Chapter 5: Automated Penetration Testing

While a vulnerability scanner identifies potential weak points, an autonomous penetration testing agent conducts deeper analysis to determine which assets are actually susceptible to compromise by an attacker.

Learning Objectives

By the end of this chapter, students will be able to:

1. Position autonomous penetration testing as **prescriptive analytics** within the SEAS-8414 taxonomy and explain how it transforms theoretical vulnerability data into confirmed exploitation evidence.
2. Describe the five-stage pentest campaign lifecycle (plan, validate, execute, evidence, report) and trace a complete campaign through the pentest campaign orchestrator.
3. Compare rule-based and PPO reinforcement learning agent architectures, identifying trade-offs in determinism, adaptability, and processing overhead.
4. Derive the PPO clipped surrogate objective from first principles and explain why clipping prevents catastrophic policy updates.
5. Explain the four safety modes (simulation, shadow, controlled, autonomous) and articulate the security, legal, and ethical rationale for each constraint level.
6. Trace the exploit executor's protocol detection and module dispatch logic from action selection through evidence recording.
7. Analyze the MDP environment's state representation, reward function, and done condition, and explain how reward shaping guides agent behavior.
8. Implement a hierarchical policy that decomposes action selection into strategic (host) and tactical (exploit type) decisions.
9. Design a curriculum for learning that progressively increases environmental complexity from single-host to full-network campaigns.
10. Construct tamper-evident evidence chains using SHA-256 hash linking and Merkle root verification, and explain their role in legal defensibility.
11. Map pentest actions to MITRE ATT&CK techniques and generate coverage reports that quantify the breadth of kill-chain exercises.
12. Evaluate the ethical and legal boundaries of autonomous offensive testing, distinguishing authorized penetration testing from unauthorized access.

Agentic Lens

For the first time, the agent is empowered to alter the system state intentionally. This leap in complexity pushes students to think rigorously not only about exploiting logic, but also about action selection, authorization, and the wisdom of knowing when to halt. A real-world example highlights what is at stake. In 2022, a cybersecurity consultancy unintentionally disrupted hospital operations during a penetration test when automated exploitation actions were performed outside the authorized window, leading to several diagnostic systems going offline for hours. The incident was traced to failures in both explicit authorization checks and proper escalation protocols. This underscores the necessity of not just technical capability, but operational discipline and strict adherence to authorization boundaries in autonomous testing.

Best Practices for Authorization: Ensuring Authorization Boundaries in Practice.

To stop similar incidents and effectively enforce authorization boundaries, organizations should implement a set of concrete best practices before, during, and after autonomous penetration testing campaigns:

- Obtain written authorization specifying scope, targets, methods, timing, and escalation contacts. Store this documentation in a location that is easily accessible to all team members and system operators.
- Explicitly configure technical controls that encode authorization limits into the agent's safety controller and campaign parameters. Use allow lists for target hosts and plug in time-window scheduling to prevent agent actions outside approved windows.
- Implement a pre-campaign checklist requiring a final review of scope, time windows, approved targets, and safety mode before enabling exploitation. Only permit the campaign launch after completing this checklist and obtaining sign-off from an authorized stakeholder.
- Use real-time audit logging and monitoring so that any attempt to interact with an out-of-scope or out-of-window system is immediately detected and triggers both automated blocking and escalation to a human operator.
- Require escalation and explicit approval for any deviation from the original authorization, whether for new targets, additional techniques, or extended time windows. Document all changes and approvals in the campaign audit trail.
- Debrief after every campaign, reviewing audit logs for any boundary violations, and update policies or technical safeguards as needed based on lessons learned from real incidents.

By following these steps, organizations set up a practical and reliable way to enforce authorization boundaries. These specific actions are just as important for safe autonomous testing as technical innovations, and they help connect legal requirements with day-to-day operations.

- **Agent role.** Choose the next bounded offensive action under explicit authorization.
- **Observations.** Attack graph state, confirmed findings, prior action outcomes, safety mode, and evidence chain status.
- **Tools.** Rule-based policies, PPO policies, exploit modules, credential checks, ATT&CK mappings, and evidence collectors.
- **State.** Current footholds, failed actions, action budget, approval mode, and evidentiary hashes.
- **Verifier.** Scope checks, module postconditions, evidence integrity checks, and human approval gates for higher-risk actions.
- **Guardrails:** The agent must operate only within its authorized limits, clearly distinguish between simulation and production, and choose actions that can be reversed and tracked.
- **A major failure can happen if a**n agent chases rewards without considering boundaries, creating attack chains that look impressive in logs but cross authorization limits or fail to produce solid evidence. The agent should follow authorized offensive testing, not act like an uncontrolled attacker.
- **Threat model.** The defender is using controlled offensive actions to measure exploitability, validate exposure, and gather stronger evidence than passive analysis alone can provide.
- **Threat model.** The adversary of interest is any actor who could exploit the same path under the same network conditions, but the chapter does not grant the agent unlimited freedom to imitate that actor.

- **Assumption.** Inputs from discovery, enrichment, vulnerability scoring, and attack graphs are sufficient to bound the candidate target set and reduce unnecessary probing.
- **Assumption.** Every action is governed by explicit policy, approval mode, and rollback constraints. The MDP is a planning abstraction inside those constraints, not a license to ignore them.
- **Assumption.** Reward weights are decision aids. They are not universal truths about business value, legal acceptability, or operational risk.
- **Scope boundary.** This chapter does not authorize destructive payloads, uncontrolled persistence, or indefinite privilege escalation for the sake of a cleaner score.

5.1 Analytics Context: From Prediction to Prescription

The analytics taxonomy structuring this course has a critical inflection point between Chapters 4 and 5. Every chapter before this one is observational. Descriptive analytics counts devices. Diagnostic analytics identifies them. Detective analytics finds their vulnerabilities. Predictive analytics models attack paths. None of these chapters changes the network state.

