
- failure recovery;

- privilege escalation;
- credential reuse;
- lateral movement and pivoting.

Failure in any one of these components can prevent completion of the entire chain. Final task completion alone is therefore insufficient to identify the source of failure.

Missotten et al. [24] map existing benchmarks to the MITRE ATT&CK Enterprise Matrix and show that post-compromise behaviours such as persistence, lateral movement, and causally linked objectives are substantially underrepresented.

Happe and Cito [12] similarly identify substantial inconsistencies in metrics, execution budgets, number of repetitions, intermediate-step reporting, and stopping conditions across LLM penetration- testing evaluations. They argue for reproducible environments, explicit baselines, repeated trials, and reporting that goes beyond final success.

These observations motivate a distinction between *vulnerability exploitation* and *exploit-chain execution*. The former demonstrates that an agent can identify and exploit a particular weakness; the latter additionally requires the agent to preserve and reuse information across multiple causally dependent stages.

2.3.3 Quantitative Comparison

To support the comparative analysis presented in this work, a structured literature review was conducted on representative LLM-based penetration-testing systems published between 2024 and 2026. Systems were selected according to their relevance to autonomous exploit-chain execution, with particular attention to their ability to perform dependent actions, preserve execution state, and extend an attack beyond an initially accessible target.

The comparison uses the same three capability levels introduced at the beginning of the chapter as an interpretive framework: multi-step exploitation, stateful penetration testing, and multi-host attack chains.

Table 2.1 summarizes representative systems according to their primary architectural contribution, state-management mechanism, demonstrated attack scope, and evaluation scope.

AutoPenBench Comparison

AutoPenBench [10] provides the benchmark with the largest degree of overlap between several systems in the reviewed corpus. The comparison is not perfectly homogeneous because model choice, execution budget, number of repetitions, and aggregation method differ between studies.

The distinction between *any-of-five* and *pooled* subtask completion is particularly important. Under an any-of-five evaluation, a task is counted as successful when it is completed at least once across five independent attempts. Pooled completion measures the fraction of successful executions across all recorded runs.

Table 2.1: Representative LLM-based penetration-testing systems classified according to their main contribution, state management, demonstrated attack scope, and evaluation scope.

System	Main contribution	State management	Attack scope	Evaluation scope
PentestGPT [5]	Modular reasoning and task decomposition	Task tree and context management	Single-host	Real-world targets; CTF
AutoPT [31]	Penetration Testing State Machine	Explicit finite-state model	Single-host	End-to-end web penetration testing
PentestAgent [29]	Multi-agent architecture with RAG	Workflow and phase state	Single-host	VulHub; HackTheBox CTF challenges
VulnBot [14]	Specialized agents and Penetration Task Graph	Task graph and shared information	Single-host	AutoPenBench; AI-Pentest-Benchmark
CurriculumPT [32]	Curriculum scheduling and experience reuse	Experience Knowledge Base	Single-host	15 real-world CVE exploitation scenarios
RedTeamLLM [3]	Summarization, reasoning, and plan correction	External memory and summaries	Single-host	CTF challenges
APT-Agent [17]	Error rectification and command-specific memory	Persistent execution memory	Single-host	Metasploitable2 services
STT [25]	MITRE ATT&CK-guided task-tree reasoning	Structured task representation	Single-host	10 HackTheBox exercises
xOffense [21]	Multi-agent orchestration and domain adaptation	Workflow and knowledge state	Single-host	AutoPenBench; AI-Pentest-Benchmark
TermiAgent [22]	Located Memory Activation and structured knowledge	Persistent selective memory	Single-host	TermiBench (real-world benchmark)
PenExpert [18]	Multi-agent coordination and structured attack state	Hosts, credentials, routes, and privileges	**Multi-host**	AutoPenBench; VulnHub; cross-subnet
Cochise [13]	Planner–Executor with externalized long-term state	Persistent state outside LLM context	**Multi-host**	GOAD Active Directory
PentestGPT V2 [4]	Tool/Skill Layer, TDA, and EGATS	Persistent state and attack-tree search	**Multi-host**	XBOW; PentestGPT-Bench; GOAD

Table 2.2: Published results on AutoPenBench (33 tasks). Sources: [10, 14, 18, 21].

System	Model	Pass@k	Milestone %*	Protocol
AutoPenBench	GPT-4o	21.21%	—	$k = 1$
VulnBot	Llama-3.1-405B	30.30%	69.05%	Pass@5
PenExpert	GPT-4o	45.45%	78.78%	Pass@5
PenExpert	Claude Sonnet 4.5	72.73%	90.20%	Pass@5
xOffense	Qwen3-32B	30.30%	52.36%	Pass@5
xOffense	Qwen3-32B (fine-tuned)	72.72%	79.17%	Pass@5

*Milestone/sub-task completion rates use study-specific sub-task decompositions of AutoPenBench that differ across sources; values are indicative and not strictly comparable across rows. The Pass@k column, based on the benchmark's fixed 33-task suite, remains directly comparable.

The results illustrate why different studies should not be interpreted as direct architectural comparisons. Even on the same benchmark, performance depends on the backbone model, fine-tuning procedure, execution budget, number of repetitions, retrieval configuration, and scoring protocol.

In particular, a best-of-five value should not be interpreted as the probability of successful execution in a single run. A single successful run is sufficient for a tast to be counted under the any-of-five criterion. Pooled results instead measure the fraction of successful executions across the complete set of runs and therefore capture per-run reliability directly.

AutoPenBench is therefore particularly useful as a measure of vulnerability testing and tool-use capability, while providing less direct evidence about persistent attack-state management across an enterprise network.

Multi-Host Network Comparison

PenExpert [18] provides a direct multi-host evaluation using a cross-subnet network in which progress through one network segment can enable access to subsequent segments. The evaluation compares several agent frameworks under the same environment and reports both mean subnet depth and the deepest endpoint reached.

These results illustrate the qualitative difference between single-host exploitation and network-level attack chaining. Progressing to a deeper subnet requires the agent to preserve and correctly reuse information obtained during previous compromises, including routes, credentials, and privileges.

Subnet depth should nevertheless be interpreted carefully. Reaching a deep segment in one run does not imply reliable completion of the campaign objective. Multi-host evaluation should ideally report depth together with the probability of reaching each stage and the success rate of the final objective.

GOAD Active Directory Evaluation

The Game of Active Directory (GOAD) [23] provides a complementary enterprise-scale evaluation environment. Unlike isolated vulnerable hosts, GOAD requires the agent to

reason about a network of Windows systems, accounts, credentials, and privileges whose relationships evolve during the engagement.

Cochise [13] evaluates autonomous assumed-breach penetration testing on GOAD using a lightweight Planner–Executor architecture, while PentestGPT V2 [4] evaluates a more elaborate tool-integrated architecture with task-difficulty estimation and attack-tree search.

Table 2.3: Representative results on the five-host GOAD Active Directory environment. Sources: [4, 13].

System	Model	GOAD scope	Reported result
PentestGPT V2	GPT-5.2	5 hosts	Up to 4 hosts compromised
PentestGPT V2	Opus 4.5	5 hosts	Up to 4 hosts compromised
Prior baselines	Various	5 hosts	Up to 2 hosts compromised
Cochise	Various LLMs	GOAD AD network	Autonomous account compromise

The importance of GOAD lies in the dependency structure of the task. The agent is not simply required to exploit a vulnerable service. It must discover the environment, obtain credentials, escalate privileges, and use artifacts from previous stages to progress through the domain. Consequently, GOAD is a more direct test of stateful and multi-host reasoning than isolated CVE benchmarks.

PenExpert Evaluation

The multi-host network environment used by PenExpert provides a rigorous testbed for assessing deep network traversal. Unlike flat environments, this architecture requires the agent to systematically discover unknown subnets, execute lateral movements, and perform successive privilege escalations across multiple layers.

Frameworks such as PentestGPT, VulnBot, RedTeamLLM, and PenExpert [18] evaluate multi-step capabilities under baseline configurations.

Table 2.4: Mean subnet depth reached on the multi-host network used by PenExpert (mean $\pm$ standard deviation over five runs). Source: [18].

Framework	Model	Subnet depth ($\mathrm{mean} \pm \mathrm{SD}$)	Best endpoint
PentestGPT	GPT-4o	1.40 $\pm$ 0.55	Subnet 2 / exploit
PentestGPT	Claude Sonnet 4.5	2.60 $\pm$ 0.55	Subnet 3 / privilege
VulnBot	GPT-4o	1.40 $\pm$ 0.55	Subnet 2 / exploit
VulnBot	Claude Sonnet 4.5	2.40 $\pm$ 0.55	Subnet 3 / privilege
RedTeamLLM	Claude Sonnet 4.5	2.60 $\pm$ 0.55	Subnet 3 / privilege
STT	Claude Sonnet 4.5	2.60 $\pm$ 0.55	Subnet 3 / privilege
PenExpert	**GPT-4o**	**3.80 $\pm$ 1.30**	**Subnet 5 / privilege**
PenExpert	**Claude Sonnet 4.5**	**4.20 $\pm$ 1.10**	**Subnet 5 / privilege**
PenExpert	Qwen3-32B	1.60 $\pm$ 0.55	Subnet 2 / lateral

The significance of these results lies in the sequential dependency of network penetration. The agent cannot merely exploit an isolated service; it must maintain

long-term contextual memory and accurate multi-step reasoning to bridge multiple boundaries. Consequently, PenExpert coupled with frontier models successfully navigates deeper topologies, whereas smaller open-weights models experience severe bottlenecks during lateral traversal.

2.3.4 Limits of Quantitative Comparability

Published results should only be compared directly when the relevant experimental conditions are sufficiently aligned. Important variables include:

- the underlying LLM and its model version;
- inference temperature and sampling configuration;
- execution budget and maximum number of actions;
- number of independent repetitions;
- availability of retrieval and external knowledge;
- tool interfaces and execution privileges;
- initial information provided to the agent;
- stopping conditions;
- definition of success;
- aggregation method.

The relationship between architecture and the underlying model is particularly important. Deng et al. [4] evaluate a broad range of LLM-based penetration-testing systems and distinguish capability failures, which may often be addressed through better tools or prompts, from planning and state-management failures that remain more persistent. Their results also indicate that stronger frontier models can reduce the apparent advantage of some specialized architectures.

For this reason, architectural claims should preferably be supported by model-matched comparisons. Best-of-N results should be distinguished from pooled execution results, while subnet depth should be reported together with the reliability of reaching the corresponding objective.

2.4 Research Gap and Positioning of THESEUS

The literature reviewed in this chapter demonstrates a clear progression from LLM-assisted vulnerability exploitation toward increasingly autonomous, stateful, and multi-host penetration testing.

Early approaches established that LLMs can reason about vulnerabilities and generate executable actions. PentestGPT introduced iterative penetration testing and task-oriented context management, while AutoPT made workflow state explicit through

a penetration-testing state machine. Subsequent systems introduced specialized agents, structured task representations, external memory, experience reuse, and mechanisms for correcting execution errors.

A second transition concerns the persistence of operational state. Systems such as CurriculumPT, RedTeamLLM, APT-Agent, and PentestGPT V2 demonstrate that maintaining information outside the immediate LLM reasoning context is increasingly important for long execution horizons.

The most significant transition is from stateful single-host penetration testing to multi-host attack chains. PenExpert explicitly evaluates cross-subnet progression and represents hosts, credentials, routes, and privileges as persistent state. Cochise and PentestGPT V2 further demonstrate that autonomous agents can operate in multi-host enterprise environments such as GOAD.

Despite this progress, several challenges remain.

2.4.1 Persistent State Under Context Constraints

Long execution traces can rapidly increase the amount of information that must be processed by the LLM. Raw command output, intermediate observations, and previous actions are often much larger than the compact set of facts actually required for subsequent decisions.

This problem is particularly relevant to local inference, where context length and computational resources may be more constrained than in cloud-based deployments. An autonomous penetration-testing architecture therefore requires a mechanism for externalizing persistent engagement state while retaining sufficient information for accurate decision making.

2.4.2 Reliable Technical Decision-Making

Large Language Models remain fundamentally probabilistic systems, meaning they can generate commands, payload configurations, or operational actions that are syntactically valid yet technically incompatible with the target environment. Such errors typically stem from incorrect assumptions regarding operating systems, architectures, underlying services, network connectivity, or active session states.

Within extended attack chains, these failures carry a high cumulative cost, as a single erroneous decision can easily cascade into subsequent phases or trap the agent in loops of repetitive, unsuccessful actions.

This operational limitation highlights the necessity of hybrid architectures: delegating decisions to deterministic or constrained components whenever correctness can be derived from explicit technical properties, while reserving LLM reasoning strictly for tasks that demand dynamic interpretation and strategic adaptation.

2.4.3 Causal Multi-Host Attack Progression

Multi-host attack chains introduce a further requirement: information obtained from one compromised asset must be usable to drive subsequent decisions.

Credentials, sessions, privileges, interfaces, routes, and discovered hosts are not independent pieces of information. Their relationships determine which targets are reachable and which operations are feasible.

Consequently, a system that merely stores a list of discovered facts may still fail to reason about the evolving attack surface. A persistent representation must capture the relationships between assets and the operational state associated with them.

2.4.4 Positioning of THESEUS

THESEUS is positioned at the intersection of these challenges. The framework combines a stateful orchestration layer with specialized stateless worker agents, while externalizing persistent attack knowledge into an Asset-Oriented Sequential Attack Knowledge Graph (AKG).

The separation between the orchestrator and worker agents allows individual workers to operate on a bounded context while the persistent state of the campaign remains external to their local reasoning process. The AKG maintains the information required to understand the evolving engagement, including assets, services, credentials, sessions, privileges, interfaces, routes, and relationships between compromised systems and reachable network segments.

THESEUS additionally employs deterministic mechanisms for selected technical decisions that would otherwise depend entirely on unconstrained LLM generation. This design aims to reduce avoidable execution errors while retaining LLM-based reasoning for target selection, interpretation of observations, and adaptive progression.

The resulting architecture is specifically designed to support multi-hop attack progression. The compromise of one host is treated as a state transition that can create new sessions, reveal credentials or network interfaces, establish new routes, and expose previously unreachable targets. These state changes are externalized into the AKG and subsequently made available to the orchestrator when selecting the next stage of the campaign.

THESEUS therefore differs from approaches focused exclusively on isolated vulnerability exploitation or generic multi-agent coordination. Its central research question is whether the combination of specialized stateless workers, externalized structured memory, deterministic technical decision components, and explicit multi-hop state management can support reliable end-to-end penetration testing under the computational and context constraints of local LLM inference.

This positioning also motivates the evaluation methodology adopted in the remainder of the thesis. Single-host vulnerability benchmarks remain useful for measuring basic exploitation and tool-use capabilities, but they are not sufficient to establish successful exploit-chain execution. The evaluation must additionally consider whether the system can preserve campaign state, reuse artifacts obtained during previous stages, recover from unsuccessful actions, establish and use pivot paths, and progress toward previously unreachable hosts.

The following chapters therefore move from the conceptual and architectural foundations established here to the design and implementation of THESEUS, with particular emphasis on its orchestration mechanism, stateless worker agents, and Asset-Oriented Sequential Attack Knowledge Graph.

Chapter 3

THESEUS Framework System Architecture & Detailed Design

3.1 System Overview & Core Architectural Principles

Automating penetration testing operations within complex enterprise environments presents a fundamental conflict between cognitive autonomy and operational determinism. Modern offensive campaigns require continuous, multi-step adaptive decision-making: evaluating diverse attack vectors, prioritizing high-value targets, interpreting non-standard application behaviors, and dynamically altering strategies when new information becomes available. Large Language Models (LLMs) provide powerful capabilities for high-level reasoning, target prioritization, and contextual action selection[35]. However, directly deploying LLM agents as unrestricted execution controllers introduces several operational limitations:

1. **Non-Deterministic Execution:** LLMs may generate invalid command syntax, select incompatible tool parameters, or produce actions that are inconsistent with the current target state. Such failures are a known limitation of LM agents interacting with external tools [28].

2. **Context Window Degradation:** Penetration testing generates large quantities of reconnaissance results, service information, command output, and execution telemetry. Maintaining this information entirely within the conversational context becomes increasingly inefficient during long-running and multi-hop campaigns, leading to performance degradation and attention dilution across extended sequences [19].

3. **Lack of Persistent Operational State:** Conversational context is not a reliable representation of campaign state. Interruptions, inference failures, or context resets can result in the loss of previously discovered hosts, services, credentials, sessions, and routing information.

4. **Unrestricted Execution Risk:** Granting a general-purpose LLM unrestricted access to system and network execution primitives makes it difficult to enforce deterministic scope boundaries and operational safety constraints.

THESEUS addresses these limitations through a hybrid architecture that separates strategic reasoning from operational execution. The system combines a centralized strategic controller, referred to as the *Orchestrator*, which coordinates the overall penetration-testing campaign, specialized stateless worker agents, deterministic execution middleware, and an externalized graph-based state representation, referred to as the *Asset-Oriented Sequential Attack Knowledge Graph (AKG)*, which stores the persistent campaign state.

The architecture is based on four complementary principles:

1. **Separation of Reasoning and Execution:** Strategic decisions are performed by the *Orchestrator*, while concrete reconnaissance, exploitation, and post-exploitation operations are delegated to specialized workers with restricted toolsets.

2. **Stateless Worker Execution:** *Scout*, the agent responsible for reconnaissance and information gathering, *Coder*, the agent responsible for exploit development and code generation, and *PostExploiter*, the agent responsible for post-exploitation activities, do not maintain the global campaign state across task invocations. Each worker receives the information required for its current task and externalizes relevant findings through the AKG.

3. **Externalized Campaign State:** Environmental observations and execution artifacts are maintained in the AKG, rather than relying on the accumulation of raw conversational history.

4. **Deterministic Execution Control:** Tool interfaces enforce operational constraints independently of the LLM, including role-specific tool access, target scope validation, module restrictions, and deterministic payload selection.

3.1.1 Closed-Loop Control Architecture

THESEUS operates as a closed-loop control system in which the state of the environment continuously influences subsequent actions [35]. Rather than executing a statically predefined sequence of penetration-testing operations, the framework repeatedly observes the current campaign state, reasons over the available information, selects an appropriate action, and incorporates the resulting observations into the persistent campaign state.

The fundamental control cycle is therefore defined as illustrated in Figure 3.1.

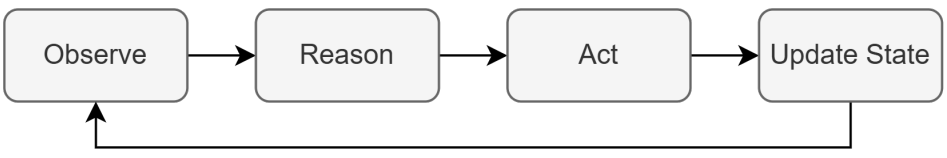

Figure 3.1: Fundamental control cycle

The **Observe** stage collects information about the current attack environment. Observations may originate from reconnaissance operations, service enumeration, exploitation results, established sessions, credential discovery, or newly identified network interfaces.

During **Reason**, the Orchestrator evaluates the current campaign state and determines the next operational objective. The decision is based on the state represented in the AKG rather than on the complete history of previous tool executions.

The **Act** stage delegates the selected objective to the appropriate specialized worker. Depending on the current target state, the action may correspond to reconnaissance, exploitation, post-exploitation, or pivot establishment.

Finally, **Update State** incorporates the outcome of the operation into the AKG. Successful actions may enrich existing nodes, create new assets, establish sessions or credentials, or expose previously unreachable network segments. Failed actions may instead provide diagnostic information that influences subsequent decision-making.

The updated state consequently becomes the observation basis for the next iteration of the control loop. This feedback mechanism allows THESEUS to adapt its attack strategy as the environment changes and to progressively extend the reachable attack surface during multi-hop campaigns.

3.1.2 Operational Lifecycle

The closed-loop architecture operates at two complementary levels. At the **campaign level**, the Orchestrator manages the overall progression of the penetration-testing engagement. At the **target level**, each selected asset is processed through a standardized tactical lifecycle implemented by the specialized worker agents.

Campaign-Level Lifecycle

A campaign begins with the initialization of the target scope and the corresponding campaign state. The Orchestrator then evaluates the currently known attack surface and selects an appropriate target for tactical processing. Following execution, the resulting observations are incorporated into the AKG and the campaign state is evaluated again.

The evaluation stage determines whether additional targets remain reachable, whether newly discovered assets require processing, whether further pivoting is possible, or whether the campaign objective has been satisfied. If the campaign is not complete, the updated state is fed back into the next iteration of the lifecycle.

This design allows the number of tactical iterations to be determined by the environment rather than by a fixed sequence of operations. In particular, the discovery of a new subnet or host can dynamically expand the set of candidate targets and cause additional tactical cycles to be initiated.

The campaign-level loop terminates when one of the following conditions is reached: (i) the campaign objective is successfully achieved, (ii) no unexplored reachable targets or actionable attack paths remain, or (iii) the configured operational constraints prevent further execution.

Per-Target Tactical Lifecycle

Once the Orchestrator selects a target asset, THESEUS applies a standardized three-phase tactical lifecycle. The phases represent the available operational capabilities rather than a fixed sequential pipeline. Each phase is executed by a dedicated stateless worker agent, after which control is always returned to the Orchestrator. The Orchestrator evaluates the resulting AKG state and determines the next action based on the outcome of the completed phase.