> In contrast, this chapter is where actions start to actively change the network itself.

Prescriptive analytics addresses the question: given all available information, what actions should be taken? In clinical contexts, this distinction is clear. Descriptive analytics identifies an elevated white blood cell count; diagnostic analytics determines a bacterial cause; predictive analytics forecasts a high likelihood of sepsis within twelve hours. Prescriptive analytics recommends a specific intervention, such as administering a particular antibiotic at a defined dosage and route. This marks the transition from analysis to action.
In cybersecurity, this change is significant. Here, “action” means taking offensive steps: the agent tries to exploit systems, sends packets to break in, and carries out credential stuffing, CVE exploitation, lateral movement, and privilege escalation. This is not just watching anymore—it is a planned test of your own systems to answer an important question: *Can an attacker really move from one point to another?*

> **Implementation Note:** This capability is planned but not yet implemented in the current codebase. The design is sound, but the implementation does not yet exist. Do not present it as operational.

Descriptive	What devices exist?	Ch 1	Unchanged
Diagnostic	What are they doing?	Ch 2	Unchanged
Detective	What vulnerabilities exist?	Ch 3	Unchanged
Predictive	What attack paths are likely?	Ch 4	Unchanged
Prescriptive	**What offensive tests should we run?**	**Ch 5 (this chapter)**	**Modified**
Simulation	What happens if we apply this fix?	Ch 6	Simulated

Table 5.0. The prescriptive inflection point. Chapter 5 is the first chapter to modify (or simulate modifying) the network state actively.

Chapter 5: Automated Penetration Testing

The Meridian Capital story shows why validation is important. Qualys found the vulnerabilities, and the Chapter 4 attack graph could have mapped the possible route. But neither proved the path was actually open. The TV's default credentials could have been changed, a firewall could have blocked the firmware flaw, or the ACL misconfiguration could have already been fixed. Only an agent that actually tests the exploitation chain can distinguish between theoretical risk and confirmed exposure.

This difference matters in the real world. When Elena Vasquez showed the pentest results to Meridian's board, she explained that it took only fourteen minutes to go from a conference room TV to the trading engine. That same day, the board approved a $2.1 million network segmentation project. For eighteen months, theoretical risk led to no action, but confirmed exploitation changed everything right away.

5.1.1 Prescriptive Analytics in Offensive Security

This prescriptive lens fundamentally shapes system design, reaching far beyond classroom theory.
A vulnerability scanner works like a measuring tool, focusing on coverage, accuracy, and speed. In contrast, an autonomous pentest agent makes decisions, aiming for high-quality actions, enforcing safety rules, keeping evidence reliable, and learning as it goes.

Think about the choices that an autonomous penetration testing agent faces. At each step, it must choose from many available actions: which host to target, which vulnerability to try, which protocol to attack, and which credentials to use. Each decision depends on what the agent currently knows, what it has already found, and safety rules set by the operator. This situation is best seen as a step-by-step decision-making problem with some uncertainty.

To model this rigorously, we turn to the Markov Decision Process (MDP), a mathematical framework suited to environments where each choice influences the evolving state. In this formalism, an agent observes the current network state, picks an action, receives feedback, and updates its knowledge for the next step. Practical solution techniques for this framework include rule-based heuristics and reinforcement learning. This chapter introduces both approaches in the context of autonomous penetration testing.

5.1.2 Inputs from Earlier Phases

The pentest engine does not operate in isolation. It consumes the complete output of Phases 1 through 14:

- **Phase 1 (Discovery):** Host inventory with IP, MAC, vendor, hostname
- **Phase 2 (Enrichment):** Service banners, TLS certificates, JARM fingerprints, open ports
- **Phase 3 (Identification):** Device type, product, firmware version, confidence score
- **Phase 4 (CVE Assessment):** Vulnerability records with CVE IDs, CVSS scores, and affected CPEs
- **Phase 7 (Default Credentials):** Known credential pairs per host and protocol
- **Phase 13 (Protocol Security):** Protocol-level findings from grammar inference and mutation fuzzing
- **Chapter 4 (Attack Graph):** NetworkX directed graph modeling reachability and attack paths

Each input shapes the agent's choices. A host with no known vulnerabilities or default credentials gives the agent only a few options. But a host with several critical CVEs, default Telnet credentials, and a protocol-fuzzing result

provides the agent with many possible actions. The agent has to balance priorities across different hosts and action types simultaneously. This chapter uses Markov Decision Processes (MDPs) to formalize this step-by-step decision-making and hierarchical policy design to show how the agent chooses both which host and which action at each step. By introducing these tools, readers can expect a clear explanation of how the agent plans and chooses actions.

A pentest campaign is not a single action. It is a managed lifecycle with five stages:

```
plan -> validate -> execute -> evidence -> report
```

The PentestCampaign class in campaign_orchestrator.py manages this lifecycle. It connects the safety controller, the agent (rule-based or PPO), and the evidence collector. It emits SSE-compatible progress events for real-time dashboard updates and enforces a hard timeout to prevent runaway campaigns.

Figure 5.1: Pentest Campaign Lifecycle

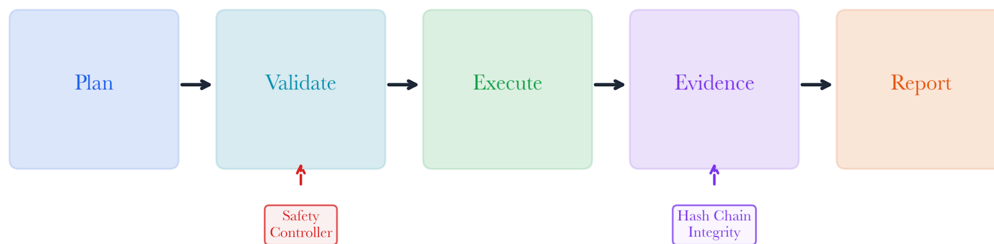

Figure 5.1: Pentest campaign lifecycle. The orchestrator manages five stages from planning through reporting, with safety controller gates at validation and post-execution filtering.