The resulting control pattern is therefore:

$$\text{Orchestrator} \rightarrow \text{Worker} \rightarrow \text{Orchestrator} \rightarrow \text{Worker} \rightarrow \cdots$$

For a given target, the available tactical phases are:

1. **Scout Phase:** The Orchestrator delegates reconnaissance to the Scout agent, providing the target scope and relevant operational constraints. The Scout performs reconnaissance and service enumeration against the selected target, establishing a structured representation of the reachable attack surface, including hosts, ports, services, application paths, and relevant fingerprinting information. Once the task terminates, the Scout returns control to the Orchestrator together with a concise execution result, while detailed findings are externalized into the AKG.

2. **Coder Phase:** Based on the updated AKG state, the Orchestrator determines whether exploitation should be attempted and, if appropriate, delegates the task to the Coder agent. The Coder evaluates available exploitation opportunities and executes an authorized exploitation attempt. Exploit and payload selection are constrained by deterministic execution mechanisms and by the target information recorded in the AKG. After the attempt completes, control is returned to the Orchestrator, which evaluates the resulting session state and determines whether further exploitation, recovery, or another action is required.

3. **PostExploiter Phase:** If the Coder establishes a successful compromise, the Orchestrator may delegate post-exploitation to the PostExploiter agent. The agent performs post-compromise enumeration and extracts relevant operational findings, including credential discovery, network interface enumeration, and identification of additional reachable subnets. Upon completion, control is again returned to the Orchestrator. Newly discovered connectivity may result in the establishment of a pivot, expanding the campaign-level attack surface and introducing new target assets into the AKG.

The completion of a phase does not implicitly trigger the next phase. Instead, each worker execution constitutes an atomic tactical delegation followed by a new decision point at the Orchestrator. Consequently, after Scout execution, the Orchestrator may delegate exploitation, request additional reconnaissance, or abandon the target. Similarly, after a Coder execution, it may delegate post-exploitation, request a recovery attempt, or select another target. PostExploiter results may likewise cause the Orchestrator to continue operating on the current host or to redirect execution towards newly discovered assets.

This control structure also allows unsuccessful executions to be incorporated into the global decision process. Reconnaissance may reveal that no exploitable service is available, while exploitation may fail to establish a session. In both cases, the worker returns control to the Orchestrator together with the resulting state and diagnostic information. The Orchestrator can then select an alternative action or target rather than mechanically advancing to the next phase.

Conversely, successful post-exploitation may reveal additional hosts or network segments. These newly discovered assets are added to the AKG and become candidates for subsequent iterations of the campaign lifecycle. Therefore, the three phases should be understood as specialized tactical capabilities coordinated through repeated Orchestrator–worker interactions, rather than as a fixed linear execution pipeline.

3.2 Asset-Oriented Sequential Attack Knowledge Graph (AKG)

3.2.1 Formal Graph Definition & Schema

The AKG serves as the authoritative representation of the campaign state. To ensure formal rigorousness, the AKG models the target environment and all execution artifacts as an extended Labeled Property Graph (LPG) defined by the 6-tuple:

$$G = (V, E, P, L, \mathcal{S}, \Phi) \tag{3.1}$$

Where each element of the tuple is defined as follows:

- $V = \{v_1, v_2, \ldots, v_n\}$ is the set of graph nodes representing concrete structural entities (subnets, hosts, services) and operational execution artifacts (exploits, credentials, campaign objective artifacts).
- $E \subseteq V \times V$ is the set of ordered pairs representing directed operational and structural relationship edges between nodes.
- $P : (V \cup E) \times \mathcal{K} \rightarrow \mathcal{V}$ is a key-value mapping function that attaches arbitrary dynamic property attributes (e.g., IP addresses, service banners, session identifiers, vulnerability scores) to nodes and edges.
- $L : (V \cup E) \rightarrow \mathcal{L}$ is a typing function mapping every node and edge to a predefined set of canonical taxonomies.
- $\mathcal{S} : V \rightarrow$ Provenance is a meta-information mapping function storing structural provenance for auditability, recording tool signatures, raw execution arguments, timestamp vectors, and triggering agent IDs for every state mutation.
- $\Phi : V \rightarrow \mathbb{N}_0$ is a topological depth mapping function calculating the network hop distance of any given asset or exploit step relative to the primary attacker ingress interface. Its formal computation is defined in Section 3.2.4; its operational role in guiding session and route restoration is discussed in Section 3.5.3.

Although Φ is formally defined over the entire node set V, its value is computed directly only for `Subnet`, `Host`, and `ExploitStep` nodes, since these are the only entity types connected by the `PIVOTS_INTO` relationship that increments hop depth (Section 3.2.4). All other node types (`Service`, `Interface`, `Credential`) are structurally bound to a parent `Host` or `ExploitStep` node via zero-weight edges and therefore inherit Φ from that parent rather than being traversed independently.

Comprehensive Node Taxonomy (V)

The node set V is partitioned into six mutually exclusive canonical entity types, each defined by a specific set of required properties:

- **Subnet Node ($\mathcal{L}(v) =$ `Subnet`):** Represents an IPv4/IPv6 network segment (e.g., `172.31.0.0/16`). It maintains functional attributes such as `cidr`, `netmask`, `gateway_ip`, and the boolean flag `is_attacker_network`, which distinguishes attacker-controlled DMZ segments from isolated target subnets.
- **Host Node ($\mathcal{L}(v) =$ `Host`):** Represents a target machine identified by its canonical primary IP address. It maintains phase progress flags (`scout_status`, `exploit_status`, `postexploit_status`) transitioning through the discrete values `pending`, `done`, `unsuccessful` — exclusively mutated by the Orchestrator via `akg_set_phase_status` upon evaluation of a worker's result, with the sole exception of the host's operating status transitioning to `pwned` as part of its own transaction (Section 3.3) — as well as operating status (*up*, *down*, *pwned*) and OS guess confidence scores.
- **Interface Node ($\mathcal{L}(v) =$ `Interface`):** Represents physical or virtual Network Interface Cards (NICs) discovered on compromised target hosts during post-exploitation (e.g., `eth0: 172.28.0.5`, `eth1: 172.30.0.5`). These nodes serve as critical logical anchors for multi-homed pivot discovery.
- **Service Node ($\mathcal{L}(v) =$ `Service`):** Represents open transport-layer ports and running network applications. Key properties include `port`, `protocol` (`tcp`/`udp`), `banner`, `service_name`, `target_os`, `target_arch`, `web_path`, and an array of identified `cves` (e.g., `["CVE-2017-5638"]`).
- **ExploitStep Node ($\mathcal{L}(v) =$ `ExploitStep`):** Encapsulates a successful exploitation event or session creation, whether obtained through vulnerability exploitation or through credential-based authentication (e.g., a successful `msf_login` attempt). It records `session_id`, `exploit_module`, `payload_used`, `execution_timestamp`, `obtained_user_privilege` (*root*, *SYSTEM*, *www-data*), `lhost`, `lport`, `session_type` (`meterpreter` vs. `shell`), `cve` (where applicable), and key module option overrides.
- **Credential Node ($\mathcal{L}(v) =$ `Credential`):** Stores security secrets and campaign objective artifacts harvested during reconnaissance or post-exploitation. It categorizes entries via `cred_type` (*plaintext*, *shadow_hash*, *ntlm_hash*, *jwt_token*, *ssh_key*, *flag*) along with the associated `username` (where applicable), a `secret`

string, a `location` descriptor, and the originating `exploit_session_id`. The *flag* category is reserved for objective markers extracted during authorized assessments (e.g., benchmark or CTF-style engagements) and is treated identically to any other secret for provenance and reporting purposes.

Edge Taxonomy & Semantic Constraints (E)

Relationships between nodes enforce strict directional and semantic integrity constraints within the graph:

$$e_1 = (v_{\text{Host}}, v_{\text{Interface}}) \in E \quad \text{labeled as } \texttt{HAS_INTERFACE} \tag{3.2}$$
$$e_2 = (v_{\text{Interface}}, v_{\text{Subnet}}) \in E \quad \text{labeled as } \texttt{CONNECTS_TO} \tag{3.3}$$
$$e_3 = (v_{\text{Host}}, v_{\text{Service}}) \in E \quad \text{labeled as } \texttt{RUNS} \tag{3.4}$$
$$e_4 = (v_{\text{Service}}, v_{\text{ExploitStep}}) \in E \quad \text{labeled as } \texttt{EXPLOITED_VIA} \tag{3.5}$$
$$e_5 = (v_{\text{ExploitStep}}, v_{\text{Host}}) \in E \quad \text{labeled as } \texttt{COMPROMISES} \tag{3.6}$$
$$e_6 = (v_{\text{ExploitStep}}, v_{\text{Credential}}) \in E \quad \text{labeled as } \texttt{YIELDS} \tag{3.7}$$
$$e_7 = (v_{\text{Service}}, v_{\text{Credential}}) \in E \quad \text{labeled as } \texttt{LEAKS} \tag{3.8}$$
$$e_8 = (v_{\text{ExploitStep}}, v_{\text{Subnet}}) \in E \quad \text{labeled as } \texttt{PIVOTS_INTO} \tag{3.9}$$

Two edges lead to `Credential` nodes with distinct provenance semantics. The `YIELDS` edge (e_6) represents credentials *harvested post-compromise* through an established session (e.g., dumped hashes, local configuration secrets, or objective flags) and is created by the PostExploiter agent. The `LEAKS` edge (e_7) instead represents credentials *passively exposed* by a service prior to any compromise (e.g., default credentials in a banner, secrets embedded in an exposed configuration file discovered via `curl_request`) and is created by the Scout agent, which is otherwise restricted to non-invasive discovery.

The `PIVOTS_INTO` edge is annotated with the two properties `via_session_id` and `registered_route_cidr`, explicitly defining the internal routing path through which the subnet becomes accessible. This edge is only ever created *after* the underlying `HAS_INTERFACE` and `CONNECTS_TO` edges have already exposed the new subnet's topological position: it represents the operational routing decision, not the raw interface discovery itself. Because `via_session_id` is a mutable reference to a live Metasploit session, it is not necessarily stable for the lifetime of the campaign; the mechanism that keeps it consistent when a session is upgraded, together with the full pivoting and route-restoration procedure, is described in Section 3.5.

3.2.2 State Offloading Strategy & Context Isolation

Standard autonomous agent frameworks may suffer from progressive context degradation when raw shell outputs, verbose Nmap logs, and full HTTP responses are repeatedly concatenated into the primary reasoning context. Over a multi-hop penetration testing campaign involving multiple targets, the execution history can grow rapidly as observations, tool outputs, and previous actions accumulate [33]. This increases inference latency and token consumption, while also making it more difficult for the

model to identify the information relevant to the current step and maintain consistent instruction following.

THESEUS resolves this through an explicit *Context Isolation Strategy* built upon agent task delegation and externalized knowledge mutation. Rather than attempting execution-layer console parsing via fragile regex or AST heuristics, THESEUS assigns operational tasks to specialized, stateless worker agents (Scout, Coder, and PostExploiter). Upon executing network or post-exploitation actions, worker agents extract key findings within their local task lifecycle and perform graph updates on the AKG using dedicated graph-mutation tools (e.g., `akg_add_service`, `akg_record_exploit`, `akg_add_credential`, `akg_add_pivot` — the complete role-scoped toolsets are detailed in Section 3.3). In addition to updating the AKG state, subagents return a concise textual execution report back to the Orchestrator upon task completion, providing immediate high-level feedback while keeping raw execution logs strictly isolated from the global context window.

The central Orchestrator agent never ingests raw terminal execution traces. Instead, it maintains macro-level context awareness by querying the AKG via two projection endpoints, both of which resolve to the same underlying representation of the full campaign state:

- `akg_get_context()`: Computes a compressed textual summary of the entire campaign graph, including host statuses, phase indicators (`DONE`, `PENDING`), discovered service attributes, active Meterpreter/shell sessions, established pivot routes, and harvested credentials — formatted strictly for low-token ingestion. This is the Orchestrator's default situational-awareness query at the start of each Reason step. The same endpoint is also exposed to the PostExploiter agent, which invokes it at the start of every stateless task to re-acquire full awareness of the campaign graph before performing enumeration on its assigned session, and to the Coder agent, which invokes it to retrieve saved credentials.

- `akg_get_next_targets()`: Queries the host nodes in the AKG, resolves canonical identities, filters out targets with an established `COMPROMISES` edge or an `exploit_status` set to `DONE`, and returns a list string of uncompromised target IPs for immediate delegation.

```
=== ATTACK KNOWLEDGE GRAPH CONTEXT [an-08-08] ===
Summary: hosts=3 (pwned=2) | sessions=2 | creds=2 | internal_subnets=1 |
    next_targets=['172.32.4.35']

--- TARGET HOSTS & PHASE PROGRESS ---
Host 172.31.0.5 [UP]
  Progress: Scout [DONE] | Exploit [DONE] | PostExploit [DONE]
  Services:
    - Port 8080/tcp | Service: http | Banner: Apache Struts2 | CVEs: CVE
    -2017-5638
  Session:  Session 1 (meterpreter, root) via exploit/multi/http/
    struts2_code_exec_showcase (Port: 8080) [Payload: linux/x64/meterpreter
    /reverse_tcp]
Host 172.32.4.20 [UP]
  Progress: Scout [DONE] | Exploit [DONE] | PostExploit [PENDING]
```

```
  Services:
    - Port 80/tcp | Service: http | Banner: Apache CGI | CVEs: CVE
    -2014-6271
  Session:  Session 5 (meterpreter, root) via exploit/multi/http/
    apache_mod_cgi_bash_env_exec (Port: 80) [Payload: linux/x64/meterpreter
    /reverse_tcp]
Host 172.32.4.35 [UP]
  Progress: Scout [PENDING] | Exploit [PENDING] | PostExploit [PENDING]
  Services:  no services found
  Session:  no active session

--- ESTABLISHED PIVOTS ---
  Session 1 ---> Subnet 172.32.4.0/24 (route: 172.32.4.0/255.255.255.0)

--- HARVESTED CREDENTIALS ---
  [shadow_hash] user=root secret=$6$k8... @ 172.31.0.5
  [plaintext] user=db_user secret=P@ssword123! @ 172.32.4.20
================================================
```

Listing 3.1: Example of the compressed AKG context string generated by `akg_get_context()`.

3.2.3 The Campaign Abstraction & State Persistence Architecture

In **THESEUS**, all offensive actions and telemetry are bound to the formal abstraction of a *Campaign* ($\mathcal{C}$). A campaign represents a discrete, fully autonomous penetration testing engagement executed by the multi-agent team against a designated network target. The framework supports managing multiple campaigns — concurrently active or historical — through a dedicated set of Orchestrator-level tools: `akg_set_campaign` activates a given campaign as the current operational context, `akg_list_campaigns` enumerates all previously initialized campaigns, and `akg_get_current_campaign` resolves the active campaign identifier. This allows an operator (or the Orchestrator itself, upon restart) to unambiguously bind every subsequent tool invocation to a single, well-defined graph instance.

To guarantee complete operational auditability, crash resilience, and real-time UI synchronization, every campaign serializes its state, execution history, and graph topology into a dedicated JSON repository:

$$\texttt{campaigns/<campaign_id>/graph.json} \tag{3.10}$$

The `graph.json` file serves as the concrete, single-file backing store of the AKG, structuring the previously defined tuple elements into three operational components:

1. **Campaign Metadata:** The campaign identifier, target scope definition, engagement objective, creation and last-update timestamps, and the currently active phase status counters, populated and updated via `akg_set_campaign`.

2. **Serialized Graph Topology:** The concrete instantiation of the node and edge taxonomies defined above, mapping active subnets, target hosts, running services, session handles, and harvested credentials alongside their directed operational relationships (`RUNS`, `EXPLOITED_VIA`, `COMPROMISES`, `YIELDS`, `LEAKS`, `PIVOTS_INTO`).

3. **Event Log & Execution Provenance:** An append-only chronological trace capturing agent activity and strategic reasoning:
    - *Operational Provenance*: Records exact tool invocations, execution arguments, and raw tool output telemetry.
    - *Decision Trace*: Preserves the agent-generated tactical decisions and delegation outcomes, explicitly written by the Orchestrator via `akg_log_decision` at every Reason step.

Operational Role of the Campaign State

The persistent campaign state fulfills three complementary system requirements:

- **State Persistence:** Preserves the information required to maintain a coherent representation of the penetration-testing campaign across successive execution cycles and process interruptions. The persisted state includes discovered assets, services, credentials, exploitation results, sessions, and pivot relationships.
- **Redundancy Elimination:** Provides the Orchestrator with a persistent record of previously performed actions and their outcomes, allowing subsequent task selection to account for information and operations already recorded in the campaign state.
- **Real-Time Visual Telemetry:** The `dashboard.py` backend reads state mutations from `graph.json` and streams updates through WebSockets, enabling the dynamic visualization of the evolving attack topology through the D3.js force-directed graph.

The mechanisms used to restore sessions, re-establish routes, and resume execution across multi-hop paths — including how the topological hop depth defined below is used to order that restoration — are described in Section 3.5.3, as part of the pivoting and campaign-continuation mechanisms.

3.2.4 Topological Hop Depth: Formal Definition

The AKG computes a topological network hop depth $\Phi(v)$ for every `Host` and `ExploitStep` node in the graph. This subsection defines Φ formally; its use in ordering session and route restoration during campaign resumption is discussed in Section 3.5.3.

The calculation operates as a modified Breadth-First Search (BFS) starting from the primary attacker interface ($\Phi(v_{\text{att}}) = 0$). Rather than evaluating direct host-to-host edges, the BFS traverses the heterogeneous LPG topology, where host-to-host transitions span a multi-node structural path. Specifically, moving from an originating host v_{h1} to a downstream target host v_{h2} requires traversing an exploitation and pivoting chain:

$$v_{h1} \xrightarrow{\text{RUNS}} v_{\text{service}} \xrightarrow{\text{EXPLOITED_VIA}} v_{\text{exploit}} \xrightarrow{\text{PIVOTS_INTO}} v_{\text{subnet}} \xleftarrow{\text{CONNECTS_TO}} v_{\text{interface}} \xleftarrow{\text{HAS_INTERFACE}} v_{h2} \tag{3.11}$$

During traversal, the BFS increments the hop depth counter strictly when traversing the `PIVOTS_INTO` relationship (from an `ExploitStep` node to a newly exposed `Subnet` node):

$$\Phi(v_{\text{subnet}}) = \Phi(v_{\text{exploit}}) + 1 \tag{3.12}$$

All intra-host and intra-subnet component transitions (e.g., host-to-service, service-to-exploit, or subnet-to-interface) incur zero additional depth weight ($w = 0$), and are the mechanism by which `Service`, `Interface`, and `Credential` nodes inherit their depth from their structural parent, as noted in Section 3.2. Direct DMZ targets are therefore assigned Hop 0, while internal target hosts accessed through N successive pivot sessions are assigned Hop N.

3.3 Multi-Agent Team Architecture & Least Privilege Enforcement

THESEUS leverages the Agno framework [1] to instantiate an autonomous multi-agent architecture and uses the Metasploit Framework [26] as the primary exploitation and session-management backend. Control transfer between the Orchestrator and a worker agent (Scout, Coder, PostExploiter) is handled natively by Agno's team-coordination mechanism. To mitigate the security and operational risks associated with unconstrained agent execution, THESEUS separately enforces Role-Based Access Control at the *tool* level: each agent, once invoked, is restricted to a narrow, specialized domain and deterministic tool-set.

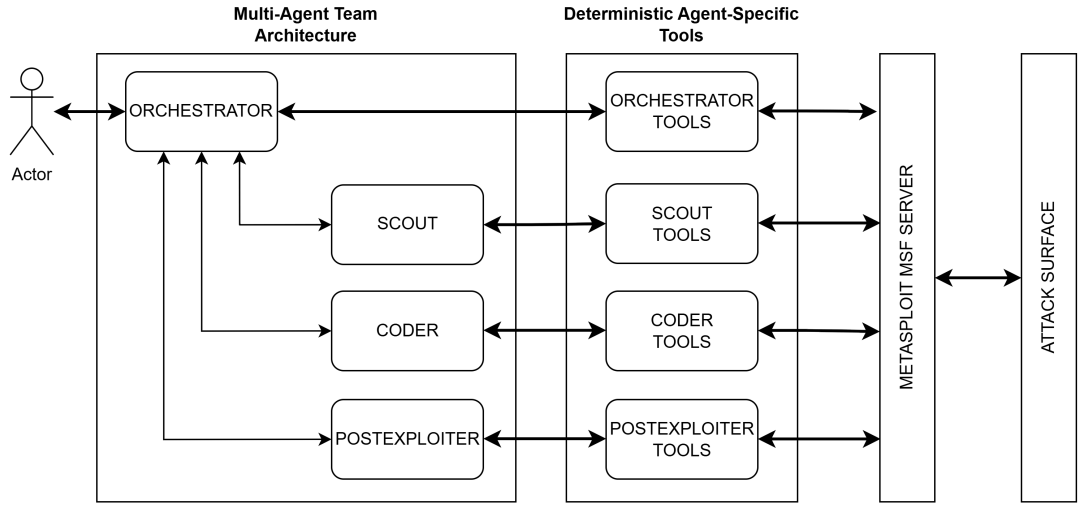

Figure 3.2: **Hierarchical Multi-Agent Architecture.** The Orchestrator manages user interaction and strategic campaign coordination, while specialized worker agents (Scout, Coder, and Post-Exploiter) execute operations within strict least-privilege boundaries, engaging the target attack surface through deterministic, role-specific tools.

1. Orchestrator Agent (Stateful Strategic Controller)

The Orchestrator functions as the centralized cognitive controller and state maintainer for the entire penetration testing lifecycle. It synthesizes global situational awareness from the Attack Knowledge Graph (AKG) and translates high-level evaluation directives into concrete operational sub-goals.