5.2.1 The PentestRunRequest

Every campaign begins with a request that specifies six parameters:

```
class PentestRunRequest(BaseModel):
    scan_id: str # Which scan's data to test against
    safety_mode: PentestSafetyMode# simulation | shadow | controlled | autonomous
    max_actions: int # Budget cap (default 100, max 10,000)
    timeout_seconds: int # Hard wall-clock limit (default 600s)
    agent_type: str # "rule" | "ppo" | "hybrid"
    target_ips: list[str] |None# Restrict to specific hosts (None = all)
```

The scan_id links the campaign to a completed scan. The agent pulls host data from Redis (scan:{id}: hosts) to construct the MDP environment. The safety_mode determines which actions the safety controller permits. The max_actions and timeout_seconds provide dual budget constraints to prevent computational runaway and network saturation.

At Meridian Capital, Elena Vasquez launched the initial campaign in simulation mode with max_actions=200 and a 10-minute timeout. When the simulated results showed the six-hop path, she escalated to controlled mode, targeting only the conference room TV and the environmental sensor, with explicit approval for each exploitation attempt.

5.2.2 Host Filtering and Target Validation

Before execution, the orchestrator applies two filters. First, if target_ips is specified, only matching hosts are included. Second, every target is validated against the safety controller. In controlled mode, a host not on the approved list is logged as blocked, but does not halt the campaign.

This filtering is vital for safety in real environments. For example, a trading firm cannot allow its order management system to be used autonomously during market hours. The target filter ensures the agent never interacts with systems outside the approved scope, regardless of what the attack graph shows as reachable.

5.2.3 Agent Execution

The orchestrator runs the agent in a separate thread to keep the event loop responsive. It uses a strict timeout to stop the agent if it takes too long to run. This is important because a reinforcement learning agent exploring many options can take much longer than a rule-based agent, and real deployments need predictable time limits.
Before the agent runs, the orchestrator optionally consults the LLM strategy advisor. This module sends a sanitized network inventory to Claude and receives prioritized attack strategy hints as a dictionary mapping IPs to priority weights between 0.1 and 2.0. The hints bias the rule-based agent's scoring function without overriding it. The LLM never generates exploit code or shell commands, nor does it output any instructions that would trigger unsafe or out-of-scope actions. It provides strategic guidance only.

All outputs from LLM are validated and sanitized before use. The orchestrator applies strict schema validation regarding the returned hints (making sure that they are limited to scores within the allowed range and target only hosts that are within the authorized scope). Any anomalous or unanticipated output is ignored. Additionally, the LLM's suggestions are only applied as scoring weights and cannot override rule-based or safety controller constraints. This assurances that no LLM-driven hint will cause the agent to take an action that violates campaign boundaries or safety policies.

If the LLM call fails, returns invalid data, or is disabled, the agent proceeds without hints, treating their absence as non-fatal degradation.

5.2.4 Post-Execution Safety Filtering 5.2.4

> After the agent finishes, the orchestrator applies a safety filter to the results. In simulation mode, all finding descriptions are prefixed with [SIMULATED]. In shadow mode, any exploit findings that bypassed the safety controller are demoted to confirmed=False with a [SHADOW - NOT EXECUTED] prefix.

This layered approach enforces safety in three ways: the safety controller checks every action before it runs, the MDP environment uses fixed success rules, and the orchestrator filters results after the agent finishes. If there is a bug in one layer, the others can catch it.

Chapter 5: Automated Penetration Testing

5.3 The MDP Environment

The pentest problem maps naturally to a Markov Decision Process. Formally, we define the MDP as the tuple **(S, A, T, R, gamma)** where:

- **S** is the state space: network configurations encoding host compromise status, discovered services, available credentials, and the agent's current network position.
- **A** is the action space: eight discrete action types (exploit CVE, exploit default credentials, exploit protocol finding, pivot, privilege escalate, exfiltrate proof, port scan, service enumerate), each parameterized by a target host and action-specific details.
- T: S x A -> Delta(S) is the transition function mapping a (state, action) pair to a distribution over successor states. In simulation mode, T is deterministic (hash-based selectors ensure reproducibility).
- R: S x A x S -> R is the reward function providing immediate numerical feedback that balances exploitation success, reconnaissance value, efficiency, and technique diversity.
- gamma = 0.99 is the discount factor, weighting near-term rewards heavily while still valuing long-horizon multi-hop attack chains.

At each step, the agent observes the current network state, selects an action, receives a reward, and transitions to a new state. The Markov property applies here because the state includes everything the agent needs to decide: what is compromised, what is reachable, and what is known. How the agent got to this state does not change which actions are best from here.

5.3.1 State Representation

The PentestState dataclass tracks seven elements:

```
@dataclass
class PentestState:
compromised_hosts: set[str] # IPs where exploitation succeeded
accessible_hosts: set[str] # IPs reachable from current position
discovered_services: dict[str, list] # IP -> list of service dicts
discovered_vulns: dict[str, list]# IP -> list of vulnerability dicts
credential_store: dict[str, list]# IP -> list of credential dicts
current_host: str # Agent's current network position
step_count: int # Actions taken so far
total_reward: float # Cumulative reward
```

For the RL agent, this state must be converted to a fixed-length numeric vector. The to_vector method produces five features per host in the environment:

```
[is_compromised, is_accessible, num_services, num_vulns, has_creds]
```

For an environment with N hosts, the state vector has 5N features. For example, a typical /24 scan with 30 hosts results in a 150-feature state vector. This simple encoding discards information about how hosts are connected while keeping the state size fixed, which works well for linear models and neural networks.