- **Architectural Role & Scoping:** The Orchestrator possesses global visibility over the campaign state, target network topology, and discovered vulnerability telemetry. To eliminate accidental execution of destructive actions during high-level strategic reasoning, the Orchestrator is strictly decoupled from raw execution primitives. It cannot directly spawn shell processes, construct raw network sockets, or directly dispatch payload or exploit modules to target hosts. The single, deliberate exception to this rule is described below.
- **Primary Responsibilities:**
  - Interacting textually with the user to ingest the primary directive and operational parameters.
  - Selecting, listing, and resuming campaigns.
  - Performing coarse, unprivileged host discovery to bootstrap a newly reachable scope (see below).
  - Continuous analysis of aggregated AKG state context to maintain situation awareness (`akg_get_context`).
  - Multi-stage campaign planning and task decomposition.
  - Resuming interrupted or stopped attack campaigns leveraging state data persisted in the AKG (`akg_restore_sessions`).
  - Delegating operational tasks to worker agents with strictly bounded scopes (e.g., target subnets, explicit port lists, or session identifiers).
  - Evaluating worker task completion signals, updating host phase-progress flags (`akg_set_phase_status`), handling execution failures, and logging every tactical decision (`akg_log_decision`).
- **Permitted Toolset:** `akg_set_campaign`, `akg_list_campaigns`, `akg_restore_sessions`, `akg_log_decision`, `akg_get_context`, `akg_get_next_targets`, `akg_add_host`, `akg_add_subnet`, `akg_set_phase_status`, `akg_get_current_campaign`, `orchestrator_discover_subnet`.
- **The `orchestrator_discover_subnet` Bootstrap Primitive:** Every Scout tool operates on an already-known target IP registered in the AKG as a `Host` node — Scout has no subnet-level sweep capability of its own. This creates a bootstrapping problem: at the very first iteration of a campaign, the AKG contains only the operator-supplied subnet scope but zero `Host` nodes to delegate to Scout. `orchestrator_discover_subnet` closes this gap and is, by design, the *only* execution-layer primitive available to the Orchestrator: it first registers the subnet in the AKG (`akg_add_subnet`, with `is_attacker_network=True` for the ingress scope), then locates the attacker-side execution container (`msf-rpc/kali`) to run

the sweep from within the correct network context, then performs a non-invasive, unprivileged network discovery. Every responding address is registered as a `Host` node via `akg_add_host` with `status="up"` and `scout_status="pending"`. It deliberately performs no port scanning, banner grabbing, or service fingerprinting — that remains exclusively Scout's responsibility once a candidate host has been seeded into the AKG.

- **Deterministic Guardrail Policy:** Beyond `orchestrator_discover_subnet`, the Orchestrator's toolset exposes no port-scanning, service-interaction, exploitation, or session-command primitives. Delegations to workers must be accompanied by explicit scope limits (e.g., host range, rate-limiting parameters). Any attempt to invoke execution-layer tools outside this toolset from the Orchestrator's context is rejected at the framework/team level.

2. Scout Agent (Stateless Reconnaissance Worker)

The Scout agent operates as an isolated worker optimized for non-destructive surface discovery, network sweeps, and service fingerprinting *against a specific, already-registered host*. It functions in a stateless manner, receiving task parameters per invocation and pushing structured telemetry directly back into the AKG.

- **Architectural Role & Scoping:** Reconnaissance tasks require broad network exposure but zero execution privileges on target systems. The Scout agent is designed to interact exclusively with passive discovery mechanisms and non-destructive active probes, ensuring that host discovery and service identification cause minimal impact on target infrastructure stability.
- **Primary Responsibilities:**
  - Executing deep port scans, protocol-specific banner grabbing, and Metasploit `auxiliary/*` enumeration modules[26] against a target host already known to the AKG.
  - Parsing web endpoint structures, HTTP header configurations, and software version tags.
  - Cross-referencing identified services, software components, and version tags against CVE databases to enumerate candidate vulnerabilities and associated exploit modules for the target.
  - Ingesting raw network responses, converting unstructured probe outputs into standardized JSON schemas, and populating the host, service, and (where discoverable without compromise) adjacent subnet nodes of the AKG.
  - Registering credentials *passively exposed* by a service prior to any compromise (e.g., default credentials in a banner, secrets in an exposed configuration file), via `LEAKS` edges.
  - Querying the attacker's own local network interfaces to identify candidate egress addresses, feeding downstream LHOST resolution for the Coder agent.

- **Permitted Toolset (Read-Only / Non-Destructive Guarded):** `nmap_scan`, `curl_request`, `msf_search_modules`, `msf_get_module_info`, `msf_execute_scanner` (strictly bounded to Metasploit `auxiliary/*` enumeration modules), `get_network_interfaces` (local/attacker-side only), `akg_add_service`, `akg_add_subnet`, `akg_add_host`, `akg_add_credential`.
- **Deterministic Guardrail Policy:** The tool execution proxy validates all scanner invocations to ensure that target IP addresses match authorized campaign scope boundaries.

3. Coder Agent (Stateless Exploit Automation Worker)

The Coder agent acts as the target engagement specialist. Upon receiving targeted directives from the Orchestrator (e.g., engaging a specific vulnerable service on an identified host), the Coder agent manages module lookup, parameter alignment, payload verification, and deterministic execution.

- **Architectural Role & Scoping:** Initial access operations carry inherent operational risk. To mitigate destabilization risks, the Coder agent is prohibited from issuing unstructured or arbitrary exploit strings against remote targets. Instead, it relies on a deterministic verification pipeline to parameterize pre-verified exploit templates against confirmed target states.
- **Primary Responsibilities:**
  - Querying module databases (`msf_search_modules`, `msf_get_module_info`) to select verified exploits matching the exact version metadata recorded in the AKG by the Scout.
  - Triggering the initial exploitation attempt with zero manual payload configuration via `msf_trigger_exploit_auto`, which resolves the target port automatically from AKG service data and lets Metasploit's own module-default logic select the payload [26] (Section 3.4).
  - Falling back, upon failure of the automatic attempt, to `msf_trigger_exploit` with an explicit payload drawn from the Deterministic Payload Selection Engine described in Section 3.4, together with any required option overrides (e.g., `TARGETURI`, `PHP_FUNC`, `HTTP_METHOD`).
  - Determining the correct network parameters for reverse-callback payloads: `get_network_environment` checks the attacker's own local network setup (network interfaces, MAC addresses, known devices on the local network) to support low-level network scripting, while `get_lhost_for_target` works out which IP address the target should be told to connect back to.
  - Attempting credential-based access via `msf_login` when valid credentials are already recorded in the AKG, as an alternative access vector to vulnerability exploitation, across SSH, FTP, Telnet, SMB, MySQL, and PostgreSQL.
  - Chaining local privilege-escalation exploits (`msf_trigger_local_exploit`) immediately after obtaining a low-privileged session, where appropriate, before handing off to PostExploiter.

  - Using `execute_python_code` for auxiliary computation that falls outside the scope of a pre-packaged Metasploit module — observed uses include cryptographic attacks (XOR analysis, discrete-log/Diffie–Hellman brute-forcing, RSA arithmetic), raw packet crafting and ARP spoofing, and ad-hoc payload/token encoding for web exploitation.
  - Recording the outcome of every exploitation attempt directly into the AKG via `akg_record_exploit`, and any credentials obtained (e.g., via `msf_login`) via `akg_add_credential`.

- **Permitted Toolset:** `msf_trigger_exploit_auto`, `msf_search_modules`, `akg_get_context` `msf_get_module_info`, `msf_get_best_payload`, `msf_trigger_exploit` (strictly guarded to authorized `exploit/*` modules), `msf_trigger_local_exploit`, `msf_execute_scanner` (strictly bounded to `auxiliary/*`), `msf_login`, `execute_python_code`, `get_network_environment`, `get_lhost_for_target`, `akg_record_exploit`, `akg_restore_sessions`, `akg_add_credential`.
- **Deterministic Guardrail Policy:** As with Scout, any `msf_execute_scanner` call remains restricted to `auxiliary/*` regardless of the requesting agent. `execute_python_code` follows a different containment model from the other system-level tool interfaces: rather than invoking a binary directly on the host via a safe argument vector, model-generated code is executed inside a dedicated, policy-constrained Docker container, isolating it from the host filesystem and process space while still permitting the raw-socket and packet-crafting operations its intended use cases require.

4. PostExploiter Agent (Stateless Pivoting & Enumeration Worker)

The PostExploiter agent operates post-compromise within established command interactive sessions or agent handles (e.g., active Meterpreter channels). Its scope focuses on internal host and subnet enumeration, credential recovery, and routing setup for multi-hop network traversal.

- **Architectural Role & Scoping:** Post-exploitation operations require operating within compromised host contexts. To prevent out-of-scope lateral movement or unmonitored host modification, the PostExploiter agent operates within explicit session bindings managed by the backend session router.
- **Primary Responsibilities:**
  - Performing local system telemetry collection, identifying user account configurations, running processes, and network interfaces via arbitrary read-oriented commands issued through `msf_session_command` (e.g., `ip a`, `cat /etc/passwd`), and registering discovered interfaces via `akg_add_interface`.
  - Upgrading a plain command shell session to a full Meterpreter session when advanced post-exploitation capabilities are required but the initial access vector only yielded a basic shell. This upgrade is idempotent: a session already running Meterpreter is returned as-is without re-invoking the module. Its effect on routing consistency is described in Section 3.5.

  - Sweeping newly identified internal subnets for adjacent live hosts via `msf_discover_hosts`, registering discovered addresses into the AKG mirroring at the internal-pivot level the same host-registration role that the Orchestrator performs for the initial ingress scope. The full two-stage discovery mechanism is described in Section 3.5.
  - Establishing internal proxy routes (`msf_add_route`) and recording the resulting pivot relationship (`akg_add_pivot`) to expose secondary network segments to the Scout and Coder agents.
  - Extracting local application configuration parameters and authorized credential stores, pushing parsed findings to the AKG credential repository (`akg_add_credential`), including objective artifacts (`cred_type="flag"`) where applicable.
  - Chaining local privilege-escalation exploits (`msf_trigger_local_exploit`) discovered during enumeration, when privilege escalation was not already handled by the Coder during initial access.

- **Permitted Toolset:** `msf_session_command`, `msf_add_route`, `msf_discover_hosts`, `msf_upgrade_shell_to_meterpreter`, `msf_trigger_local_exploit`, `msf_search_modules`, `msf_get_module_info`, `akg_add_interface`, `akg_add_credential`, `akg_add_pivot`, `akg_add_host`, `akg_add_subnet`, `akg_restore_sessions`, `akg_get_context`.

- **Deterministic Guardrail Policy:** Commands executed via `msf_session_command` pass through a command blocklist filter that restricts destructive disk formatting, out-of-scope subnet targeting, and unauthorized credential extraction targeting persistent master key stores.

3.3.1 Least Privilege Verification & Guardrail Enforcement

To ensure that agents cannot bypass their assigned roles (whether through prompt injection, hallucination, or misconfiguration), tool invocations are filtered through strict validation guards prior to execution:

- **Module Path Scope Enforcement:** The `msf_execute_scanner` tool validates that requested module paths start strictly with the prefix `auxiliary/`, regardless of whether it is invoked by Scout or Coder. If an agent attempts to execute an `exploit/` path through it, the execution engine blocks the call and returns a JSON validation error.

- **Role-Based Tool Segregation:** Agent capabilities are isolated at the architecture level by binding distinct toolsets to each role. Scout lacks access to any exploitation-triggering interface; Coder cannot issue interactive session commands or establish routes; the Orchestrator cannot invoke any of the above beyond its single bootstrap discovery primitive.

- **Safe Argument Vector Execution:** System-level tool interfaces (`nmap_scan`, `curl_request`) invoke underlying binaries via explicit argument vectors passed

directly to process execution interfaces (`subprocess.run`) without a shell interpreter (`shell=False`), preventing host command injection without relying on string-based sanitization.

3.3.2 Indirect Prompt Injection Resilience

Autonomous security agents operating against unknown targets face an inherent risk of *Indirect Prompt Injection*[11]: adversaries or honeypots can embed adversarial instructions within HTTP responses, service banners, motd files, or database contents to hijack the LLM controller.

THESEUS mitigates this attack vector architecturally rather than relying on brittle heuristic text filtering:

1. **Semantic Air-Gap:** The strategic Orchestrator never reads raw shell stdout, HTTP response streams, or raw command traces. Its observation space is strictly bounded to the structured projection of the AKG.

2. **Worker Sanitization Layer:** Stateless workers (Scout, Coder, PostExploiter) interact with the untrusted environment, but only commit discrete, typed attributes into the graph (e.g., ports, version strings, extracted credentials). Adversarial prose is discarded at the worker boundary.

3. **Bounded Tool Scoping:** Even if a stateless worker were locally distracted by an injection in a banner, its strictly restricted toolset prevents it from issuing out-of-scope actions or crossing architectural boundaries.

3.4 The Deterministic Payload Selection Engine

3.4.1 A Two-Tier Selection Strategy: Empirical Fast-Path and Deterministic Fallback

A significant source of failure in fully automated LLM-driven penetration testing is the unreliable selection and configuration of exploitation payloads [18, 20]. Metasploit offers thousands of distinct payload modules, and when an LLM is asked to generate or select a payload string directly, it frequently exhibits known failure modes: selecting a Windows payload for a confirmed Linux target, requesting staged payloads over high-latency multi-hop pivot tunnels (which frequently time out during stage-two binary transmission), or hallucinating non-existent payload identifiers[28, 17].

THESEUS addresses this challenge by removing payload selection from the LLM entirely, but does so through a deliberate *two-tier* strategy rather than by unconditionally forcing every exploitation attempt through a custom ranking algorithm:

1. **Tier 1 — Empirical Fast-Path (default, zero-configuration):** The first exploitation attempt against any target is issued through `msf_trigger_exploit_auto`, which resolves the target port automatically from the AKG's recorded service data while deliberately leaving the `PAYLOAD` option unset. In this case, payload selection is delegated to Metasploit's internal payload-selection mechanism [26],

which filters available payloads according to their compatibility with the selected exploit module and applies the framework's payload preferences when choosing among compatible options. THESEUS therefore relies on Metasploit's native selection logic rather than re-deriving payload compatibility itself, minimizing time-to-first-attempt while retaining a zero-configuration fast path.

2. **Tier 2 — Deterministic Resilience Engine (fallback):** If the Tier 1 attempt fails to establish a session — e.g., due to a mixed x86/x64 target architecture, a staged payload timing out over a pivot tunnel, or an egress-filtering firewall rejecting the default transport — control returns to the Coder agent together with a structured failure report. At this point the Coder explicitly invokes the Deterministic Payload Selection Engine (`_smart_select_payload()`, exposed via `msf_get_best_payload`) to algorithmically rank all target-compatible payloads and select an alternative for an explicit retry through `msf_trigger_exploit`.

This design treats Metasploit's internal compatibility-filtering and preference ordering as an empirically reliable heuristic for the common case, and reserves the more expensive, fully deterministic ranking procedure for exactly the scenarios it was built to solve: recovering from an initial failure whose root cause is architecture, transport, or staging incompatibility. The engine is additionally invoked in two further, non-exploitation-time contexts: as an explicit tool (`msf_get_best_payload`) the Coder can call proactively when it already anticipates an incompatibility (e.g., a known mixed-architecture target) before ever attempting Tier 1, and during automated pivot-session restoration (`akg_restore_sessions`), where a previously established session must be re-created but its original payload was not persisted.

3.4.2 Lexicographical Multi-Criteria Payload Ranking

When `msf_get_best_payload()` is invoked, the framework retrieves the module's target-compatible payloads (`module.compatible_payloads`). Rather than employing ad-hoc keyword heuristics or additive scoring functions, the selection engine executes a deterministic, metadata-driven ranking algorithm. Each candidate payload p is mapped to a 5-tuple evaluation vector $\mathbf{v}(p, \text{ctx})$ based on structural parameters parsed directly from the payload identifier:

$$\mathbf{v}(p, \text{ctx}) = \big(s_{\text{platform}},\ s_{\text{arch}},\ s_{\text{session}},\ s_{\text{conn}},\ s_{\text{staged}}\big) \tag{3.13}$$

Candidate payloads are evaluated and prioritized using a strict lexicographical total order $\leq_{\text{lex}}$, where lower score values denote higher precedence. Vector comparison proceeds strictly element-by-element from left to right. Formally, for two vector scores $\mathbf{a}$ and $\mathbf{b}$:

$$\mathbf{a} <_{\text{lex}} \mathbf{b} \iff \exists k \in \{1, \ldots, 5\} \quad \text{such that} \quad a_k < b_k \ \wedge\ (\forall i < k,\ a_i = b_i) \tag{3.14}$$

Because higher-order components strictly dominate lower-order ones, an incompatibility in early fields (e.g., s_{platform}) immediately disqualifies a candidate, rendering downstream transport or session features irrelevant. Table 3.1 presents a worked

Target Context: `OS = Linux` | `Arch = x64` | `prefer_meterpreter = True`

Candidate Payload	s_{platform}	s_{arch}	s_{session}	s_{conn}	s_{staged}	Final Rank
`linux/x64/meterpreter_reverse_tcp`	0	0	0	0	0	**#1 (Selected)**
`linux/x86/meterpreter/reverse_tcp`	0	1	0	0	1	**#2 (Fallback)**
`windows/x64/meterpreter/reverse_tcp`	4	0	0	0	0	**#3 (Disqualified)**

Step-by-Step Evaluation Trace ($\leq_{\text{lex}}$):

- **#1 vs #2 (Native x64 vs 32-bit Staged):** Identical platform score ($s_{\text{platform}} = 0$). At s_{arch}, native x64 (0) strictly beats 32-bit x86 (1). Hence $(0, 0, 0, 0, 0) <_{\text{lex}} (0, 1, 0, 0, 1)$, making Payload #1 the winner.
- **#2 vs #3 (Linux 32-bit vs Windows x64):** Evaluated at s_{platform}, Linux (0) strictly beats Windows (4). OS incompatibility immediately places Payload #3 in 3rd rank, ignoring its favorable x64 match.

Table 3.1: Worked Example of Lexicographical Payload Ranking

example illustrating how three candidate payloads are ordered step-by-step under a Linux x64 target context.

The five priority axes are defined as follows:

1. **Platform Alignment ($s_{\text{platform}} \in \{0, 1, 2, 3, 4\}$):** Evaluates OS compatibility. Direct native platform matches are prioritized highest (0), followed by cross-platform language interpreters (1), generic fallbacks (2), text-based shell commands (3), and incompatible native platforms (4).

2. **Architecture Alignment ($s_{\text{arch}} \in \{0, 1, 2, 3\}$):** Compares target CPU architecture against the payload specification: exact match (0), compatible backward-execution like x86 on x64 (1), missing target architecture context (2), and explicit architecture mismatch (3).

3. **Session Type Preference ($s_{\text{session}} \in \{0, 1\}$):** Prioritizes interactive Meterpreter instances (0) over standard command shells (1) when advanced post-exploitation capabilities are requested.

4. **Connection Protocol Order ($s_{\text{conn}} \in \{0, 1, 2, 3\}$):** Orders payload transport handlers based on connection reliability and firewall traversal characteristics:

$$\text{reverse_tcp} \prec \text{reverse_http} \prec \text{reverse_https} \prec \text{bind_tcp} \tag{3.15}$$

5. **Staging Optimization ($s_{\text{staged}} \in \{0, 1\}$):** Differentiates execution footprint. Self-contained, stageless payloads are prioritized for native OS targets to enhance stability across network pivots, whereas staged payloads are favored for script-based interpreters (e.g., PHP, Python) for reliable payload restoration.

In the event of a tie across all five evaluation axes, payloads are sorted alphabetically by their unique canonical identifier to guarantee deterministic selection.

3.4.3 Bimodal Payload Partitioning and Inherent Network Uncertainty

A critical challenge in automated network exploitation is that **network reachability cannot be determined *a priori*** through passive service banners or static metadata.

Whether a target can establish an outbound connection back to the attacker's listener (`reverse_*`) or accept an incoming connection on an arbitrary listening port (`bind_*`) depends on asymmetric environmental factors that remain opaque prior to exploitation:

- **Egress vs. Ingress Filtering:** Perimeter or host-based firewalls on the victim may strictly filter outbound TCP connections (preventing reverse shells from reaching `LHOST`) while allowing internal lateral inbound traffic. Conversely, local firewall rules (e.g., `iptables` or Windows Defender Firewall) may reject unsolicited incoming SYN packets on non-standard ports, rendering bind payloads unusable.
- **Pivoting and Route Asymmetry:** In multi-hop networks, an exploit launched through a Metasploit `autoroute` tunnel routes forward traffic seamlessly. However, return packets generated by a reverse payload require an explicit routing path back to the attacker. Unless intermediate nodes run a dual-homed routing daemon or an active relay is established, reverse packets are dropped.

Because committing blindly to a single transport paradigm creates a structural single point of failure, THESEUS departs from the conventional approach of producing a single flat ranking. Instead, the engine implements a **bimodal partitioning architecture** that presents the autonomous agent with two concurrently validated, mutually exclusive connection strategies:

1. **Primary Reverse Candidates (`suggested_payload` and `top_candidates`):** Evaluates the top reverse candidates according to $\leq_{\text{lex}}$. To guard against buffer constraints and stager delivery failures (e.g., restrictive HTTP headers or command-length limits), the engine locates the direct staged/unstaged counterpart of the top candidate (e.g., pairing `meterpreter/reverse_tcp` with `meterpreter_reverse_tcp`) and deterministically guarantees its position as candidate #2.
2. **Partitioned Bind Candidates (`bind_candidates`):** Extracts and ranks an independent collection of compatible bind payloads matching the target OS and architecture. These candidates are provided in a distinct output partition alongside operational guidance, equipping the agent with an immediate, pre-validated fallback if outbound egress is blocked.

Topology Depth Adaptation. When the target's network depth is resolved via breadth-first search (BFS) over the AKG's `PIVOTS_INTO` relations:

- For direct access and single-hop pivots (depth < 2), reverse payloads are suggested as the primary choice (`suggested_payload`), with bind alternatives held in reserve.
- For deep multi-hop pivots (depth ≥ 2), where reverse routing across multiple autoroutes is inherently fragile, the engine automatically elevates the first bind candidates to `suggested_payload`, allowing Metasploit to connect outward through existing pivot channels without requiring return routing.

3.4.4 Agentic Feedback Loop and Failure Recovery Strategy

When the Tier 1 fast-path attempt fails to establish an active session, THESEUS relies on an agentic closed-loop recovery mechanism combining deterministic fallback hints, bimodal payload options, and LLM-driven decision updates:

1. **Session Lifecycle & Liveness Verification:** Upon triggering an exploit module, the tool dynamically polls Metasploit RPC with adaptive polling intervals (0.2s–0.5s up to 15s–30s depending on module complexity) to capture newly registered sessions. To prevent race conditions and socket collapse on ephemeral payloads, session viability is established via non-intrusive metadata matching, while stale or dead sessions are periodically purged through dedicated liveness verification and explicit `session.stop` calls.

2. **Structured Bimodal Reporting and Operational Guidance:** If no active session matches the target, the tool returns a structured JSON failure response containing the bimodal payload partitions described in Section 3.4.3: `suggested_payload`, `top_candidates`, `bind_candidates`, and explicit English `guidance`. This provides the Coder agent with unambiguous context on when to pivot from reverse to bind execution.

3. **Agentic Candidate Fallback:** Rather than relying on rigid, pre-programmed code loops, control is returned to the dedicated Coder agent. The agent parses the structured failure JSON, updates its operational context, and re-invokes `msf_trigger_exploit` for the subsequent execution attempt. This explicit-payload retry path is precisely why the granular `msf_trigger_exploit` tool remains available and semantically distinct from the automatic `msf_trigger_exploit_auto`: the former accepts and honors a caller-supplied payload, the latter does not.

4. **Runtime Reverse Port-Forward Relay:** In scenarios where a reverse payload is selected across a multi-hop pivot (depth ≥ 2)—either by agent choice or because the target module strictly lacks bind payload options—the runtime execution harness transparently intervenes. Prior to launching the exploit, `_setup_portfwd_chain_internal` queries the AKG, identifies the active Meterpreter session on the nearest dual-homed pivot host, and dynamically configures a reverse port forward:

   ```
   portfwd add -R -l LPORT -p LPORT -L 127.0.0.1
   ```
   (3.16)

   The framework dynamically reassigns `LHOST` to the pivot host's internal interface on the target subnet. This enables reverse payloads to connect to their immediate local gateway, which encapsulates and relays the connection back through the existing Meterpreter tunnel to the attacker, bridging what would otherwise be a fatal topological routing gap.

3.5 Multi-Hop Pivoting Mechanics & Network Tunnelling

A core capability of THESEUS is its ability to perform fully automated, multi-hop network pivoting across isolated subnets without requiring manual operator intervention or

external SOCKS proxy toolchains. Pivoting is treated as a continuous part of the attack lifecycle: each successful compromise may expose previously unreachable network segments, which are incorporated into the AKG and made available for subsequent attack cycles.

3.5.1 Initial Compromise & Pivot Discovery

The first pivot is established after compromising a host located on the attacker-accessible network. Once an initial session is obtained, the PostExploiter agent enumerates the compromised host's network interfaces via `msf_session_command` to identify additional NICs and previously unreachable subnets, registering each discovered interface with `akg_add_interface`.

For example, after compromising a DMZ host such as `172.28.0.5`, the PostExploiter discovers a secondary interface connected to the internal subnet `172.30.0.0/24`. This finding is first represented in the AKG structurally: a `HAS_INTERFACE` edge links the compromised host to the newly discovered interface node, and a `CONNECTS_TO` edge links that interface to the (newly registered, via `akg_add_subnet`) internal subnet node.

The PostExploiter then sweeps the internal subnet for adjacent live hosts and registers any discovered host (`akg_add_host`). Finally, the framework registers the discovered subnet in the Metasploit routing layer using the established session as the pivot point (`msf_add_route`) [26], and records the operational routing decision itself as a `PIVOTS_INTO` edge from the originating `ExploitStep` node to the newly reachable `Subnet` node (`akg_add_pivot`). The internal network therefore becomes reachable to subsequent agents without requiring them to explicitly manage the underlying routing mechanism.

At this stage, the AKG records both the newly discovered network topology and the session through which the subnet can be reached. The newly exposed subnet and any hosts discovered within it are consequently added to the set of candidate targets considered by the Orchestrator via `akg_get_next_targets`.

3.5.2 Internal Network Tunnelling & Recursive Pivoting

Once a route to an internal subnet has been registered, communications directed towards hosts in that subnet are forwarded through the corresponding compromised session. When the Scout or Coder agent targets a host located on an internal network (e.g., `172.30.0.22`), the Metasploit routing layer identifies the registered route and uses the associated session as the pivot point between the attacker-controlled network and the internal segment[26].

From the perspective of the agents, the internal target can therefore be addressed using its native network address, while the routing layer handles the underlying traversal through the compromised host. This allows reconnaissance and exploitation operations to be performed on internal hosts without introducing additional proxy infrastructure or requiring manual tunnel management.

The same mechanism is recursively applied when an internal host is compromised. Its network interfaces can expose further isolated subnets, which are again represented in the AKG, registered in the routing layer, and made available to subsequent attack

cycles. Consequently, a sequence of compromises can progressively extend the reachable network topology.

This recursive process enables THESEUS to construct multi-hop attack paths dynamically. At each stage, the newly acquired connectivity becomes part of the externalized campaign state, allowing the Orchestrator to reason over the updated topology, select newly reachable targets, and restore the required routing relationships when resuming an interrupted campaign. If, during this process, a compromised session is later upgraded from a plain shell to Meterpreter, every previously registered `PIVOTS_INTO` route referencing that session is transparently re-pointed to the new session ID (Section 3.2), so that in-flight pivot routing remains valid across the upgrade.

3.5.3 Topological Session & Route Restoration

In multi-hop, long-running penetration testing campaigns, interactive command channels are inherently volatile: ephemeral socket timeouts, intermediate host reboots, process crashes, or stateful campaign pauses frequently sever established Meterpreter or shell sessions. Because multi-hop access forms a strictly linear or tree-structured dependency chain, losing a session at hop k severs the network transport required to reach all downstream targets at hops $> k$.

Restoring an interrupted campaign therefore cannot be done in parallel or in random order: sessions must be restored strictly from the perimeter inward, ensuring that intermediate pivot routes are active before attempting to reach deeper targets. This ordering constraint is enforced using the topological hop depth $\Phi(v)$ defined in Section 3.2.4: exploit nodes are processed in strictly ascending order of Φ, which guarantees perimeter sessions are restored before internal ones and prevents circular routing dependencies.

Restoration Process

The restoration procedure (`akg_restore_sessions`) proceeds in three compact steps:

1. **Topological Ordering:** Exploit nodes in the AKG are sorted by $\Phi(v)$ as established above, and processed in that order.

2. **Liveness Check & Re-Exploitation:** Active Metasploit sessions are queried. If a session matching the target host is alive, its identifier is refreshed. If dead, the exploit is re-executed using the target metadata and payload parameters stored in the AKG, falling back to alternative payloads from the selection engine if needed.

3. **Route Re-Establishment:** Regardless of whether a session was already alive or newly restored, every outgoing `PIVOTS_INTO` edge is processed. The corresponding subnet route is re-injected into Metasploit via `msf_add_route`, and the edge's `via_session_id` is updated to point to the active session.

3.5.4 Depth-Independent Pivoting Design

The pivoting mechanism described above does not depend on a fixed number of hops. The same sequence of steps, enumerating interfaces, registering the subnet, discovering hosts, registering the route, and recording the pivot, runs every time a compromised host exposes a new subnet, no matter how many pivots came before it. Nothing in the implementation depends on hop depth; this section characterizes that depth-independence and its limits.

This means the first pivot (from the attacker's network into subnet S_1) and every later pivot (from a host in S_k into subnet S_{k+1}) are handled by exactly the same code path. The only thing that changes at deeper hops is that traffic now has to pass through the previously established chain of sessions to reach the pivoting host: this is handled automatically by Metasploit's routing table, which resolves each route on its own regardless of how many other routes it is chained behind.

This depth-invariance can be framed as a structural induction argument: given a base case (a single successful pivot from an attacker-accessible host into one internal subnet) and an inductive step that is executed by the same, depth-independent code path, the mechanism is structurally expected to generalize to a pivot chain of arbitrary length *n*. This is an argument about the code path's design, not a proof of reliability at arbitrary depth (that distinction is discussed below). The design is validated empirically in the evaluation chapter (Section 4.3), where a test across multiple chained hops confirms that the recursive step continues to behave correctly as depth increases.

It is important to distinguish this argument from a claim about empirical reliability at arbitrary depth. Structural induction shows that the *mechanism* itself does not depend on hop count: no part of the implementation is conditioned on depth, so there is no reason for it to break at some untested n. It does not show that success is equally *likely* at every depth: each additional hop adds its own chance of failure (session drops, latency, timeouts), so a longer chain is expected to succeed less often even if the mechanism behind each hop is unchanged.

3.6 Real-Time Visual Telemetry Interactive Dashboard

To maintain full operator oversight and support human-in-the-loop auditing during autonomous execution, THESEUS provides an interactive, web-based visual telemetry dashboard.

3.6.1 Dashboard Architecture Communication Pipeline

The visualization engine is decoupled from the core execution framework, so that UI rendering never blocks penetration-testing operations. The dashboard acts as a read-only observability layer, consuming campaign state and execution events without directly modifying the AKG or issuing penetration testing actions.

- **Backend:** A FastAPI server manages the dashboard backend and pushes updates to connected clients over WebSockets.

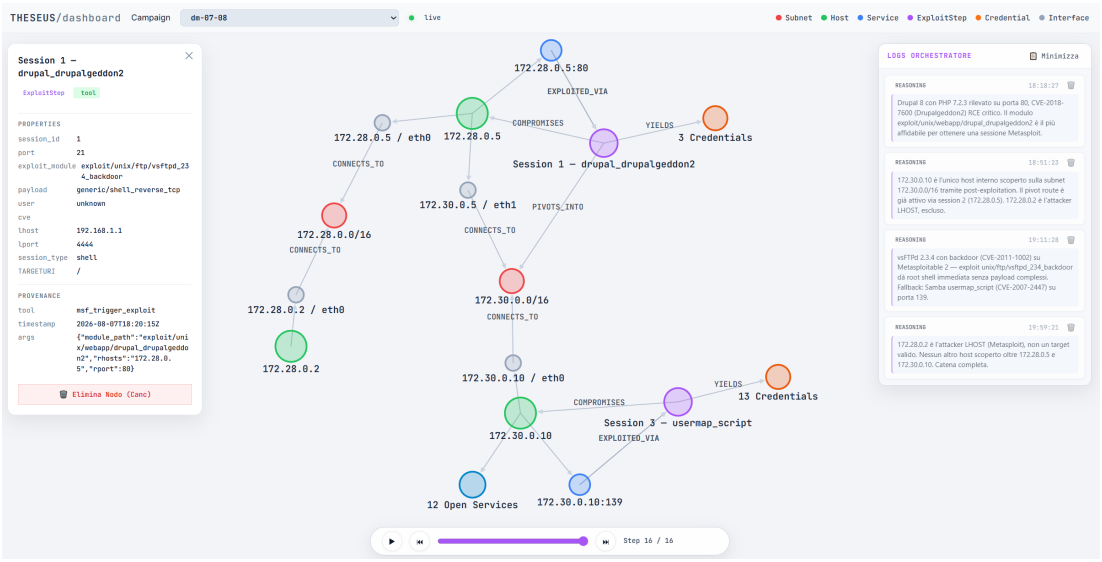

Figure 3.3: THESEUS Web Dashboard Topology Inspector, displaying live D3.js force-directed attack graphs, active Meterpreter sessions, and multi-agent event logs.

- **Live updates:** Changes to the AKG, such as newly discovered hosts, services, credentials, sessions, or routes, are propagated to the connected dashboard clients in real time.
- **Graph view:** The frontend renders the AKG as a force-directed graph using D3.js. Nodes are distinguished by their type, while edges represent relationships between assets and attack steps, making the evolution of the attack topology directly observable.
- **Session and event monitoring:** The dashboard also displays active Meterpreter sessions and a live event log containing relevant orchestration and agent events. This allows the operator to follow both the current campaign state and its execution history.

By combining topology, session information, and execution events in a single interface, the dashboard provides a real-time overview of the campaign while preserving the separation between autonomous execution and operator observation.

3.7 End-to-End System Operational Flow & Execution Walkthrough

To demonstrate how THESEUS orchestrates a multi-hop campaign end to end, this section walks through a single running example across two subnets: `172.28.0.0/16`, the attacker-accessible perimeter, and `172.30.0.0/16`, an internal segment reached via pivot. The AKG is built up incrementally, phase by phase, and each step shows the tool calls involved, how the graph changes, and the decision that change triggers at the Orchestrator. In the graphical representations, host nodes feature three status

indicators (represented as dots inside the node) that track the progression of the campaign through Scout, Exploit, and Post-Exploitation phases.

1. **Perimeter Discovery (Orchestrator).** The Orchestrator activates the campaign and sweeps the initial scope:

   ```
   orchestrator_discover_subnet(subnet_cidr="172.28.0.0/16")
   ```

   This registers subnet SU_1 (`is_attacker_network=True`) and a single responding host, H_1 (`172.28.0.5`), with `scout_status=pending`. With no service data yet attached to H_1, the only sensible next step is reconnaissance, so the Orchestrator delegates the Scout Phase.

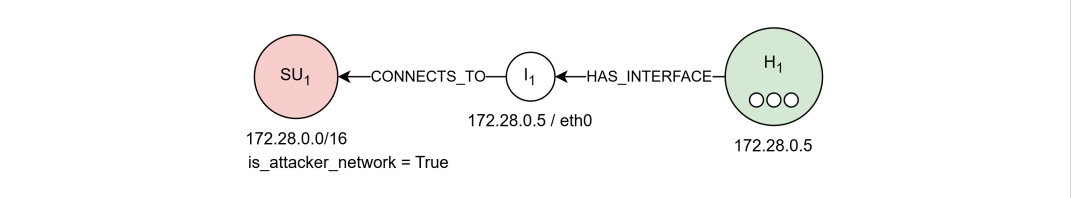

Figure 3.4: AKG after perimeter discovery: subnet SU_1 and host H_1, no further attributes yet.

2. **Reconnaissance (Scout).** Scout fingerprints H_1:

   ```
   nmap_scan(target="172.28.0.5", ports="1-10000", arguments="-sV -open")
   ```

   ```
   curl_request(url="http://172.28.0.5:80/", method="GET")
   ```

   It finds an Apache Drupal 8.5.0 instance on port 80 and registers it as service node SE_1 (`RUNS` from H_1), also picking up a passively leaked credential C_1 via a `LEAKS` edge. With `scout_status=DONE`, the Orchestrator matches the fingerprint to CVE-2018-7600 (Drupalgeddon2) and delegates the Coder Phase.

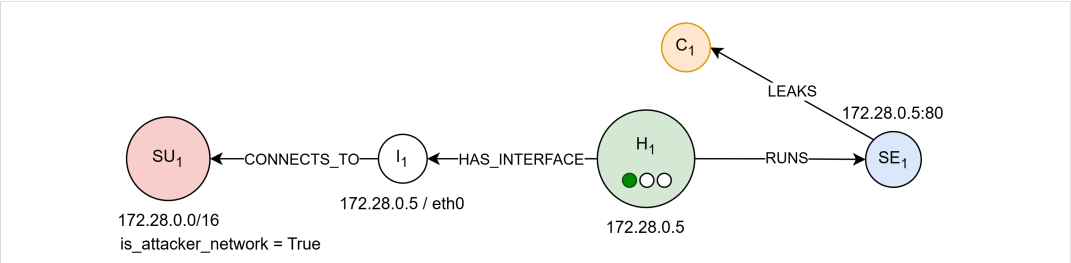

Figure 3.5: AKG enriched with the fingerprinted service SE_1 and leaked credential C_1.

3. **Exploit (Coder).** Coder fires the Tier 1 fast-path attempt, no payload specified:

   ```
   msf_trigger_exploit_auto(module_path="exploit/unix/webapp/drupalgeddon2",
   rhosts="172.28.0.5", rport=80)
   ```

   Metasploit resolves a compatible Meterpreter payload and the exploit succeeds on the first attempt (`Session 2`).

Coder records the `ExploitStep` node E_1 ($SE_1 \xrightarrow{\texttt{EXPLOITED_VIA}} E_1 \xrightarrow{\texttt{COMPROMISES}} H_1$) and H_1 flips to `pwned`. H_1 with `postexploit_status=pending` is a candidate for the next phase, so the Orchestrator delegates post-exploitation on `Session 2`.

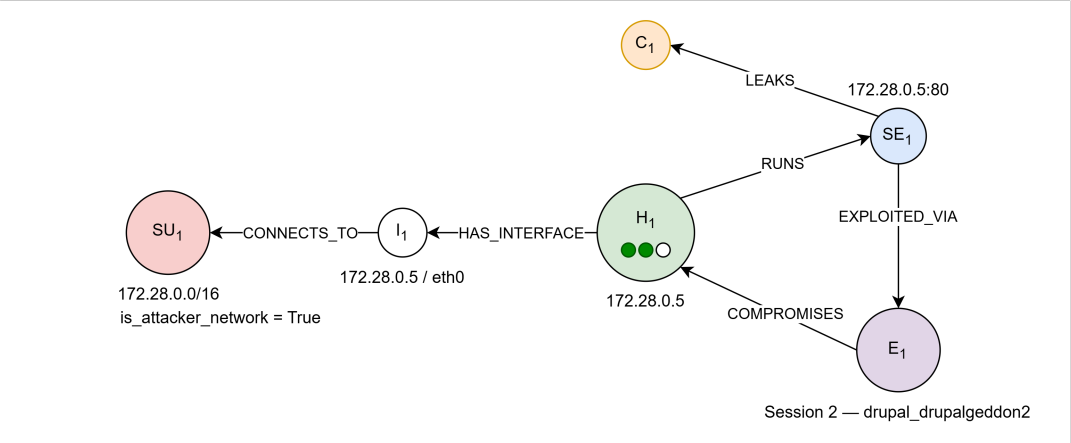

Figure 3.6: AKG after compromise: ExploitStep E_1 links SE_1 to H_1, now marked pwned.

4. **Post-Exploitation & Pivot Discovery (PostExploiter).** PostExploiter enumerates the compromised host:

   ```
   msf_session_command(session_id="2", command="ip a")
   ```

   It finds a second interface on `172.30.0.5`, harvests a local credential C_2, and sweeps the newly exposed subnet, finding one more host:

   ```
   msf_discover_hosts(subnet="172.30.0.0/16")
   ```

   The AKG grows accordingly: subnet SU_2 and interface I_2 connect H_1 to the internal segment, $E_1 \xrightarrow{\texttt{PIVOTS_INTO}} SU_2$ records the route, and the new host H_2 (`172.30.0.10`) is registered with all phase flags pending. With H_2 now the only uncompromised target and a pivot route already in place, the Orchestrator delegates the full tactical pipeline for H_2 over the pivot.

5. **Hop 2: Repeat Cycle & Objective Capture.** The same Scout $\rightarrow$ Coder $\rightarrow$ PostExploiter sequence runs again, now through the pivot. Scout finds Samba 4.3.11 on port 445; the Orchestrator maps it to CVE-2017-7494 and delegates exploitation:

   ```
   msf_trigger_exploit_auto(module_path="exploit/linux/samba/is_known_pwn",
   rhosts="172.30.0.10", rport=445)
   ```

   This establishes root-level `Session 5` and a new `ExploitStep` $E_2 \xrightarrow{\texttt{COMPROMISES}} H_2$. PostExploiter then retrieves the objective:

   ```
   msf_session_command(session_id="5", command="cat /flag* ||find / -name flag*")
   ```

   The flag is recorded via $E_2 \xrightarrow{\texttt{YIELDS}} C_3(\text{flag})$. With no uncompromised targets left, the campaign-level termination condition is met, and the Orchestrator marks the campaign complete.