Figure 5.2: State Vector Encoding

	is_compromised	is_accessible	num_services	num_vulns	has_creds
Host A 10.4.2.5	1	1	3	2	1
Host B 10.4.2.17	0	1	0	5	0
Host C 10.4.3.42	0	0	0	0	0

```
State vector = [1, 1, 3, 2, 1, 0, 1, 0, 5, 0, 0, 0, 0, 0, 0]
```

Dimensionality: 5 features x 3 hosts = 15

Figure 5.2: State vector encoding. Each host contributes five features to the fixed-length state vector consumed by the RL agent's policy network.

5.3.2 Action Space

The agent can perform eight discrete action types:

exploit_cve	T1190	Exploit a known CVE on a target host
exploit_default_creds	T1078	Try known default credentials
exploit_protocol_finding	T1210	Exploit a protocol-level weakness
pivot	T1021	Move laterally to a new host
privilege_escalate	T1068	Escalate privileges on a compromised host
exfiltrate_proof	T1041	Collect proof-of-access artifacts
port_scan	T1046	Scan ports on an accessible host
service_enum	T1046	Enumerate services on an accessible host

Table 5.1. Discrete action types with MITRE ATT&CK Enterprise technique mappings.

Each action includes a target IP, an optional target port, and other details like CVE information, credential pairs, or the source host for pivoting. The total set of possible actions at any time is the set of all combinations of action types, target hosts, and these details. For example, on a network with 30 hosts, each with 5 vulnerabilities and 2 credential pairs, there can be more than 500 possible actions.

The available_actions method generates the legal action set from the current state. It enforces preconditions: you cannot exploit a CVE on a host whose services are undiscovered, you cannot pivot from a host that is not compromised, and you cannot escalate privileges on a host you have not compromised. These preconditions prune the action space and prevent wasted actions on moves that are impossible.

5.3.3 Transition Dynamics

State transitions are deterministic in simulation mode. The environment uses hash-based selectors instead of random number generators, ensuring that the same (action, state) pair always produces the same outcome. This reproducibility is essential for debugging, for comparing agent strategies, and for producing evidence that can be independently verified.
CVE exploitation success depends on the CVSS score:

- CVSS >= 7.0: exploitation succeeds (high-severity vulnerabilities are reliably exploitable)
- CVSS >= 5.0: exploitation succeeds only if the target port has been discovered via service enumeration
- CVSS 5.0: exploitation fails (too difficult to exploit reliably without additional research)

Default credential exploitation succeeds deterministically when the host's scan data includes known credentials. Pivoting succeeds with high probability (95% threshold) when the source host is compromised, and the target is accessible. Service enumeration always succeeds on accessible hosts and populates the state discovered services and discovered vulnerabilities from the base scan data.
This predictable behavior is helpful for learning. Students can see exactly why the agent succeeded or failed at each step. There is no randomness from different seeds. Running the same campaign on the same data always gives the same results.

5.3.4 Reward Function

The reward function guides agent behavior through a combination of positive incentives and negative penalties:

Step cost	-1.0	Every action (encourages efficiency)
Failed action	-5.0	Action did not achieve its goal
New host compromised	+10.0	First successful exploit on a host
Critical host compromised	+20.0	Compromised host is high-value (router, SCADA, DC)
New service discovered	+5.0	Service enumeration reveals new data
Credential found	+3.0	Pivoting or scanning reveals credentials
Unique MITRE technique	+2.0	First use of a MITRE technique in this campaign

Table 5.2. Reward signal components for the MDP environment.

The -1.0 step cost is the most important part of the reward. Without it, the agent would try to take as many actions as possible, collecting small rewards from tasks such as service scans and credential checks. The step cost pushes the agent to be efficient and find the shortest path to important targets.
The MITRE technique bonus rewards the agent for exercising diverse tactics across the kill chain rather than repeating the same exploit type. A campaign that uses only exploit_default_creds might compromise many hosts but exercises only one ATT&CK technique. The novelty bonus encourages the agent to pursue broader coverage, resulting in more comprehensive testing results.
At Meridian Capital, the +20 critical host bonus drove the agent to prioritize the path toward the trading engine over easier but less consequential targets. The environmental sensor in the IoT VLAN was compromised not because it was the easiest target, but because it served as the gateway to higher-value systems.

5.3.5 Accessible Host Computation

The environment determines which hosts the agent can reach at each step. Two strategies are available:

1. **Attack graph traversal:** When a NetworkX attack graph is provided from Chapter 4, reachable hosts are computed by following outgoing edges from compromised nodes. This model's actual network topology, firewall rules, and VLAN segmentation.
2. **Same-subnet heuristic:** When no attack graph is available, the envirSame-subnet heuristic: When no attack graph is available, the environment uses a /24 subnet heuristic: all hosts on the same /24 as any compromised host are considered accessible. This reasonably approximates flat networks but overestimates in segmented environments. The entry point models an external attacker who can access all externally facing services. As the agent compromises hosts and pivots, the accessible set may expand to include hosts on previously unreachable subnets.

5.3.6 Episode Termination

An episode ends when either condition is met:

1. All accessible hosts have been compromised (the agent has achieved maximum coverage).
2. The step count reaches max_steps (the agent has exhausted its action budget).

The first condition means the agent succeeded completely. The second means the agent ran out of allowed actions before it could finish exploring the network. The difference between these outcomes is a good way to measure performance: if campaigns often end at the action limit, it may mean the action budget should be higher, or the agent's strategy needs improvement.

5.4 Rule-Based Agent

The simplest agent strategy is a deterministic priority heuristic that mirrors a human pentester's decision process. The RuleBasedPentestAgent selects actions by scoring every available action and choosing the highest-scoring one.