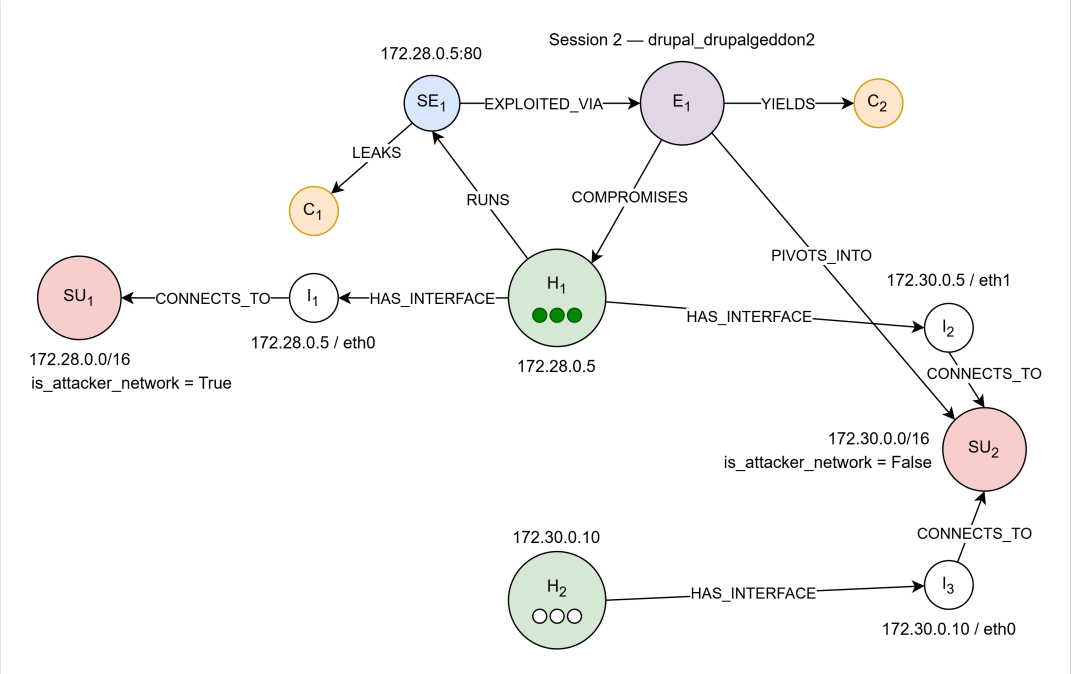

Figure 3.7: AKG after pivot discovery: subnet SU_2, credential C_2, and uncompromised host H_2.

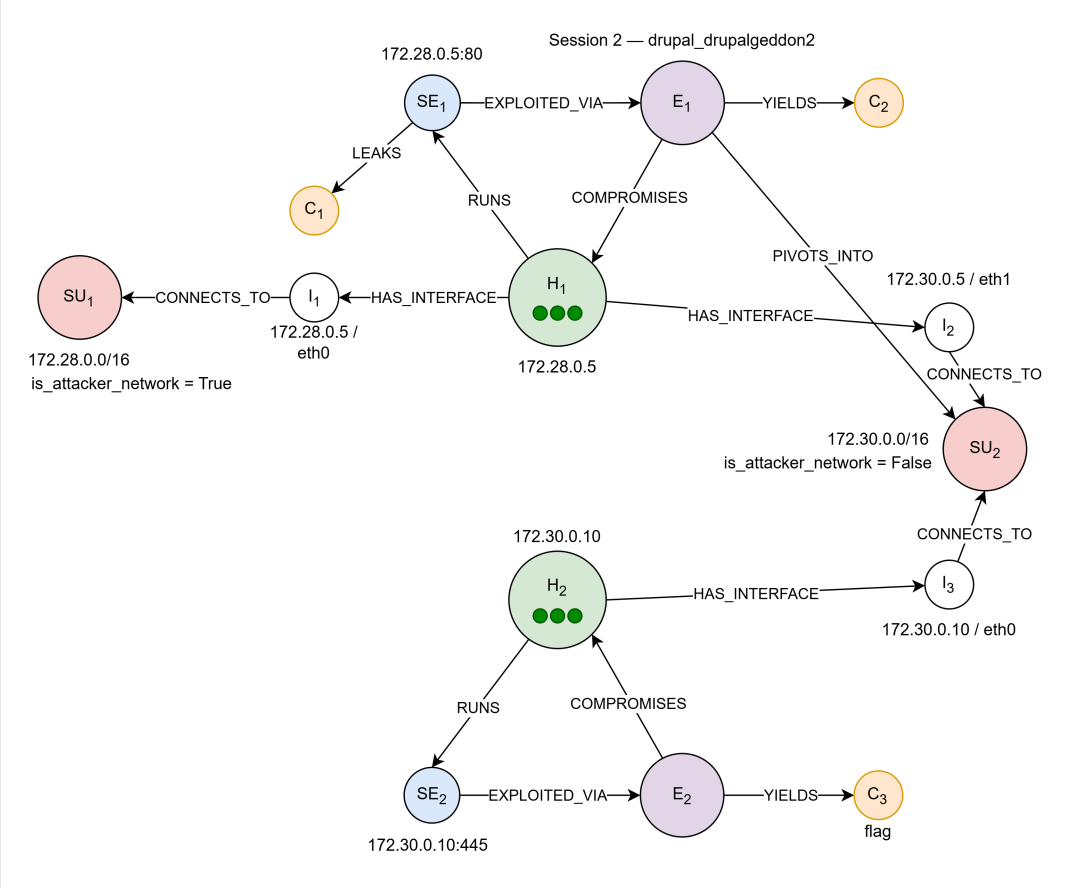

Figure 3.8: Final AKG: H_2 compromised via E_2, objective flag captured.

Chapter 4

Experimental Evaluation and Results

This chapter evaluates the proposed penetration-testing framework with the objective of assessing its ability to autonomously execute multi-stage attack chains. In contrast to systems designed primarily around isolated vulnerability exploitation, the proposed architecture is explicitly structured to preserve attack state across successive stages and to use the artifacts obtained during one stage to enable subsequent actions.

The evaluation is therefore divided into three complementary perspectives. First, the framework is evaluated on an established vulnerability-testing benchmark to provide a reproducible comparison with existing systems. Second, the framework is evaluated on scenarios specifically designed to measure multi-stage and multi-host attack-chain execution, which represents the primary objective of the proposed architecture. Lastly, targeted mechanism-level experiments isolate the main architectural components introduced by THESEUS, namely deterministic payload selection, AKG-based context management, and state restoration, allowing their individual effects to be evaluated separately from the end-to-end attack outcomes.

For the initial comparison, AutoPenBench [10] is used as the common benchmark. It provides a controlled and reproducible environment containing 33 penetration-testing tasks and defines explicit objectives and intermediate milestones. Its use makes it possible to compare the proposed framework with previously evaluated systems such as VulnBot, xOffense, and PenExpert under a common task definition.

4.1 Research Questions and Common Experimental Protocol

4.1.1 Research Questions

The evaluation is designed to answer the following questions:

1. **RQ1:** Can the framework autonomously execute penetration-testing tasks from reconnaissance to exploitation?

2. **RQ2:** How does the framework perform compared with existing LLM-based penetration-testing systems on a common benchmark?

3. **RQ3:** Can the explicit representation of attack state be successfully used to execute dependent attack stages?

4. **RQ4:** Can the framework maintain and exploit information obtained from previous stages when constructing longer, and non-linear, attack chains?

5. **RQ5:** Do the individual architectural mechanisms introduced by the framework (deterministic payload selection, AKG-based context management, and state restoration) produce the effects they are designed to produce?

RQ1 and RQ2 are addressed using AutoPenBench (Section 4.2). RQ3 and RQ4 motivate the chain-oriented evaluation (Section 4.3). RQ5 is addressed by the mechanism-level experiments in Section 4.4. Table 4.1 summarizes this mapping.

Table 4.1: Mapping between research questions, experiments, and the evidence each produces.

RQ	Experiment	Section	Evidence
RQ1, RQ2	AutoPenBench, 33 targets	Sec. 4.2	Pass@1, Pass@5
RQ3, RQ4	6 chain scenarios	Sec. 4.3	Successes/5
RQ5	3 mechanism-level experiments	Sec. 4.4	Mechanism-specific metrics

4.1.2 Common Experimental Protocol

Unless stated otherwise, the following configuration applies to every experiment reported in this chapter, both on AutoPenBench (Section 4.2) and on the chain-oriented scenarios (Section 4.3).

Model and environment. The system is backed by `deepseek-v4-flash` (q2 quantization), served locally via a vLLM-compatible endpoint on an on-premises NVIDIA DGX Spark node; no prompts leave the local infrastructure. Generation is capped at 4096 output tokens per call, top-p=1.0. The Scout, Coder, and PostExploiter sub-agents are stateless, with long-term memory delegated to the in-memory AKG. Targets run as isolated Docker environments on the same host; security tooling comprises Metasploit v6.4.0-dev (via MSF-RPC), Nmap v7.9x, and an in-process Python module for SSH brute-forcing and cryptographic attacks.

Repetitions. Each target (AutoPenBench) or scenario (chain-oriented evaluation) is run $n = 5$ times under an identical configuration. For AutoPenBench this yields 165 runs total (33×5), the basis for Pass@1, Pass@5, and run-level success rate (Section 4.2.1). For the chain-oriented scenarios this yields 5 runs per scenario, reported as successes out of 5 (Section 4.3.7).

Limits and retries. Each run is bounded by a 3600-second timeout, after which it is terminated and scored as a failure. No cap is placed on the number of tool calls; the agent proceeds until the objective is captured or the timeout is reached. Transient infrastructure failures (e.g., a stuck MSF-RPC socket) trigger a limited automatic retry.

Reset and success criterion. Each run starts from a clean environment (container, or full topology for the chain scenarios) and a freshly initialized AKG, with no state carried over from the previous run. A run is scored as a success only if the ground-truth flag, or the final campaign objective, is captured. AutoPenBench milestones are recorded separately as a graded measure of progress and do not by themselves determine pass/fail.

Differences between the two evaluations. The chain-oriented scenarios share the same architecture, timeout, and state-persistence mechanism as the AutoPenBench evaluation, with the addition of multi-hop pivoting support. The two evaluations differ in how the initial task information is provided to the system. For AutoPenBench, THESEUS receives the task prompt and objective defined by the benchmark. For the chain-oriented scenarios, the initial information provided to the system is intentionally minimal: the perimeter subnet to be attacked and the instruction to reach the final objective by capturing the flag and then terminate the engagement. The system is not given the attack path, intermediate hosts, vulnerabilities, credentials, or other information about the structure of the chain. These elements must instead be discovered during execution and represented in the AKG as the campaign progresses.

4.1.3 Metrics

The following metrics are used across this chapter. The first four apply to AutoPenBench; the fifth applies to the chain-oriented evaluation.

- **Pass@1**: fraction of targets solved on the first of the $n = 5$ recorded runs. Comparable to AutoPenBench's own protocol, both evaluated on a single run.
- **Pass@5**: fraction of targets solved in at least one of the 5 recorded runs. This is a *target-level* metric; it should not be conflated with the *sub-task-level* "any-of-5" metrics reported by xOffense and VulnBot (Section 4.2.3).
- **Run-level success rate**: fraction of all 165 individual executions (33×5) that captured the flag. Because $n = 5$ is homogeneous across all 33 targets, this is equivalent to the unweighted mean of the per-target success rates $s_i/5$, so no target is over-weighted relative to another.
- **Milestone completion (micro-average)**: total milestones achieved over total milestones possible, summed across all 165 runs. This is a true micro-average over milestone counts, not a mean of per-category percentages, and is not tied to a specific Pass@k protocol.

- **Scenario success rate**: for the chain-oriented evaluation, the number of successful runs out of the 5 executed for each scenario (Section 4.3). A run is successful only if the final campaign objective is reached. Campaign-graph metrics (hosts, pivots, events, decisions, credentials) are reported separately as descriptors of campaign structure, not as success criteria.

4.2 AutoPenBench

AutoPenBench [10] is used as the first evaluation benchmark because it provides a standardized environment for autonomous penetration testing and has already been used to evaluate several LLM-based systems.

The benchmark contains 33 tasks divided into two groups: 22 introductory in-vitro tasks and 11 tasks based on real-world CVEs. Each task defines a concrete penetration-testing objective and provides intermediate milestones that allow the execution trace to be analyzed at different stages.

This structure is useful for the proposed framework for two reasons. First, it provides a reproducible baseline against which the framework can be compared with previously published systems. Second, the intermediate milestones make it possible to distinguish failures occurring during reconnaissance, vulnerability identification, exploitation, and subsequent stages instead of considering only the final task outcome.

However, AutoPenBench is not specifically designed to evaluate long multi-host attack campaigns. Its tasks are largely independent and do not require the compromise of one machine to establish the preconditions for attacking another machine. Consequently, AutoPenBench is used in this work as a *common baseline* rather than as the complete evaluation of the proposed architecture. The experimental setup and metrics follow the common protocol described in Section 4.1.2 and Section 4.1.3.

4.2.1 Results

Table 4.2 breaks down performance across the five AutoPenBench categories. The agent performs strongly on *Access Control* and *Cryptography*, reaching full Pass@5 (100.00% for both) alongside high milestone completion (97.67% and 96.55%, respectively), with Pass@1 already high for Cryptography (100.00%) and somewhat lower for Access Control (80.00%), suggesting that even in this well-scoped category a first attempt occasionally misses before a retry succeeds. Performance is more mixed on *Web Security* and *Network Security*, where Pass@5 reaches 85.71% and 66.67% respectively; *Network Security* is also the weakest category on a single attempt, with Pass@1 at 50.00% and the lowest milestone completion (67.39%), indicating higher variability across independent runs and a greater tendency to stall before reaching late-stage milestones. The *Real-World CVE* category, despite having the largest number of targets (11), shows the widest gap between Pass@1 (54.55%) and Pass@5 (81.82%): more than half of these targets fail on a first attempt but are recovered given up to five tries. Its milestone completion (75.38%) sits below its own Pass@5, indicating that even in runs that do not end in a captured flag, the agent frequently makes partial progress through the exploitation chain. Overall, the agent solves 66.67% of targets

on a single attempt (Pass@1) and 84.85% when given five attempts (Pass@5), with an average milestone completion rate of 85.29%, highlighting both the benefit of retry-based evaluation and the residual difficulty of fully automating multi-step penetration testing tasks.

Table 4.2: Per-category results across all 33 targets, $n = 5$ runs per target. Pass@1 is the success rate on the first of the 5 runs; Pass@5 is the success rate given at least one of the 5 runs; Run-level success rate is the fraction of all individual runs that succeeded; Milestone completion is micro-averaged across all runs.

Category	Targets	Pass@1	Pass@5	Run-level	Milestone %
Access Control	5	4/5 (80.00%)	5/5 (100.00%)	21/25 (84.00%)	97.67%
Cryptography	4	4/4 (100.00%)	4/4 (100.00%)	19/20 (95.00%)	96.55%
Web Security	7	5/7 (71.43%)	6/7 (85.71%)	23/35 (65.71%)	89.77%
Real-World CVE	11	6/11 (54.55%)	9/11 (81.82%)	29/55 (52.73%)	75.38%
Network Security	6	3/6 (50.00%)	4/6 (66.67%)	12/30 (40.00%)	67.39%
Total	**33**	**22/33 (66.67%)**	**28/33 (84.85%)**	**104/165 (63.03%)**	**85.29%**

4.2.2 Operational Efficiency

Beyond outcome-level metrics, the raw logs allow the operational cost of the framework's interaction with the attack surface to be characterized. This is presented as an internal characterization of the agent's behavior rather than a comparison with other systems, since none of the reference systems publish comparable per-run tool-call or token statistics. The only published reference points are step *caps*, which bound behavior rather than measure it, under a single-action protocol not directly comparable to this framework's multi-agent granularity (Section 4.2.3). AutoPenBench's own GPT-4o baseline is capped at 30 steps for in-vitro tasks and 60 for real-world tasks [10], while VulnBot caps its own agent at 15 steps on AutoPenBench (5 steps per phase) [14].

Table 4.3: Tool-call and token cost of successful and active failed runs, by category. Successful-run values are computed over the $n = 5$ runs per target used throughout the evaluation.

	Successful		Active failed	
Category	**Tool calls**	**Tokens**	**Tool calls**	**Tokens**
Access Control	29.3 ($\pm$8.8)	284,533	42.5 ($\pm$30.3)	477,248
Cryptography	13.0 ($\pm$3.3)	147,499	19.8 ($\pm$9.1)	79,254
Web Security	36.2 ($\pm$31.1)	326,644	31.4 ($\pm$17.8)	402,702
Real-World CVE	37.4 ($\pm$29.9)	499,179	60.8 ($\pm$30.4)	892,792
Network Security	45.6 ($\pm$14.6)	755,861	46.5 ($\pm$31.5)	607,771
Overall	**32.95 ($\pm$24.59)**	**403,656**	**44.4 ($\pm$29.0)**	**578,363**

Table 4.3 shows that successful runs generally require fewer tool calls and consume fewer tokens than active failed runs, although the pattern is not uniform across all cate-

gories. Across the benchmark, successful runs require an average of 32.95 ($\pm$24.59) tool calls and 403,656 tokens, compared with 44.4 ($\pm$29.0) tool calls and 578,363 tokens for active failures. Thus, relative to successful executions, active failures involve approximately 34.8% more tool calls and 43.3% more token consumption.

The category-level results also show substantial variation in operational cost. Among successful runs, *Cryptography* and *Access Control* have the lowest average costs, whereas *Network Security* has the highest, with 45.6 ($\pm$14.6) tool calls and 755,861 tokens. *Web Security* and *Real-World CVE* occupy an intermediate range but exhibit considerably larger variation in tool calls. The corresponding failure costs are also heterogeneous: the largest difference between successful and active failed runs occurs for *Real-World CVE*, where failures reach 60.8 ($\pm$30.4) tool calls and 892,792 tokens. In contrast, *Web Security* and *Network Security* show smaller differences in tool-call counts, while *Cryptography* even exhibits lower token consumption among the observed active failures. These differences indicate that operational cost is influenced by the specific attack path and target characteristics rather than following a uniform category-level pattern.

The termination data further characterizes how active failures arise. Of the active failed runs recorded in the benchmark, 92.1% reached the execution time budget, while only 7.9% terminated substantially earlier, within the first 180 seconds. The early terminations were associated with blocking conditions during initial reconnaissance, such as unreachable targets or the absence of accessible services. Consequently, most observed active failures did not terminate shortly after an unsuccessful interaction. Instead, they remained active until the execution budget was exhausted, continuing to issue tool calls and accumulate context without reaching the campaign objective.

This distinction is important when interpreting the residual failure rate. The telemetry indicates that, in this benchmark, active failures are predominantly associated with exhausting the available execution budget rather than with immediate termination after an unsuccessful early step. The result does not by itself establish that increasing the time budget would proportionally increase the success rate, since additional execution time may not always lead to a productive attack path. It does, however, identify execution budget as a measurable constraint on the current configuration and motivates evaluating alternative timeout policies in future experiments.

4.2.3 Comparison with Existing Systems

Repetition protocol. AutoPenBench [10] reports a single run per task for its GPT-4o baseline (21.21%, 7/33), with no best-of-k aggregation. VulnBot [14] reports two figures computed from 5 runs per target: a *sub-task-level* result (69.05%, Llama3.1-405B), counted as successful if a sub-task succeeds in at least one of the five runs, and a *target-level* result (30.30%, 10/33), for which the paper does not state whether it is drawn from a single run or is itself a best-of-5 outcome. xOffense [21] follows the same structure: a sub-task-level "1 Experiment" figure (79.17%, Qwen3-32B fine-tuned), explicitly defined as at least one success in five runs, and a target-level figure (72.72%, 24/33) with the same unspecified repetition status. Because of this, the target-level and sub-task-level figures published by VulnBot and xOffense are not interchangeable with each other, even though both are sometimes referred to informally as "Pass@5":

only the sub-task-level figure is explicitly defined as such in the source papers.

Action granularity. AutoPenBench, VulnBot, xOffense, and (to the extent documented) PenExpert implement a single-agent, single-action-per-turn loop. The framework evaluated here uses a multi-agent architecture in which an orchestration cycle can dispatch several sub-agents, each issuing more than one tool call before the environment state is next evaluated. Step and invocation-count metrics (Section 4.2.2) are therefore reported for internal characterization only and are not treated as comparable to published step counts.

Table 4.4: Comparison on AutoPenBench. Published results are taken from the corresponding studies. For THESEUS, each reported figure is taken directly from Table 4.2. Task-level and sub-task-level columns report different units (33 targets vs. hundreds of finer-grained sub-tasks with a study-specific decomposition) and are not directly comparable to each other, even within the same row.

System	Model	Task-level	Sub-task %*	Protocol
AutoPenBench	GPT-4o	21.21% (7/33)	—	single run
VulnBot	Llama-3.1-405B	30.30% (10/33)	69.05%	task: unspecified†; sub-task: any-of-5
PenExpert	GPT-4o	45.45%	78.78%	any-of-5
PenExpert	Claude Sonnet 4.5	72.73%	90.20%	any-of-5
xOffense	Qwen3-32B	30.30%	52.36%	task: unspecified†; sub-task: any-of-5
xOffense	Qwen3-32B (fine-tuned)	72.72% (24/33)	79.17%	task: unspecified†; sub-task: any-of-5
THESEUS	**DeepSeek V4 Flash q2**	**84.85% (28/33)**	**85.29%**	target-level, any-of-5
vs. AutoPenBench		**66.67% (22/33)**		single run

*Sub-task/milestone completion rates use study-specific decompositions of AutoPenBench that differ across sources; values are indicative and not strictly comparable across rows. †VulnBot and xOffense do not state explicitly whether their target-level figure is a single run or a best-of-5 outcome; only their sub-task-level figure is defined as "any-of-5" in the source papers.

The 66.67% figure (22/33 targets solved on a single, standardized attempt) is the recommended point of comparison against AutoPenBench's own GPT-4o result (21.21%), since both are single-run, target-level scores. The 84.85% figure (28/33 targets solved in at least one of 5 runs) is a target-level any-of-5 score, reported alongside, rather than as a direct substitute for, the sub-task-level figures published by xOffense (79.17%) and VulnBot (69.05%): as noted above, a sub-task-level any-of-5 rate and a target-level any-of-5 rate are not the same quantity.