5.4.1 Scoring Function

The scoring function assigns priority bands that reflect how a skilled human pentester allocates attention:

```
exploit_default_creds on new host->100(cheapest, highest success rate)
exploit_cve with high CVSS->up to 100(CVSS * 10)
exploit_protocol_finding->60
privilege_escalate->50
```

```
pivot to uncompromised host->40 + vuln_count * 2 (cap 20)
exfiltrate_proof->30
service_enum on unexplored host->20
port_scan on unexplored host->20
```

Already compromised targets receive a 0.1x multiplier on exploit actions, preventing redundant attacks on hosts that are already compromised. Pivot targets receive a bonus proportional to the number of known vulnerabilities on the destination, steering the agent towards hosts with rich attack points.

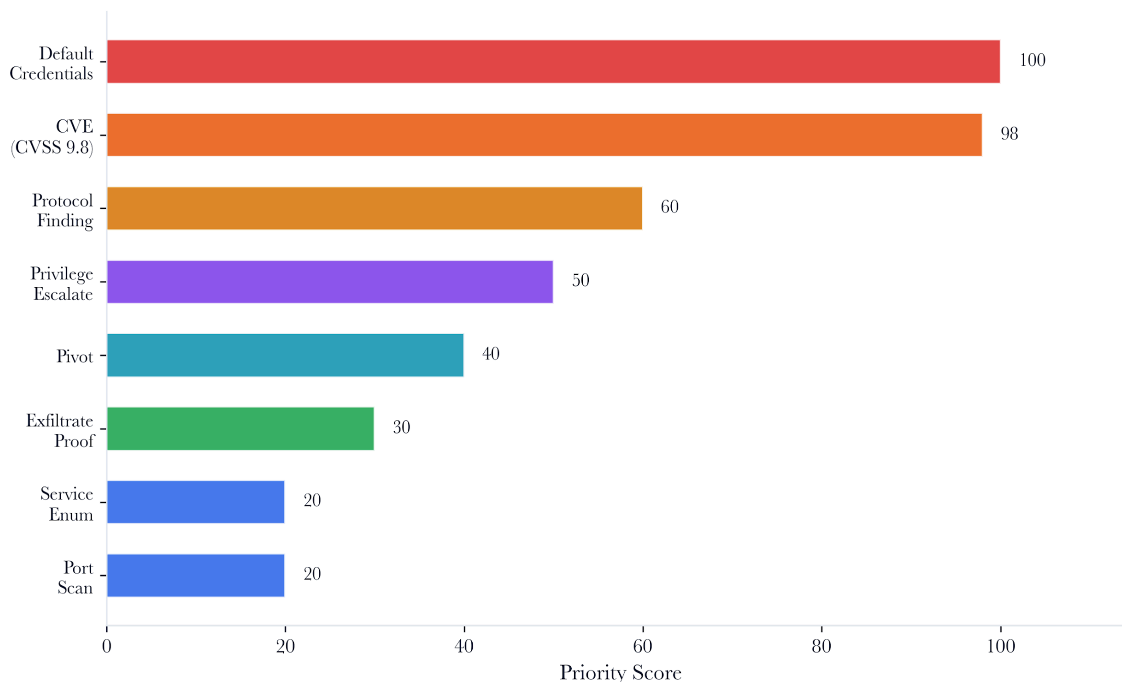

Figure 5.3: Rule-based agent scoring function. Priority bands mirror a human pentester's decision hierarchy: credentials first, then CVEs by severity, then lateral movement.

5.4.2 LLM Strategy Hints

The rule-based agent accepts optional strategy hints from the LLM advisor. Each hint is a weight between 0.1 and 2.0 for a specific target IP. The weight multiplies the scoring function's output for actions targeting that IP, capped at 2.0x. This allows the LLM's strategic assessment to influence priority ordering without overriding the deterministic logic.

The cap is important. An unbounded LLM weight could cause the agent to fixate on a single target regardless of its vulnerability profile. The 2.0x cap ensures the LLM can promote targets without dominating the scoring function.

At Meridian Capital, the LLM advisor suggested a priority weight of 1.8 for the environmental sensor at 10.4.2.17, recognizing that its SCADA-adjacent network position made it a high-value pivot point. Without the hint, the rule-based agent would have reached the same target eventually, but the LLM prioritization reduced the number of wasted actions on lower-value hosts.

5.4.3 Determinism and Reproducibility

The rule-based agent is fully deterministic. Given the same host data, scoring function, and strategy hints, it produces identical results every run. This property is essential for several use cases:

- **Regression testing:** If a code change alters pentest results, the change is in the logic, not in random seed sensitivity.
- **Evidence integrity:** The same inputs must produce the same outputs for forensic and legal review.
- **A/B comparison:** Comparing the rule-based agent to the PPO agent requires holding the environment constant.

5.4.4 Evidence Recording

Every action the agent takes is recorded with a SHA-256 evidence hash computed over the canonical JSON serialization of the action, result, and timestamp. The hash chains into the campaign audit trail, providing a tamper-evident record of exactly what the agent did and what it observed.

```
evidence_data = {
    "step": step,
    "action_type": action.action_type.value,
    "target_ip": action.target_ip,
    "target_port": action.target_port,
    "success": result.success,
    "reward": result.reward,
    "timestamp": timestamp_iso,
    "details": result.info,
}
evidence_hash = hashlib.sha256(
    json.dumps(evidence_data, sort_keys=True).encode()
).hexdigest()
```

The sort_keys=True parameter ensures canonical JSON ordering, so the same data always produces the same hash regardless of dictionary insertion order. This is a subtle but critical detail: without it, Python's dictionary ordering could produce different hashes for semantically identical evidence, breaking the chain's integrity guarantee.

5.5 PPO Reinforcement Learning Agent

The rule-based agent works well for straightforward networks. But it cannot adapt. Its priority ordering is fixed at design time. On networks with unusual topologies, unexpected service configurations, or novel vulnerability combinations, a fixed heuristic may miss opportunities that a learning agent would discover.

The PPOPentestAgent implements Proximal Policy Optimization, a policy gradient algorithm that learns from experience across campaigns. It uses a hierarchical policy with two decision levels: strategic (which host to target) and tactical (which action type to use on that host).