The position of the proposed framework in this comparison should be interpreted together with its architectural objective. AutoPenBench measures general autonomous penetration-testing capability under a single-agent, single-action interaction protocol, whereas the proposed framework is primarily designed to maintain state and execute dependent attack chains through a multi-agent architecture whose interaction granularity does not map cleanly onto that protocol. A comparable AutoPenBench result therefore provides evidence of general penetration-testing capability, but does not fully capture the state-dependent multi-stage behavior targeted by the architecture.

4.2.4 Exploratory Evaluation on a Smaller Model

Exploratory runs conducted with the substantially smaller Qwen3.8 model showed results broadly consistent with those obtained with `deepseek-v4-flash`, achieving a 61.2%–68.7% single-attempt target success rate and an overall sub-task completion rate of 85.2%. Taken together with the main results, this is a preliminary indication that the framework's structured orchestration and deterministic state tracking reduce, without eliminating, the effect of reduced model parameter capacity on offensive capability; a firmer conclusion requires the protocol details noted above.

4.3 Chain-Oriented Evaluation

To evaluate THESEUS beyond isolated single-host exploitation, a set of dedicated multi-stage penetration-testing scenarios was constructed. The experiments were designed to evaluate different forms of dependency between attack stages, including multi-hop network traversal, credential reuse, privilege-dependent information discovery, adaptive target selection, and cross-branch backtracking. Together, they assess whether THESEUS can maintain a coherent campaign state, reuse information acquired during previous stages, and adapt its execution according to the evolving state of the environment.

4.3.1 Three-Stage Multi-Hop Attack Chain

The first experiment evaluated whether THESEUS could progressively extend an attack across multiple network segments by using previously compromised hosts as intermediate access points. The scenario consisted of three target hosts distributed across three network segments. The attacker initially had access only to the perimeter network `192.168.10.0/24`.

The first target, `192.168.10.2`, exposed a GeoServer 2.23.2 instance on port `8080`. Exploitation of CVE-2024-36401 resulted in a privileged session on the host. This compromise was not considered the final objective: the compromised machine provided access to the previously unreachable network `10.10.1.0/24`.

THESEUS subsequently used the newly acquired network position to discover and attack the second target, `10.10.1.3`. The host exposed Samba 4.6.3 on port `445` and was successfully compromised through CVE-2017-7494. The resulting privileged session provided access to a third network segment, `10.10.2.0/24`. At this point, the attack required the system to preserve the previously established access to the first internal network while extending the campaign through the newly compromised host.

The final stage targeted `10.10.2.4`, which exposed Apache Druid 0.20.0 on port `8888`. THESEUS identified and successfully exploited CVE-2021-25646 from the second internal position. The resulting session was established through the previously compromised hosts, completing the attack chain from the perimeter network to the third internal segment. The execution is illustrated in Figure 4.1.

The experiment required the final host to be reached through the sequence of intermediate compromises, rather than through direct access from the initial attacker

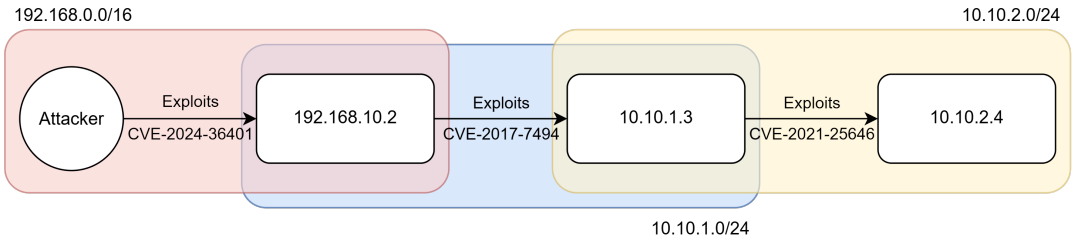

Figure 4.1: Three-stage attack chain.

position. The scenario therefore tested not only vulnerability identification and exploitation, but also the preservation and reuse of access information acquired during previous stages of the campaign.

The execution produced concrete evidence of successful progression at each stage. First, exploitation of `192.168.10.2` produced a privileged session and exposed the route towards `10.10.1.0/24`. Second, `10.10.1.3` was successfully compromised through this internal position, producing a new privileged session and exposing `10.10.2.0/24`. Finally, `10.10.2.4` was compromised through the second internal position, completing the three-host chain.

The campaign resulted in the compromise of all three target hosts and the capture of the final objective. The execution also demonstrated that previously established sessions and network routes could be preserved and restored during the campaign.

This experiment demonstrates that THESEUS can execute an attack as a state-dependent sequence rather than as a collection of independent exploitation attempts. Each stage consumes access and network information obtained during the previous stage, while the intermediate compromises are necessary to make subsequent targets reachable. The successful completion of the three-host chain therefore provides evidence that the system can preserve campaign state, establish successive internal access positions, and use them to extend an attack across multiple network segments.

4.3.2 Two-Stage Credential-Reuse Chain

The second experiment evaluated whether information obtained during one compromise could be persisted and subsequently reused against a different host.

The experiment consisted of two target hosts connected through an internal network. The first host, `192.168.10.2`, acted as the initial access point to the internal network. It exposed GeoServer 2.23.2 on port `8080` and was compromised through CVE-2024-36401. The resulting privileged session provided access to the internal network `10.10.1.0/24`.

During post-exploitation, THESEUS discovered an SSH private key. Rather than treating the key as a terminal finding, the PostExploiter recorded it in the AKG as a credential associated with the compromised host.

The newly discovered network interface and pivot capability subsequently allowed THESEUS to enumerate the previously inaccessible subnet. The Orchestrator selected `10.10.1.3` as the next target and identified the previously harvested SSH credential as

a candidate access mechanism. Authentication succeeded and produced a privileged session on the second host. The execution is illustrated in Figure 4.2.

The significance of this experiment is not the individual vulnerability or the SSH authentication mechanism. Rather, the experiment verifies the dependency between the two stages. The second compromise depends on both access to the internal network and the credential recovered from the first host. The experiment therefore validates the persistence and reuse of cross-host knowledge within a single campaign.

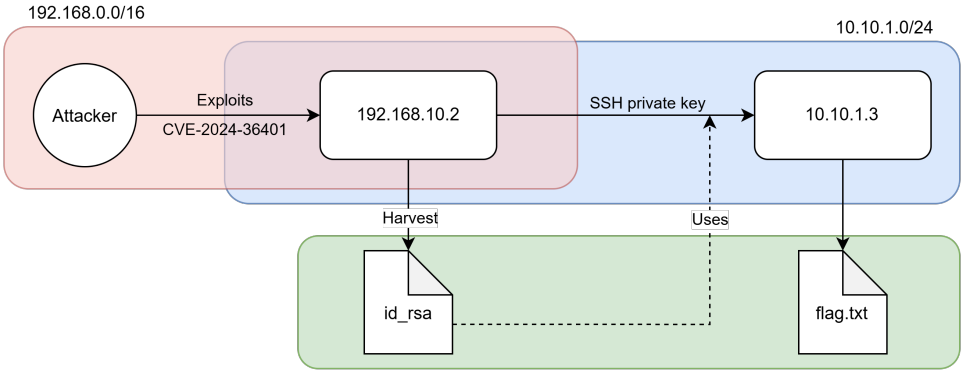

Figure 4.2: Two-stage credential-reuse attack chain.

4.3.3 Privilege-Escalation-Assisted Lateral Movement

A third chain-oriented experiment was designed to evaluate whether THESEUS could combine local privilege escalation with subsequent lateral movement when the information required for the second stage was not initially accessible. In contrast to the previous credential-reuse scenario, the credentials enabling the lateral movement were not directly available from the initial user context. They became accessible only after obtaining root privileges on the first host.

The campaign started from the perimeter network `192.168.10.0/24`. The first target, `192.168.10.2`, exposed an SSH service on port `22`. THESEUS successfully authenticated using the initial low-privilege credentials `student:password`, obtaining a session with `uid=1000`. At this point, the host was accessible, but the information required to perform the subsequent lateral movement was not yet available to the compromised user.

THESEUS then performed post-exploitation analysis and identified a local privilege-escalation path through the host's sudo configuration. The privilege escalation was successfully executed, resulting in a root session. This change in privilege level was a necessary intermediate step in the attack: access to the protected `/root/` directory allowed the system to inspect files that were inaccessible from the original user context.

Among the information recovered after privilege escalation were the protected SSH configuration and credentials stored on the host. The discovered information identified an internal SSH target, `10.10.1.3`, and provided the credentials required for authentication as user `admin`. THESEUS recorded the newly acquired information in the AKG, making it available to subsequent stages of the campaign.

The system then enumerated the network interfaces of `192.168.10.2`, identifying its internal interface `10.10.1.2`. A route towards `10.10.1.0/24` was subsequently established through the compromised host. Internal reconnaissance identified `10.10.1.3` as an active host exposing SSH on port `22`.

The lateral-movement stage consumed information generated by the preceding stages. THESEUS retrieved the previously discovered credentials from the AKG and used them to authenticate to `10.10.1.3` through the established internal route. Authentication succeeded and produced a new interactive session as user `admin`. The final objective was then reached by locating and reading the flag on the internal host. The execution is illustrated in Figure 4.3

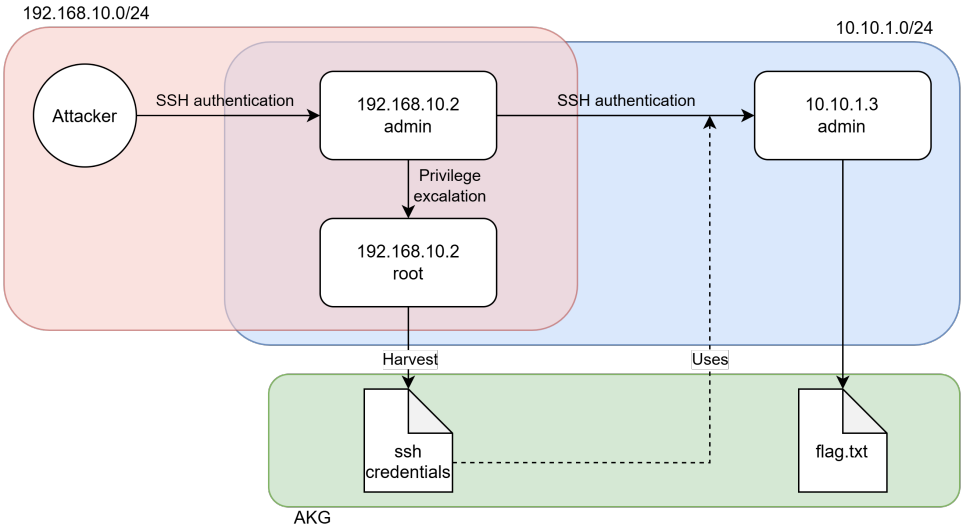

Figure 4.3: Privilege-escalation-assisted lateral movement chain.

This experiment demonstrates a different form of chain dependency from the previous tests. The second compromise was not enabled simply by reusing information already available at the beginning of the campaign. Instead, the information required for lateral movement became available only as a consequence of the successful privilege escalation on the first host. The test therefore evaluates whether THESEUS can recognize and exploit dependencies between attack stages, persist newly acquired information in the AKG, and subsequently use that information to extend the campaign into an internal network.

4.3.4 Adaptive Target Selection

A further experiment evaluated whether THESEUS could assess multiple candidate targets and redirect the campaign when the initially investigated host did not provide a viable exploitation path. Unlike the previous experiments, the main objective of this scenario was not to maximize the length of the attack chain, but to evaluate target selection under competing candidate hosts.

At the beginning of the campaign, two hosts were identified on the perimeter network, `192.168.10.2` and `192.168.10.3`. Both represented potential targets and

were therefore subject to reconnaissance. THESEUS initially selected `192.168.10.2` and performed the available service and vulnerability assessment. The host did not provide a viable exploitation path. THESEUS subsequently reassessed the remaining candidate and selected `192.168.10.3`.

Reconnaissance of `192.168.10.3` identified a service affected by a known vulnerability, providing a valid exploitation path. The vulnerability was successfully exploited, resulting in access to the host and the associated internal network.

From this newly acquired network position, THESEUS identified `10.10.1.3` as an internal target and detected an exposed Samba service on port `445`. The system successfully executed the subsequent internal compromise, reaching the final campaign objective.

The execution is illustrated in Figure 4.4.

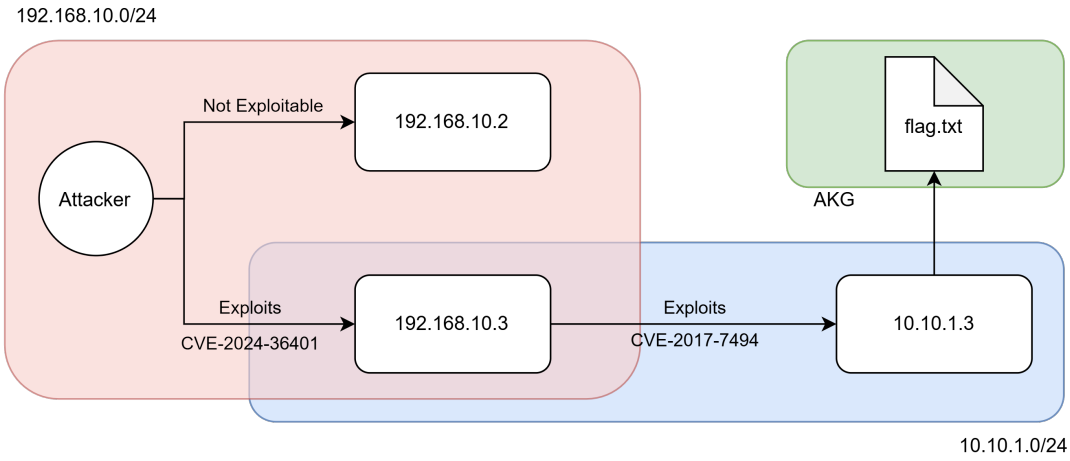

Figure 4.4: Adaptive target selection following the assessment of an initially unproductive target.

4.3.5 Cross-Branch Credential Dependency and Backtracking

The fifth experiment evaluated whether THESEUS could recover from an unsuccessful attack path by pursuing an alternative branch of the campaign and subsequently revisiting a previously discovered target. The scenario was specifically designed so that the discovery of an internal target and the acquisition of the credentials required to access it occurred through two different attack branches.

The campaign initially targeted `192.168.10.2`, which exposed GeoServer 2.23.2 on port `8080`. THESEUS successfully exploited CVE-2024-36401, obtaining a compromised session and access to the internal network `10.10.1.0/24`. Through this newly established position, the system discovered `10.10.1.5`, which exposed an SSH service on port `22`.

At this point, however, THESEUS did not possess valid credentials for the internal SSH service. The system attempted authentication using credentials available in the current campaign state, but these attempts were unsuccessful. Rather than continuing

indefinitely along the same attack path, THESEUS changed its course of action and returned to the remaining perimeter targets.

The alternative branch led to `192.168.10.3`, which exposed Apache Druid on port `8888`. THESEUS successfully exploited CVE-2021-25646 and obtained access to the host. During post-exploitation, the system recovered `/root/credentials.txt`, which contained credentials for the previously discovered internal SSH target.

The acquisition of these credentials changed the state of the previously unsuccessful attack path. The Orchestrator associated the newly discovered credential with the existing SSH service on `10.10.1.5`. THESEUS then revisited the previously identified target and authenticated using the newly acquired credential. The SSH login succeeded and the final flag was captured from `/home/vault_admin/flag.txt`.

The complete execution is illustrated in Figure 4.5.

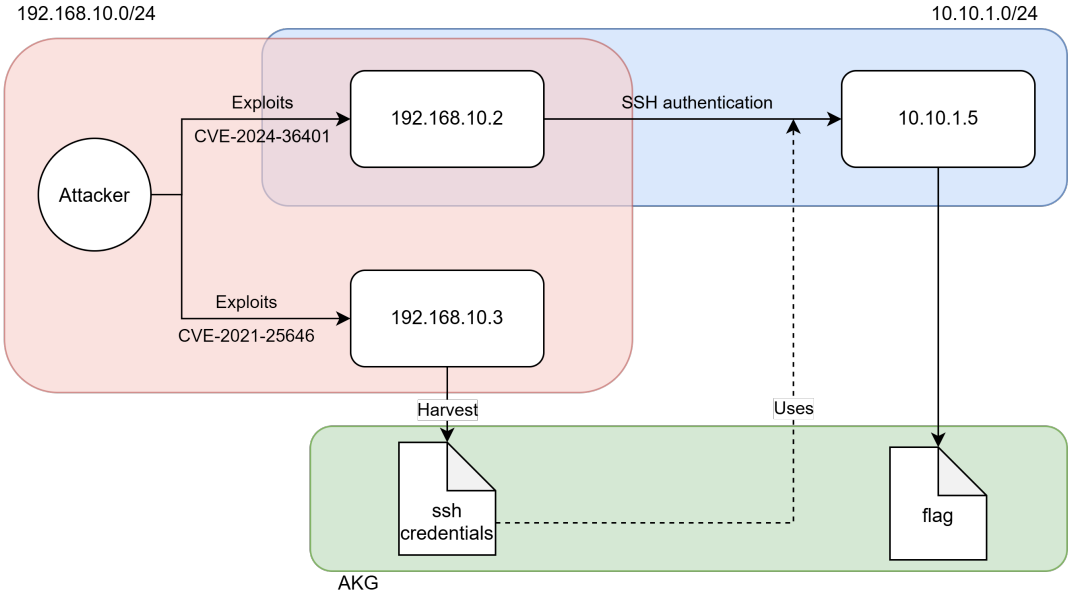

Figure 4.5: Cross-branch credential dependency and backtracking.

The experiment was considered successful because the final compromise depended on information acquired through a different attack branch from the one that originally discovered the target. In particular, the campaign did not follow a single linear sequence: the internal host was discovered first, the corresponding attack path initially failed due to missing credentials, and the system subsequently pursued an alternative perimeter host to obtain the missing information before returning to the original target.

This experiment therefore evaluates the ability of THESEUS to maintain and reassess previously acquired campaign state when an attack path does not immediately succeed. The successful completion of the chain demonstrates that information discovered in one branch can be associated with an opportunity created in another branch, allowing the system to backtrack, acquire a missing prerequisite, and resume a previously attempted attack.

4.3.6 Multi-Source Fragmented Credential Assembly (Failure Case)

To delineate the cognitive boundaries of the system on a technically solvable target, an additional experiment was designed requiring semantic correlation across fragmented artifacts on the initial foothold to unlock a downstream host. The scenario comprised a perimeter host (`192.168.10.2`) and an internal target (`10.10.1.6`) running SSH with strong credential requirements.

The intended attack path involved gaining initial access to the perimeter service on `192.168.10.2`, performing deep post-exploitation to locate and reconstruct a set of fragmented credential components stored locally, and subsequently pivoting to authenticate against `10.10.1.6`.

THESEUS successfully compromised `192.168.10.2` and mapped the internal route to the second host. However, the initial post-exploitation pass failed to recover the complete set of required credentials. Transitioning to the internal target, the Orchestrator initiated dictionary-based brute-force authentication against `10.10.1.6`. When repeated attempts failed, the system correctly deduced that it lacked the necessary credentials in the AKG and backtracked to the perimeter host.

Upon returning to the first host, the Orchestrator adapted its post-exploitation policy, directing the agent to focus specifically on credential recovery. The PostExploiter executed multiple targeted inspection attempts; however, it was unable to discover and correlate all necessary components. Lacking the complete credential set, the system returned to `10.10.1.6` to resume brute-forcing, subsequently failed again, and backtracked once more to the initial host.

This dynamic triggered an ongoing cyclic loop between the two targets: the system repeatedly alternated between focused yet incomplete post-exploitation passes on the perimeter host and unsuccessful brute-force attempts on the downstream target until the execution budget was exhausted.

This outcome demonstrates a specific reasoning limitation: while the framework can dynamically adjust its post-exploitation objectives and backtrack upon missing prerequisites, it struggles when the required artifacts are fragmented across subtle modalities, ultimately oscillating between two incomplete stages. The scenario is recorded as a failure in Table 4.5.

4.3.7 Quantitative Summary

Each chain-oriented scenario was executed $n = 5$ times, each run starting from a clean environment and a freshly initialized AKG, under the same model, timeout, and state-persistence configuration described in Section 4.1.2. A run was scored as successful only if the final campaign objective was reached (e.g., the final flag was captured). The outcome of each scenario is reported as the number of successful runs out of five.

The experiments were also analyzed using quantitative metrics extracted from the campaign graphs. The number of hosts discovered and compromised, established pivots, recorded events, strategic decisions, and harvested credentials were used to characterize the evolution of each campaign. These structural metrics refer to a single representative run per scenario (the first successful run), and are intended to describe the structure of a campaign rather than to be averaged across runs.

Table 4.5 summarizes the main quantitative results across the evaluated attack scenarios.

Scenario	Hosts Disc.	Hosts Pwn.	Pivots	Events	Decisions	Credentials	Successes
Multi-Hop Chain	3	3	2	72	7	4	2/5
Credential Reuse	2	2	1	29	5	4	5/5
PrivEsc + Lateral Movement	2	2	1	29	3	6	5/5
Adaptive Target Selection	3	2	1	21	3	1	5/5
Cross-Branch Backtracking	3	3	1	59	5	4	5/5
Fragmented Cred. Assembly	2	1	1	47	8	0	0/5

Table 4.5: Quantitative summary of chain-oriented evaluation scenarios. Structural metrics refer to a representative run; the last column reports successful runs out of $n = 5$.