5.5.1 Why PPO?

The choice of PPO over other RL algorithms reflects three practical constraints:

1. **Sample efficiency:** Pentest campaigns are expensive to simulate. Each campaign involves tens to hundreds of environmental steps. PPOs on-policy learning with mini-batch updates extracts maximum information from each campaign's experience.
2. Stability: The clipped surrogate objective prevents catastrophic policy updates where a single bad batch of experience causes the agent to forget everything it has learned. In a security context, stability matters: an agent that randomly "unlearns" how to exploit default credentials after a bad training batch would be worse than useless.
3. Simplicity: PPO requires only a policy network and a value network. There are no target networks to synchronize (as in DQN), no replay buffers to manage (as in SAC), and no separate actor and critic update schedules (as in A2C). The implementation fits in a single file with no external dependencies beyond numpy.

5.5.2 Hierarchical Policy Architecture

The action space in autonomous pentesting is two-dimensional: the agent must choose *where* (which host) and *what* (which action type). Treating this as a flat action space over (host, action_type) pairs leads to a product space that grows linearly with the number of hosts and quadratically with the number of vulnerabilities per host.
The hierarchical policy decomposes this into two sequential decisions:

1. **Strategic level:** A softmax distribution over hosts. The agent first selects which host to target.
2. **Tactical level:** A softmax distribution over action types. Given the chosen host, the agent selects what to do.

Both levels use linear function approximation: the state vector is multiplied by a weight matrix to produce logits, which are converted to probabilities via softmax. The combined log-probability of an action is the sum of the strategic and tactical log-probabilities.

Figure 5.4: Hierarchical Policy Architecture

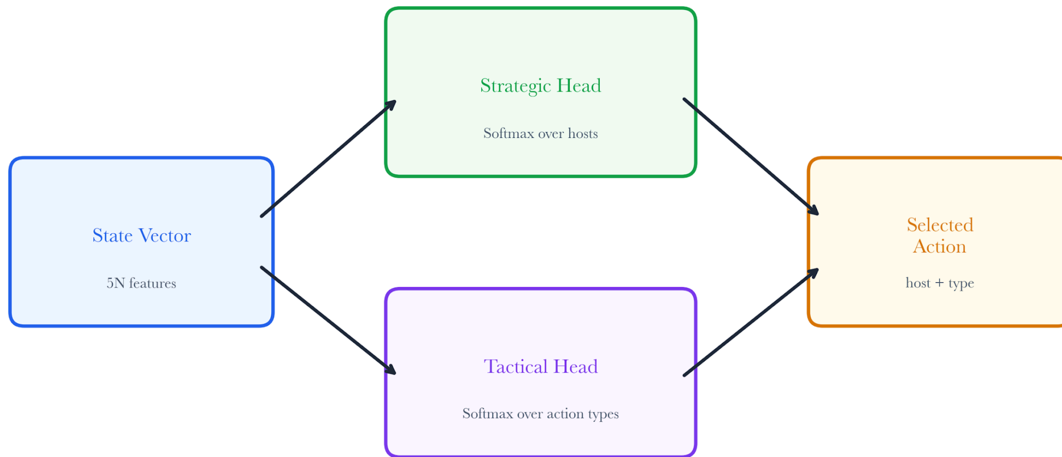

Figure 5.4: Hierarchical policy architecture. The state vector feeds two parallel softmax heads, one for host selection (strategic) and one for action type selection (tactical). Masking ensures only valid actions receive probability mass.

```
class HierarchicalPolicy:
def __init__(self, state_dim, num_hosts, num_action_types=8, lr=3e-4):
self.strategic_weights = np.zeros((state_dim, num_hosts))
self.tactical_weights = np.zeros((state_dim, num_action_types))
self.value_weights = np.zeros(state_dim)
```

The zero initialization is deliberate. At the start of training, both distributions are uniform (a softmax over zeros), meaning the agent explores randomly. As training progresses, the weights diverge from zero, concentrating probability mass on hosts and action types that yield high rewards.

5.5.3 Action Masking

Not every host has every action type available. A host with no known vulnerabilities cannot be targeted with exploit_cve. A host that has already been scanned should not receive another service_enum. The policy handles this via logit masking: unavailable hosts receive a logit of -1e9 before softmax, effectively setting their probability to 0.

```
mask = np.full(self.num_hosts, -1e9)
for ip in available_host_ips:
idx = ip_to_idx.get(ip)
if idx is not None and idx <self.num_hosts:
    mask[idx] =0.0
strategic_probs =self._softmax(strategic_logits + mask)
```

This masking is applied at both the strategic and tactical levels. The result is that the agent can never select an impossible action, even if its learned weights would otherwise assign it high probability. This is a hard constraint, not a soft penalty in the reward function.

Note: In reinforcement learning, an 'impossible action' is one whose preconditions are not satisfied in the current state. This is a technical term, not a claim about physical impossibility. The action space is defined by state-dependent preconditions.

5.5.4 The PPO Objective

The standard PPO clipped surrogate objective balances two goals: improving the policy based on new experience and preventing the new policy from diverging too far from the old one.
Let pi_old(a|s) denote the probability of taking action a in state s under the old policy, and pi_new(a|s) denote the same under the updated policy. The probability ratio is:

```
r(theta) = pi_new(a|s) / pi_old(a|s)
```

The clipped objective is:

```
L = min(r * A, clip(r, 1-epsilon, 1+epsilon) * A)
```

where A is the advantage estimate, and epsilon is the clipping parameter (default 0.2). When the advantage is positive (the action was better than expected), the ratio is clipped to 1.2 to prevent the policy from moving too aggressively toward that action. When the advantage is negative, the ratio is clipped to 0.8 to prevent the policy from moving too aggressively away from the action.

Figure 5.5: PPO Clipped Surrogate Objective

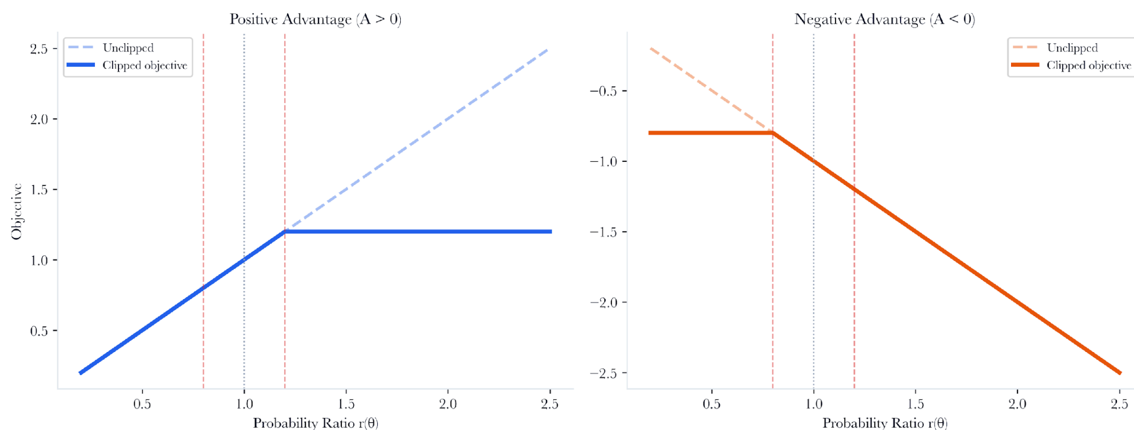

Figure 5.5: PPO clipped surrogate objective. The blue region shows the unclipped objective; the orange shows the clipped objective. The flat segments prevent large policy updates.

The implementation computes the clipped objective per sample in the mini-batch:

```
ratio = np.exp(new_lp - old_lp)
clipped = np.clip(ratio, 1.0- clip_epsilon, 1.0+ clip_epsilon)
surrogate =min(ratio * adv, clipped * adv)
```

5.5.5 Generalized Advantage Estimation

The advantage function measures how much better an action is compared to the average action in that state. PPO uses Generalized Advantage Estimation (GAE), which balances bias and variance through a parameter lambda:

```
A_t = sum_{l=0}^{T-t} (gamma * lambda)^l * delta_{t+l}
where delta_t = r_t + gamma * V(s_{t+1}) - V(s_t) is the temporal difference error.
```

The implementation computes GAE in reverse order through the trajectory:

```
for t inreversed(range(n)):
if dones[t]:
        next_value =0.0
        gae =0.0
        delta = rewards[t] +self.gamma * next_value - values[t]
        gae = delta +self.gamma *self.gae_lambda * gae
        advantages[t] = gae
        returns[t] = advantages[t] + values[t]
        next_value = values[t]
```

With gamma=0.99 and lambda=0.95, the GAE heavily weights near-term TD errors while still incorporating long-horizon returns. This is appropriate for pentesting, where the reward for compromising a critical host may be many steps removed from the initial exploitation that enabled the attack path.

5.5.6 Value Function

The value function V(s) estimates the expected cumulative reward from state s. It uses linear function approximation:

```
V(s) = s^T * w_value
```

The value function is updated via gradient descent on the mean squared error between predicted values and actual returns:

```
L_value = (1/N) * sum (V(s_i) - R_i)^2
```

The value function serves two purposes. First, it provides the baseline for advantage estimation (the “compared to average” in “how much better than average”). Second, it provides diagnostic information: if the value function consistently overestimates, the agent is more optimistic about its capabilities than the evidence supports.

5.5.7 PyTorch MLP Policy

When PyTorch is available, and ml_torch_enabled is set in the scanner configuration, the agent uses a neural network policy rather than linear function approximation. The PentestMLP is a three-layer network with a shared backbone (256, 128 hidden units with ReLU activations) and three heads:

- **Strategic head:** Linear(128 -> num_hosts), produces logits for host selection
- **Tactical head:** Linear(128 -> num_action_types), produces logits for action type selection

- **Value head:** Linear(128 -> 1), produces scalar state value estimate

The shared backbone enables the network to learn common features (which hosts are compromised and which have rich vulnerability profiles) that inform both host selection and action type selection. Xavier initialization ensures reasonable initial gradient magnitudes. Gradient clipping at max_norm=0.5 prevents exploding gradients during early training.
The PyTorch policy has the same API as the numpy policy (select_action, get_value, update_policy, update_value, save, load), allowing transparent swapping without changes to the agent loop.

5.5.8 Training Loop

The PPO agent collects experience during a campaign (in the run() method) and then updates its policy (in the update() method). The update iterates num_epochs times over the experience buffer in shuffled mini-batches of size batch_size:

```
for _epoch inrange(self.num_epochs):
        indices = np.arange(n_samples)
        np.random.shuffle(indices)

for start inrange(0, n_samples, self.batch_size):
        end =min(start +self.batch_size, n_samples)
        batch_idx = indices[start:end]
        p_loss =self.policy.update_policy(...)
        v_loss =self.policy.update_value(...)
```

With default settings (4 epochs, batch size 32), a campaign that produces 100 experience samples generates approximately 4*ceil (100/32) = 16 gradient updates per training iteration. This is sufficient for meaningful policy improvement on small networks without overfitting to a single campaign's experience.