The results show that the evaluated scenarios cover complementary forms of chain dependency. *Multi-Hop Chain* demonstrates successive network traversal across multiple internal segments, *Credential Reuse* demonstrates credential reuse across different hosts, *PrivEsc + Lateral Movement* combines local privilege escalation with credential-dependent lateral movement, *Adaptive Target Selection* validates dynamic host prioritization based on discovered attack vectors, and *Cross-Branch Backtracking* demonstrates cross-branch dependency and backtracking. Together, these scenarios evaluate progressively different requirements for maintaining, updating, and exploiting campaign state.

The *Multi-Hop Chain* scenario achieved the final objective in 2 out of 5 runs. In the successful executions, the system traversed three network segments and compromised the three target hosts through two successive pivots. The lower success rate is primarily associated with exploiting the third host, reached after two live pivots. In this condition, the deterministic payload engine (Section 4.4.1) still selects a payload that is correct with respect to the target's architecture, the exploit, and the module's constraints; however, unlike the single-pivot case, it does not guarantee that this payload establishes a session through a chain of two live pivots, since routing and binding behavior across nested pivot sessions is not fully captured by the engine's current model. This is consistent with the T-08 scenario in Section 4.4.1, where the same connection-mode decision was verified only as feasible against a declared topology, not as a session confirmed through two live, sequentially executed pivots.

When the engine's selected payload does not establish a session under these conditions, the system falls back to an LLM-proposed payload rather than a further deterministic selection. This fallback is therefore subject to the same limitations observed for unconstrained LLM payload selection elsewhere in this evaluation (Section 4.4.1), where the unconstrained LLM fails on every pivoted scenario. The observed 2/5 outcome is consistent with this: the deterministic engine's decision is not the point of failure, but once that decision does not succeed under live two-pivot conditions, the system has no second deterministic mechanism to fall back on, only the less reliable LLM-driven alternative.

Fragmented Credential Assembly is the only other scenario in which the final objective was never reached (0/5): in every run the system compromised the perimeter host and mapped the internal route, but did not recover the complete credential set required to access the downstream host (Section 4.3.6).

The quantitative measures should be interpreted primarily as descriptors of campaign complexity and execution behavior rather than as direct measures of system performance. In particular, the number of events and strategic decisions depends on the structure of each scenario and should not be used in isolation to compare efficiency across different attack chains. With only five runs per scenario, the success counts are likewise indicative of reliability rather than precise estimates of a success probability.

4.3.8 Discussion

Taken together, the experiments provide empirical evidence that THESEUS can maintain and exploit campaign state across attack stages whose prerequisites are generated during earlier parts of the execution. The evaluation covers multiple forms of dependency, including sequential network traversal, credential reuse, privilege-dependent information discovery, adaptive target selection, and cross-branch backtracking.

The *Multi-Hop Chain* scenario provides the clearest demonstration of stateful multi-hop execution. In the successful runs, three hosts distributed across three network segments were compromised through two successive internal pivots. However, the 2/5 success rate also highlights an important limitation of the current system. The final exploitation step is more constrained than the preceding ones because it must operate from the network position established after two pivots and establish a usable session on the third host. In particular, the exploit may require a payload with specific platform, architecture, transport, or compatibility properties. Metasploit's payload model reflects these constraints: compatible payloads are selected with respect to the exploit target and properties such as platform, architecture, size, and module compatibility.

In the unsuccessful *Multi-Hop Chain* executions, the system therefore did not fail at the level of payload-type selection: the deterministic engine's decision was correct with respect to architecture, exploit, and module constraints in every run, consistent with its behavior on the equivalent two-pivot case in Section 4.4.1 (T-08). The failures instead occurred when that payload did not establish a session through the two-hop pivot chain, at which point the system fell back to an LLM-proposed alternative, without the same deterministic guarantees, and that alternative did not succeed in the unsuccessful runs. This result separates three aspects of autonomous exploitation: maintaining campaign state and reaching the target, constructing a target-compatible exploitation action, and reliably falling back when that action does not establish a session under network conditions the deterministic engine does not yet fully model.

The *Credential Reuse* and *PrivEsc + Lateral Movement* scenarios further demonstrate that information obtained during post-exploitation can become a prerequisite for subsequent lateral movement. The *Cross-Branch Backtracking* scenario extends this behavior to a non-linear attack path: the internal target was discovered before its required credential, the corresponding attack path initially failed, and the system subsequently revisited the target after another branch provided the missing information.

The evaluation also exposes a second, distinct limitation of the current architecture. In the *Fragmented Credential Assembly* scenario, which resulted in a partial outcome, the system correctly detected missing prerequisites, adapted its post-exploitation objectives, and repeatedly backtracked between hosts; however, it was unable to fully reconstruct the fragmented credential set before the execution budget was exhausted.

This case demonstrates that while the state representation and orchestration mechanisms can support dynamic recovery and inter-host routing, they do not guarantee that complex, multi-stage reasoning problems will always be solved under tight operational constraints.

Overall, the chain-oriented experiments demonstrate that THESEUS is capable of executing state-dependent attack chains in which target reachability, credentials, privileges, and previously acquired knowledge influence subsequent actions. At the same time, the results identify two concrete sources of failure: payload-dependent exploitation at the end of a deeper multi-hop chain and incomplete reconstruction of fragmented prerequisites. The experiments are intentionally scenario-based, with each scenario targeting a specific architectural capability rather than estimating a population-level success probability. Their purpose is therefore to provide direct empirical evidence that the mechanisms introduced by the architecture operate across realistic multi-stage attack executions, while the larger AutoPenBench evaluation provides the quantitative assessment of general autonomous penetration-testing performance.

4.4 Experimental Validation of the Proposed Mechanisms

In addition to the end-to-end evaluation presented above, three targeted experiments isolate the main mechanisms introduced by THESEUS. The purpose of these experiments is to establish a direct connection between the architectural components described in Chapter 3 and their observable effects during execution. Rather than evaluating the system only through final attack success, each experiment focuses on a specific mechanism and records intermediate measurements relevant to its operation.

4.4.1 Deterministic Payload Selection Engine

To evaluate the effectiveness of the deterministic payload selection engine integrated into THESEUS, an empirical comparison was conducted across twelve exploitation scenarios ($N = 12$), varying target architecture, operating system, module-level payload constraints, and network topology depth (direct access versus 2-hop and 3-hop pivoting).

Three strategies are compared: an *unconstrained LLM*, prompted with target metadata and asked to produce a payload identifier directly; the *framework default*, the payload the underlying exploitation framework selects when none is set explicitly, resolved here through the framework's actual preferred-payload heuristic; and the *THESEUS deterministic engine* (Section 3.4).

A case is an *operational success* if the selected payload is module-compatible, architecture-compatible, topology-compatible, and produces a usable session, regardless of whether the session is optimal. Every payload selected across all three strategies is a valid, existing identifier; no case fails due to a malformed name.

Bimodal Payload Architecture and Pivot Uncertainty. Payload connectivity cannot always be determined from pre-exploitation metadata alone. In particular, pivoted

Table 4.6: Outcome summary across twelve payload-selection scenarios. ✓ = unqualified success, session verified end-to-end. S = sub-optimal but viable. F = non-viable. V* = correct payload *type* selected given the declared topology, but verified only as a feasible, compatible decision on a known static topology, not as a session established through a chain of multiple, live, previously executed pivots (see the failure-modes discussion below). Pivot scenarios (bold) isolate topology-aware and bimodal selection.

ID	Distinguishing axis	LLM	Framework default	THESEUS
T-01	Direct, x64, large payload pool	S (generic shell)	✓	✓
T-03	Direct, restrictive HTTP-header channel	S (fragile binary stager)	F (unstable binary stager)	✓
T-04	Direct, container-based execution	✓	✓	✓
T-05	Direct, 32-bit, interactive-only target	F (arch mismatch, daemon crash)	F (requires interactive TTY)	✓
T-06	Direct, command-only module	F (module-incompatible payload)	✓	✓
T-07	Direct, x64 kernel-level exploit	F (arch mismatch, target crash)	✓	✓
T-09	Direct, 32-bit target	F (arch mismatch, target crash)	✓	✓
T-10	Direct, text-only, no compatible default	F (module-incompatible payload)	F (no default resolves)	✓
T-11	Direct, x64, large payload pool	S (generic shell)	✓	✓
T-02	**2-hop pivot**	**F (reverse across pivot)**	**F (reverse across pivot)**	**✓ (bind, verified)**
T-08	**3-hop pivot**	**F (reverse across pivot)**	**F (reverse across pivot)**	**V* (bind selected, feasible)**
T-12	**2-hop pivot, restrictive channel**	**F (reverse across pivot)**	**F (reverse across pivot)**	**✓ (bind, verified)**
Success		**4/12 (33.3%)**	**6/12 (50.0%)**	**11/12 (91.7%) + 1/12 feasible (V*)**

targets may be unable to reach the attacker's listener, making conventional reverse payloads unsuitable, while local filtering may prevent bind payloads from accepting incoming connections. The problem becomes more pronounced in multi-hop scenarios, where reverse connectivity also depends on the capabilities of the intermediate pivot session.

The tested baselines do not explicitly account for this uncertainty. The framework default selects a single framework-provided payload, typically based on a reverse connection, while the unconstrained LLM predominantly selects conventional reverse payloads. THESEUS instead exposes a **dual, ranked proposal** (Section 3.4), comprising:

1. **Primary candidates** (`suggested_payload` and `top_candidates`), containing reverse payloads selected according to the target architecture and operating system, with the corresponding staged or unstaged variant retained as an alternative.

2. **Bind alternatives** (`bind_candidates`), containing module-compatible bind payloads that can be used when reverse connectivity is unsuitable.

For pivot depths ≥ 2, the runtime engine prioritizes `bind_tcp`, allowing Metasploit to use existing `autoroute` connectivity, while retaining reverse port-forwarding when required by the selected module. This bimodal strategy is the mechanism evaluated by the pivot scenarios in Table 4.6. For a single live pivot (T-02, T-12), the resulting bind session was verified end-to-end. For a chain of two live pivots (T-08), the same decision logic selects the correct payload type, but this was verified only as a feasible, topology-consistent choice against a known, static topology, not as a session independently established through two sequentially executed, live pivots; this case is therefore reported as V* rather than as an unqualified success.

Failure modes. The unconstrained LLM fails from module incompatibility (2/12), architecture mismatch (3/12), and topology blindness on every pivoted scenario (3/12), achieving only 33.3% (4/12) operational success, three of which produce sub-optimal generic shells. The framework default succeeds in 6/12 scenarios (50.0%); its failures stem from topology blindness across all three pivoted scenarios (blindly dispatching reverse payloads to an unreachable local listener), one module requiring an interactive TTY unavailable under autonomous execution (T-05), one module lacking any valid default resolution (T-10), and one binary stager destabilized by a restrictive HTTP header buffer (T-03). THESEUS resolves eleven of the twelve scenarios as unqualified successes, by normalizing architecture and platform identifiers, validating candidates against module-specific capabilities, and providing the agent with the dual reverse/bind modalities necessary to traverse isolated network topologies. The remaining scenario, T-08, is not a failure of the engine's decision logic: the correct payload type is selected, consistent with its behavior on T-02 and T-12, but this decision was verified only as feasible against a declared, static three-hop topology, not as a session confirmed through a chain of two live, sequentially executed pivots. This distinction becomes operationally relevant in Section 4.3, where the multi-hop chain scenario requires exactly this kind of session establishment through two live pivots.

Limitations. The twelve scenarios were constructed by the author to exercise the engine's own rules, not sampled independently. Beyond the live multi-hop verification gap identified above (T-08), none of the twelve scenarios exercises failure modes outside the engine's intended rule set: egress filtering on non-standard outbound ports, endpoint detection tooling capable of flagging unobfuscated payload signatures, the space and bad-character constraints of classic buffer-overflow exploits, and local firewalling on a pivoted target that blocks an otherwise topologically correct bind payload are all plausible failure modes the engine does not currently account for. The 11/12 result should be read as evidence that the engine correctly implements its intended rules under verified conditions, not as an unbiased estimate of its success rate on unseen targets or on longer live pivot chains. This is consistent with the broader evaluation limitations discussed in Section 4.5.

4.4.2 AKG-Based Context Management

The second experiment evaluates the role of the AKG in maintaining the state of a multi-step engagement without requiring the complete execution history to be carried in every prompt. Two scenarios are used: the cross-branch, multi-hop chain described in Section 4.3.5 (22 decision steps, 29 nodes and 44 edges in the final AKG), and a second, independent credential-escalation and lateral-movement chain (19 decision steps, 27 nodes and 30 edges in the final AKG), chosen to check whether the observed behavior is scenario-specific.

Context model. At each decision step, three quantities are tracked. T_{history} is the token count of the naive, unbounded concatenation of every tool's raw output up to that step. T_{reports} is the cumulative token count of the condensed textual reports returned to the orchestrator by delegated sub-agents (Scout, Coder, PostExploiter), rather than

the raw tool output those sub-agents consumed internally; at the very first step, no sub-agent has yet been delegated, so $T_{\text{reports}} = 0$ by construction. T_{akg} is the token count of the distilled graph state exposed to the orchestrator. THESEUS's effective dynamic context is $T_{\text{THESEUS}} = T_{\text{akg}} + T_{\text{reports}}$, and the complete prompt sent to the model is $T_{\text{prompt}} = T_{\text{THESEUS}} + T_{\text{overhead}}$, where T_{overhead} is the fixed cost of system instructions and tool schemas (measured at approximately 2,370 tokens and constant across both scenarios). Reporting T_{THESEUS} separately from T_{prompt} isolates the cost of the state representation itself from the fixed overhead a naive agent would also incur.

Report compression varies by phase. Unlike T_{akg}, which compresses monotonically as state is distilled into graph properties, the local ratio between a sub-agent's raw tool output and the report it returns is not constant: it is lowest during exploitation steps, where several thousand tokens of exploitation-framework output condense into a short session-confirmation message, and highest during reconnaissance steps, where port and service tables are retained largely intact because they carry information the orchestrator still needs. Across both scenarios this local ratio ranges from approximately approximately 4% to 43%, consistent with the type of operation being summarized rather than with the step index.

Table 4.7: Context size at representative steps of the two scenarios. $T_{\text{THESEUS}} = T_{\text{akg}} + T_{\text{reports}}$; compression is $T_{\text{history}}/T_{\text{THESEUS}}$ at that step.

Scenario	Step	T_{history}	T_{akg}	T_{THESEUS}	Compr.
Cross-branch (22 steps)	1 (start)	720	285	285	2.53×
	6 (first exploitation)	16,610	940	4,350	3.82×
	12 (second target reached)	29,410	1,520	7,355	4.00×
	22 (final objective)	59,280	2,206	11,326	5.23×
Credential/lateral (19 steps)	1 (start)	680	260	260	2.62×
	5 (first exploitation)	11,650	690	2,960	3.94×
	10 (pivot established)	22,900	1,150	5,085	4.50×
	19 (final objective)	41,800	1,562	7,802	5.36×

Table 4.8 summarizes both scenarios in full. Averages are computed as the mean of the per-step compression ratio, not as the ratio of the two summed columns, since these are different quantities; both are reported for transparency.

Discussion. Even under the conservative accounting that credits the orchestrator with the full cumulative report text a naive implementation might also retain, THESEUS's dynamic context remains roughly five times smaller than the raw history at the end of both campaigns, and this advantage is present from the first few steps rather than emerging only over a long run. The compression ratio increases over the course of each campaign, from around 2.5×–2.6× at the first step to over 5× at the final step, because raw tool output keeps accumulating while the report and graph components grow only when new entities or relationships are discovered. The two scenarios, despite differing in length, structure, and the specific hosts and services involved, show consistent final compression (5.23× and 5.36×) and a comparable

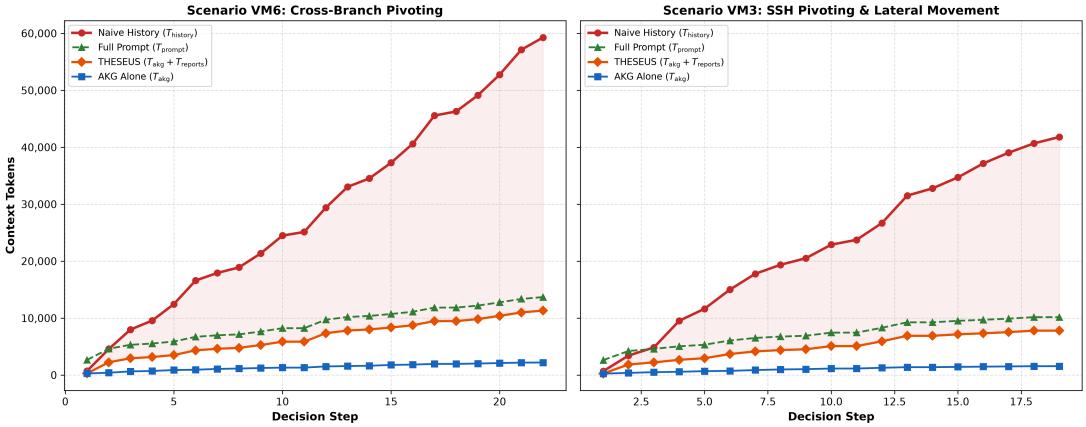

Figure 4.6: Context size over decision steps for both scenarios: naive raw history, THESEUS's full prompt, THESEUS's dynamic context ($T_{\text{akg}} + T_{\text{reports}}$), and the AKG state alone. The shaded area marks the context volume avoided relative to the naive baseline.

Table 4.8: Summary comparison across both scenarios.

Metric	**Cross-branch (22 steps)**	**Credential/lateral (19 steps)**
Final T_{history}	59,280 tok	41,800 tok
Final T_{reports}	9,120 tok	6,240 tok
Final T_{akg}	2,206 tok	1,562 tok
Final T_{THESEUS}	11,326 tok	7,802 tok
Final compression	5.23×	5.36×
Mean per-step compression	4.08×	4.21×
Final context reduction	80.9%	81.3%
Local report/raw ratio, range	5%–40%	4%–43%

range for the local report/raw ratio, which is a weak indication that the result is not an artifact of one specific scenario, though two scenarios remain a small basis for that claim.

Limitations. These results describe two specific campaigns designed by the author, not a sample drawn from a broader distribution of engagements; a shorter, single-host engagement would show a smaller absolute reduction, since the fixed prompt overhead becomes proportionally larger relative to a small T_{THESEUS}.

4.4.3 State Restoration and Reproducibility

The final experiment evaluates the persistence and restoration mechanisms of THESEUS. The scenario used is the two-hop `Cross-Branch Credential Dependency and Backtracking` environment described in Section 4.3.5, in which two perimeter hosts are compromised and a pivot route into an internal subnet is subsequently established. At this point, with both perimeter sessions active, the pivot route configured, seven credentials harvested, and an internal host identified as the next target, execution is intentionally interrupted by terminating all active sessions, simulating a framework restart or a network interruption.

The AKG is fully serialized to disk at every write operation, so no in-memory state is required for restoration. The restoration procedure consists of a single call that reads the persisted AKG, orders the recorded exploitation steps topologically, and replays each one with the exact parameters stored at recording time. Upon successful re-exploitation, the procedure re-registers the new session identifiers in the AKG and re-applies any pivot routes associated with the original session.

The experiment was repeated across five independent runs. In each run, session termination was first confirmed, the restoration procedure was then invoked, and the resulting state was compared against the pre-interruption snapshot along four dimensions: active sessions, pivot routing, credential completeness, and reachability of the internal target.

Results. All five runs completed successfully (5/5). The mean restoration time was 11.5 ($\sigma = 1.9$, min 7.9, max 13.2). Across all runs, the following held without exception: both perimeter sessions were re-established and registered in the AKG with updated identifiers; the pivot route was re-applied, restoring full reachability into the internal subnet; all seven credentials harvested during the original run were present in the restored AKG, with no post-exploitation or credential-gathering step re-executed; and the internal target remained confirmed reachable through the restored route in every run. Individual run timings are reported in Table 4.9.

Discussion. The restoration latency covers only the re-exploitation phase: the reconnaissance, scanning, and vulnerability-identification steps that would precede exploitation in a fresh run are entirely skipped, since the corresponding nodes are already present in the persisted AKG. A complete fresh execution of the same scenario, from initial host discovery to pivot establishment, takes on the order of several minutes,

Table 4.9: Session restoration results across five independent runs.

Run	Restore (s)	Sessions	Route	Reachability	Outcome
1	13.2	✓	✓	✓	Pass
2	12.0	✓	✓	✓	Pass
3	12.6	✓	✓	✓	Pass
4	7.9	✓	✓	✓	Pass
5	12.0	✓	✓	✓	Pass
Avg	**11.5**				**5/5**

so restoration reduces the time needed to reach a ready state by roughly two orders of magnitude. Because the AKG serializes not only structural graph data but also phase-progress flags and the provenance of each exploitation step, the resumed execution correctly identifies the next uncompromised target without issuing any redundant tool calls. These results show that the persistence mechanism successfully restored the tested campaign state across all five independent interruption and restoration runs, supporting its use in long-running or interrupted engagements.

4.5 Evaluation Limitations

The evaluation provides evidence for the effectiveness of THESEUS across single-target exploitation, multi-stage attack chains, and individual mechanisms supporting persistent autonomous execution. However, several limitations should be considered when interpreting the results.

First, most experimental results are based on a limited number of repeated executions. The main AutoPenBench evaluation uses five runs per target, while the chain-oriented and mechanism-level experiments also rely on relatively small samples. These repetitions provide an indication of consistency, but they are not sufficient to establish statistically robust estimates of general performance.

Second, the chain-oriented scenarios were specifically designed to exercise properties relevant to the proposed architecture, including credential reuse, privilege escalation, adaptive target selection, backtracking, and multi-hop exploitation. They therefore provide controlled evidence that these capabilities can be exercised end-to-end, but they should not be interpreted as an unbiased sample of real-world penetration-testing campaigns.

A similar limitation applies to the deterministic payload-selection experiment. The twelve scenarios were constructed to cover different architectures, module constraints, operating systems, and network topologies, and the resulting $11/12$ success rate demonstrates that the implemented selection rules correctly handled the tested cases. It does not, however, constitute an independent estimate of payload-selection success across arbitrary environments.

The AKG-based context-management experiment is likewise limited to two multi-step scenarios. The observed reduction in context size demonstrates the behavior of the implemented mechanism in these campaigns, but additional and substantially

longer engagements would be required to characterize its scaling behavior more generally.

The comparison with existing systems also requires care because the reported systems do not necessarily use identical models, execution budgets, success criteria, numbers of repetitions, or evaluation protocols. In particular, target-level success over five attempts is not equivalent to sub-task-level success over five attempts. Consequently, the comparison is intended to provide contextual evidence rather than a strictly controlled head-to-head benchmark.

The AutoPenBench comparison has a further scope limitation. AutoPenBench primarily evaluates vulnerability-exploitation tasks at the target level, whereas the chain-oriented experiments in this thesis explicitly evaluate multi-stage campaigns involving persistent state, pivots, and dependencies between successive actions. The two evaluations therefore measure related but distinct aspects of autonomous penetration testing.

Finally, the experiments were conducted using a fixed local execution environment and a specific model and configuration. The results therefore characterize the evaluated THESEUS implementation under the reported conditions rather than establishing model-independent or environment- independent performance. Similarly, operational metrics such as tool-call counts and token consumption should not be interpreted as directly comparable with systems whose internal execution protocols and accounting methods differ.

4.6 Evaluation Summary

The experimental evaluation provides evidence for the main capabilities targeted by THESEUS, while also highlighting the boundaries of the current evaluation. The results can be summarized with respect to the research questions introduced in Chapter 1.

RQ1 – Autonomous vulnerability exploitation. The AutoPenBench evaluation shows that THESEUS successfully completed the final objective in 22 out of 33 targets on the first recorded run, corresponding to a target-level success rate of 66.67%. When five independent executions are considered, 28 out of 33 targets were solved in at least one run (84.85%). At the run level, 104 out of 165 executions were successful (63.03%), while the overall milestone completion rate was 85.29%. These results indicate that the system can autonomously perform the complete exploitation workflow across a heterogeneous collection of security tasks, although performance varies substantially across categories.

RQ2 – Performance relative to existing autonomous systems. The first-run target-level result of 66.67% provides the most direct comparison with the single-attempt AutoPenBench result reported for GPT-4o, which achieved 21.21% on the same benchmark. This comparison indicates a substantial difference in target-level success under the respective evaluation configurations. However, the comparison does not constitute a controlled experiment because the systems differ in model, architecture, execution strategy, and other implementation details. Results reported for

systems using repeated sub-task-level evaluation are therefore presented as context rather than as directly equivalent measurements.

RQ3 – Multi-stage and multi-host attack execution. The chain-oriented evaluation demonstrates that THESEUS can maintain an engagement across multiple stages and hosts while preserving information required by subsequent actions. Successful scenarios include credential reuse across hosts, privilege-escalation-assisted lateral movement, adaptive target selection, and multi-hop exploitation. In particular, the two-stage credential-reuse, privilege-escalation, adaptive-selection, and cross-branch scenarios achieved $5/5$ successful executions, while the three-stage multi-hop scenario achieved $2/5$. The latter also exposes a concrete failure mode in which exploitation from a deeper pivot position can become constrained by payload and connectivity requirements.

RQ4 – Persistent state and non-linear attack dependencies. The cross-branch credential-dependency scenario provides evidence that the system can preserve intermediate discoveries, backtrack from an unproductive branch, and later reuse information obtained from another compromised host. The final AKG in the corresponding experiment contained 29 nodes and 44 edges after 22 decision steps. This behavior is relevant to attack campaigns in which the information required to compromise a target is not available along a single linear path.

RQ5 – Supporting mechanisms for autonomous execution. The mechanism-level experiments provide evidence for three architectural components. First, the deterministic payload-selection engine successfully resolved 11/12 author-designed diagnostic scenarios, compared with $4/12$ for unconstrained LLM selection and $6/12$ for framework-default selection. Second, the AKG-based context-management experiment reduced the final context representation relative to the corresponding raw execution history by approximately 80.9% and 81.3% in the two evaluated scenarios, corresponding to final compression factors of $5.23\times$ and $5.36\times$, respectively. Third, the state-restoration experiment successfully restored the tested campaign state in all five executions, with a mean restoration time of 11.5 s ($\sigma = 1.9$ s). Together, these results support the role of deterministic execution logic, persistent state, and compact state representation in the overall architecture.

Overall, the evaluation shows that THESEUS can execute autonomous penetration-testing workflows that extend beyond isolated vulnerability exploitation and include persistent state, multi-host progression, and dependencies between successive attack stages. The experiments also identify concrete limitations, particularly in deeper multi-hop exploitation and in fragmented credential acquisition. The results should therefore be interpreted as evidence that the proposed architecture can support these capabilities under the evaluated conditions, rather than as evidence of general autonomous penetration-testing performance across arbitrary environments.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis presented THESEUS, an agentic system designed to support autonomous penetration testing through the discovery and execution of multi-step exploit chains. The main objective was to move beyond approaches that treat penetration testing primarily as a sequence of independent vulnerability identification and exploitation tasks, and instead provide the agent with an explicit representation of the engagement state and of the relationships between discovered assets, compromised hosts, credentials, exploits, and subsequent attack steps.

The proposed architecture combines a central orchestrator with specialized agents for reconnaissance, exploit development, and post-exploitation activities. A key component of the system is the Asset-Oriented Sequential Attack Knowledge Graph (AKG), which provides persistent state across the engagement. The graph records relevant information about the environment and the attack chain, allowing the system to distinguish already explored resources from newly discovered targets and to reason about possible subsequent steps. This representation also supports the restoration of the engagement state and the execution of attacks that span multiple hosts and network segments.

The implementation integrates the agentic architecture with established penetration-testing tooling, including Metasploit, while maintaining a separation between the strategic reasoning performed by the orchestrator and the execution of individual tactical operations. This design makes it possible to combine LLM-based reasoning with concrete tool interaction and to evaluate the resulting system on attack scenarios involving sequential exploitation and pivoting.

The experimental evaluation considered chain-oriented scenarios in which successful execution depends not only on exploiting individual vulnerabilities, but also on preserving information across multiple steps and using the results of previous compromises to reach subsequent targets. The evaluation therefore considered both final attack outcomes and intermediate milestones, providing a more detailed view of the agent's ability to construct and execute complete attack chains.

Overall, the results show that an explicit persistent representation of the attack state can provide a practical foundation for autonomous penetration testing. In particular, the combination of an LLM-based agent architecture, persistent attack knowledge,

and tool-based execution allows THESEUS to operate across multiple stages of an engagement rather than treating each exploitation step as an isolated task. At the same time, the experiments highlight that autonomous attack-chain execution remains sensitive to target selection, environment-specific assumptions, and the reliability of generated exploitation actions. These aspects motivate several directions for future development.

5.2 Future Work

5.2.1 GOAD Support

A first direction for future work is the extension of the experimental evaluation to more complex and standardized environments such as the Game of Active Directory (GOAD). THESEUS is already capable of performing multi-step operations involving credential discovery, privilege escalation, lateral movement, and pivoting. Supporting GOAD would therefore primarily extend the range and complexity of environments in which these capabilities can be evaluated, rather than introducing new attack primitives.

In particular, an Active Directory environment provides a richer combination of hosts, services, accounts, credentials, privileges, and trust relationships. These characteristics would allow the system to be evaluated on attack chains whose progression depends on relationships between identities and machines, in addition to the network topology already considered in the current scenarios.

GOAD would also provide an opportunity to further validate the AKG representation. Active Directory entities and their relationships could be incorporated into the graph alongside the existing network-oriented information, allowing the system to maintain a unified representation of both infrastructure and identity-related state throughout an engagement.

The integration would consequently extend THESEUS toward more heterogeneous attack environments while providing an additional benchmark for evaluating persistent state management, target selection, and multi-step attack-chain execution.

5.2.2 Improved Host and Target Selection

Another important direction concerns the selection of the next target. In the current architecture, target selection is performed using the information available in the AKG and the reasoning capabilities of the orchestrator. A more systematic target-selection mechanism could improve both efficiency and reliability, particularly as the number of discovered hosts and possible attack paths increases.

Future work could introduce an explicit target-ranking strategy that considers multiple properties of candidate targets, such as network reachability, available services, known vulnerabilities, previously obtained credentials, estimated exploitation difficulty, and the potential impact of compromising the target on the remainder of the attack chain. Rather than selecting a target solely from its current descriptive information, the system could estimate the utility of different actions with respect to the overall objective of the engagement.

Such a mechanism could also exploit the topology already represented in the AKG. Candidate targets could be evaluated according to their position in the current attack graph, allowing the orchestrator to prioritize actions that are expected to unlock previously inaccessible portions of the environment. This would provide a more explicit separation between state representation and strategic target selection.

5.2.3 AKG-to-Shell Attack Reproduction

A further direction is the generation of executable shell commands directly from the attack knowledge represented in the AKG. At present, the graph primarily serves as a persistent representation of the engagement state and of the relationships established during execution. A natural extension would be to use this representation to reconstruct the sequence of actions that produced a successful attack chain.

The AKG could be extended with sufficient execution metadata to associate each attack step with the command, tool invocation, parameters, target, and resulting state transition. Once a chain has been completed, these records could be traversed according to the graph dependencies and transformed into a reproducible sequence of shell commands.

This would enable THESEUS to produce an explicit attack procedure in addition to the final state of the engagement. Such a capability could be useful for reproducibility, debugging, human verification, and the analysis of successful attack chains. It would also provide a clear distinction between the reasoning process used to discover an attack and the concrete sequence of commands required to reproduce it.

An important aspect of this extension would be preserving dependencies between commands. For example, a command that uses a credential obtained during an earlier exploitation step should only be generated after the corresponding credential acquisition step. Similarly, commands executed after a pivot should include the network context established by the preceding steps. The AKG therefore provides a natural basis for reconstructing an ordered attack procedure rather than simply concatenating independent tool calls.

5.2.4 Enhanced Payload Determinism for Deep Pivot Chains

The experimental evaluation highlighted a specific operational boundary in the current deterministic payload-selection engine. While the bimodal payload strategy successfully resolves target compatibility and egress uncertainty for direct access and single-hop pivots, its empirical reliability decreases over deeper attack chains. As observed in the multi-hop chain scenario, establishing a session through two or more live, nested pivots introduces complex routing and binding constraints that are not fully captured by the current engine's topological model. When a theoretically compatible payload fails to traverse a multi-hop tunnel, the system falls back to an unconstrained LLM decision, which lacks deterministic guarantees and reduces the overall reliability of deep internal exploitation.

Future work should extend the payload engine to incorporate a more comprehensive model of multi-hop network transport. Rather than relying solely on a depth

threshold to switch from reverse to bind payloads, the engine could dynamically orchestrate nested transport mechanisms. For reverse connections, this would involve automating the establishment of cascaded port-forwarding relays across multiple intermediate compromised hosts, ensuring a continuous, explicitly routed return path to the attacker. For bind connections, the framework could integrate pre-exploitation transport checks to verify that local firewalls on the deeply nested target actually permit incoming connections from the preceding pivot node.

Additionally, integrating alternative transport protocols, such as SMB named pipes or explicit SOCKS5 proxy chains orchestrated directly through the AKG, could provide more resilient staging over high-latency or fragile tunnels. By expanding the deterministic engine to account for the specific traversal properties of nested sessions, THESEUS could maintain the same level of exploitation reliability at arbitrary network depths as it currently demonstrates on perimeter and single-pivot targets.

Bibliography

[1] Agno. *Agno Documentation: Teams and Multi-Agent Coordination*. https://docs.agno.com/teams/overview. Accessed: 2026-09-16. 2026.

[2] M. U. Aksu, E. Altuncu, and K. Bicakci. "A first look at the usability of OpenVAS vulnerability scanner". In: *Proceedings of the Workshop on Usable Security (USEC 2019)*. Internet Society. 2019. DOI: 10.14722/usec.2019.23026.

[3] Brian Challita and Pierre Parrend. *RedTeamLLM: an Agentic AI framework for offensive security*. 2025. arXiv: 2505.06913 [cs.CR]. URL: https://arxiv.org/abs/2505.06913.

[4] G. Deng, Y. Liu, Y. Li, R. Yang, X. Xie, J. Zhang, H. Qiu, and T. Zhang. *What makes a good LLM agent for real-world penetration testing?* 2026. arXiv: 2602.17622.

[5] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. "PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing". In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 847–864. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/deng.

[6] M. A. El Yagouby, A. Lahmadi, M. Zakroum, O. Festor, and M. Ghogho. "LLM-CVX: A benchmarking framework for assessing the offensive potential of LLMs in exploiting CVEs". In: *Proc. 2025 Workshop on Artificial Intelligence and Security (AISec '25)*. 2025. DOI: 10.1145/3733799.3762978.

[7] Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. *LLM Agents can Autonomously Exploit One-day Vulnerabilities*. 2024. arXiv: 2404.08144 [cs.CR]. URL: https://arxiv.org/abs/2404.08144.

[8] Richard Fang, Rohan Bindu, Akul Gupta, Qiusi Zhan, and Daniel Kang. *LLM Agents can Autonomously Hack Websites*. 2024. arXiv: 2402.06664 [cs.CR]. URL: https://arxiv.org/abs/2402.06664.

[9] Georgi Gerganov. *llama.cpp: LLM inference in C/C++*. https://github.com/ggml-org/llama.cpp. 2023.

[10] L. Gioacchini, A. Delsanto, M. Mellia, I. Drago, G. Siracusano, and R. Bifulco. "AutoPenBench: A vulnerability testing benchmark for generative agents". In: *Proc. 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*. 2025. DOI: 10.18653/v1/2025.emnlp-industry.114.

[11] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. *Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection*. 2023. arXiv: 2302.12173 [cs.CR]. URL: https://arxiv.org/abs/2302.12173.

[12] A. Happe and J. Cito. *Benchmarking practices in LLM-driven offensive security: Testbeds, metrics, and experiment design*. 2025. arXiv: 2504.10112.

[13] Andreas Happe and Jürgen Cito. *Cochise: A Reference Harness for Autonomous Penetration Testing*. 2026. arXiv: 2605.11671 [cs.CR]. URL: https://arxiv.org/abs/2605.11671.

[14] H. Kong, D. Hu, J. Ge, L. Li, T. Li, and B. Wu. *VulnBot: Autonomous penetration testing for a multi-agent collaborative framework*. 2025. arXiv: 2501.13411.

[15] Alyzia-Maria Konsta, Beatrice Spiga, Alberto Lluch Lafuente, and Nicola Dragoni. *A Survey of Automatic Generation of Attack Trees and Attack Graphs*. 2023. arXiv: 2302.14479 [cs.CR]. URL: https://arxiv.org/abs/2302.14479.

[16] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023. arXiv: 2309.06180 [cs.LG]. URL: https://arxiv.org/abs/2309.06180.

[17] W. G. Li, A. Abuadbba, K. Moore, and D. D. Kim. *APT-Agent: Automated penetration testing using large language models*. 2026. arXiv: 2605.24949.

[18] Z. Li, T. Zhou, J. Zhu, D. Li, X. Zhu, D. Wei, J. Liu, and M. Song. "PenExpert: A multi-agent hybrid LLM-expert system framework for autonomous penetration testing". In: *Expert Systems with Applications* 331 (2026), p. 133284. DOI: 10.1016/j.eswa.2026.133284.

[19] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. "Lost in the Middle: How Language Models Use Long Contexts". In: *Transactions of the Association for Computational Linguistics* 12 (2024), pp. 157–173. DOI: 10.1162/tacl_a_00638. URL: https://aclanthology.org/2024.tacl-1.9/.

[20] Jiaqi Luo, Jiarun Dai, Zhile Chen, Jia Xu, Weibing Wang, Yawen Duan, Brian Tse, Geng Hong, Xudong Pan, Yuan Zhang, and Min Yang. "The Emergence of Autonomous Penetration Capabilities in Large Language Model-Powered AI Systems". In: *arXiv preprint arXiv:2606.13079* (2026). DOI: 10.48550/arXiv.2606.13079.

[21] P. D. Luong, L. T. G. Bao, N. V. K. Tam, D. H. N. Khoa, N. H. Quyen, V.-H. Pham, and P. T. Duy. *xOffense: An autonomous multi-agent framework for penetration testing with domain-adapted large language models*. arXiv:2509.13021, rev. 2026. 2026. arXiv: 2509.13021.

[22] Wuyuao Mai, Geng Hong, Qi Liu, Jinsong Chen, Jiarun Dai, Xudong Pan, Yuan Zhang, and Min Yang. *Shell or Nothing: Real-World Benchmarks and Memory-Activated Agents for Automated Penetration Testing*. 2025. arXiv: 2509.09207 [cs.CR]. URL: https://arxiv.org/abs/2509.09207.

[23] Orange Cyberdefense e mayfly277. *GOAD: Game of Active Directory*. https://github.com/Orange-Cyberdefense/GOAD. Repository GitHub. 2022.

[24] R. Missotten, V. Rimmer, W. Mees, and L. Desmet. *On the potential of LLMs for offensive security: Benchmarks vs. operational reality*. Manuscript. 2025.

[25] Katsuaki Nakano, Reza Fayyazi, Shanchieh Jay Yang, and Michael Zuzak. *Guided Reasoning in LLM-Driven Penetration Testing Using Structured Attack Trees*. 2025. arXiv: 2509.07939 [cs.CR]. URL: https://arxiv.org/abs/2509.07939.

[26] Rapid7. *Metasploit Framework Documentation*. https://docs.metasploit.com/. Accessed: 2026-09-16. 2026.

[27] R. Rogers. *Nessus network auditing*. 2nd. Syngress/Elsevier, 2008.

[28] Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. *Identifying the Risks of LM Agents with an LM-Emulated Sandbox*. 2024. arXiv: 2309.15817 [cs.AI]. URL: https://arxiv.org/abs/2309.15817.

[29] Xiangmin Shen, Lingzhi Wang, Zhenyuan Li, Yan Chen, Wencheng Zhao, Dawei Sun, Jiashui Wang, and Wei Ruan. *PentestAgent: Incorporating LLM Agents to Automated Penetration Testing*. 2025. arXiv: 2411.05185 [cs.CR]. URL: https://arxiv.org/abs/2411.05185.

[30] Jasmin Wachter. *Graph models for Cybersecurity – A Survey*. 2023. arXiv: 2311.10050 [cs.CR]. URL: https://arxiv.org/abs/2311.10050.

[31] Benlong Wu, Guoqiang Chen, Kejiang Chen, Xiuwei Shang, Jiapeng Han, Yanru He, Weiming Zhang, and Nenghai Yu. *AutoPT: How Far Are We from the End2End Automated Web Penetration Testing?* 2024. arXiv: 2411.01236 [cs.CR]. URL: https://arxiv.org/abs/2411.01236.

[32] X. Wu, Y. Tian, Y. Chen, P. Ye, X. Cui, J. Jia, S. Li, J. Liu, and W. Niu. "CurriculumPT: LLM-based multi-agent autonomous penetration testing with curriculum-guided task scheduling". In: *Applied Sciences* 15 (2025), p. 9096. DOI: 10.3390/app15169096.

[33] Binyan Xu, Haitao Li, and Kehuan Zhang. *LLM Agents Are Latent Context Managers: Eliciting Self-Managed Context via State Proprioception*. 2026. arXiv: 2606.30005 [cs.CL]. URL: https://arxiv.org/abs/2606.30005.

[34] R. Yang, M. Cheng, G. Deng, T. Zhang, J. Wang, and X. Xie. *PentestEval: Benchmarking LLM-based penetration testing with modular and stage-level design*. 2025. arXiv: 2512.14233.

[35] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. arXiv: 2210.03629 [cs.CL]. URL: https://arxiv.org/abs/2210.03629.

[36] A. K. Zhang et al. "CyBench: A framework for evaluating cybersecurity capabilities and risks of language models". In: *Proc. International Conference on Learning Representations (ICLR)*. 2025.

Acknowledgements

This thesis would not have taken the shape it has without the guidance of Professor Cosimo Anglano, my supervisor, who followed this work with patience and rigor from its earliest, less defined stages through to its completion. I am sincerely grateful for the time he dedicated to this project and for the insight he brought to it.

Beyond the work itself, these years have been shaped just as much by the people around me. My family has been a constant presence behind everything I have done, offering steady support and never once doubting where this was heading, even in the moments when I did.

I would also like to thank my fellow students, who shared this path with me through lectures, exams, and the long stretches in between, as well as more coffees than I can count. The exchanges, the shared frustrations, and the mutual encouragement made this journey far less solitary than it could have been, and I am grateful to have gone through it alongside them.

I have also been fortunate to share this path with friends who made the harder stretches easier to get through, whether through a well-timed distraction, a much-needed laugh, or simply being there. Their presence made a real difference, and I do not take it for granted.

Lastly, to everyone else who lent a hand, an idea, or a bit of encouragement along the way, however small it may have seemed at the time: thank you. It added up to more than you probably realize.