5.6 Reward Shaping

The base reward function in the MDP environment (Table 5.2) provides coarse guidance: a +10 for compromise, -1 per step, -5 for failure. While sufficient for basic learning, this sparse signal leads to three degenerate behaviors: exploitation-only convergence (the agent skips reconnaissance), technique monoculture (the agent fixates on credential exploitation), and stealth blindness (the agent generates unnecessary detectable scanning traffic). Reward shaping addresses these pathologies by decomposing the reward into independently weighted components that balance exploitation success against stealth, breadth against depth, and confirmation against exploration.
The shaped reward is computed as:

R_shaped = R_base + w_cov * R_coverage + w_conf * R_confirmation - w_stealth * P_stealth + w_nov * R_novelty

The RewardShaper module implements four fine-grained components that can be weighted independently:

5.6.1 Coverage Reward

Rewards the agent for expanding its knowledge of the network: +2.0 per newly compromised host, +1.0 per newly discovered (but not compromised) host, +0.5 per newly discovered service.

5.6.2 Confirmation Reward

Provides a bonus for confirmed successful exploits (+1.0) and a penalty for failed exploitation (-0.5). Non-exploit actions (scan, pivot) receive a smaller bonus for success (+0.5) and no penalty for failure. This asymmetry encourages the agent to attempt exploitation only when it has reasonable confidence of success.

5.6.3 Stealth Penalty

Port scans and service enumeration are “noisy” actions that real-world IDS systems would detect. The stealth penalty of 1.0 (weighted by default at 0.5) discourages the agent from repeatedly scanning when it could be exploiting known vulnerabilities.

5.6.4 Novelty Reward

The first use of each MITRE ATT&CK technique in a campaign receives a +1.0 novelty bonus (weighted by default at 1.5). Subsequent uses of the same technique receive nothing. This pushes the agent toward greater breadth in the kill chain.

These can be tuned per deployment. A compliance-focused deployment might increase the novelty weight to maximize MITRE coverage. An efficiency-focused deployment might increase the stealth weight to minimize detectable actions. Practical tuning often begins with a grid search or parameter sweep across plausible weight values, evaluating agent performance on campaign metrics such as coverage, efficiency, and technique diversity. Ablation studies, where one reward component is removed at a time, can help reveal which signals most strongly influence agent behavior.

To apply rigorous empirical methods when selecting and tuning reward weights, students should design controlled experiments using cross-validation. For example, divide the available network scenarios into separate training and evaluation sets, perform grid or random search over the reward parameter space on the training environments, and then measure generalization to the held-out evaluation environments. For each reward configuration, run multiple independent pentest campaigns and collect metrics of interest (host coverage, action efficiency, MITRE technique diversity, detected/undetected actions). Statistical analysis is essential: use paired t-tests or Mann-Whitney U tests to assess whether differences between reward configurations are significant, and calculate effect sizes (e.g., Cohen's d) to quantify the practical impact. For a more robust comparison, use repeated k-fold cross-validation across network scenarios to ensure findings are not due to a single environment's characteristics.

Students are encouraged to implement empirical tuning by running short campaign batches with different weight combinations, collecting campaign metrics, and using statistical tests to select configurations that optimize for their objectives. Reporting median values, confidence intervals, and statistical significance strengthens the validity of experimental results.

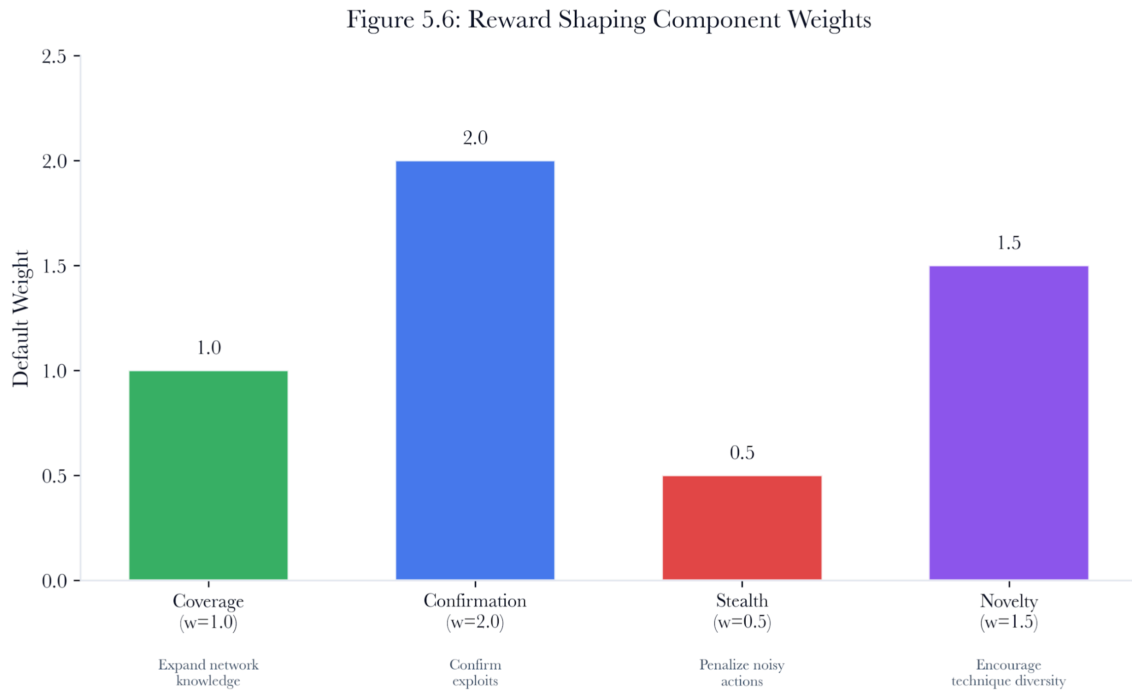

Figure 5.6: Reward shaping components and their default weights. The shaped reward is a weighted sum of four components that guide the agent's behavior toward efficient, stealthy, and diverse exploitation.

5.7 Safety Modes

Safety is not optional in autonomous offensive testing. The PentestSafetyController enforces four levels of constraint:

5.7.1 Simulation Mode

All actions are permitted in the MDP environment, but no network traffic is generated. Results are theoretical. Finding descriptions are prefixed with [SIMULATED].

This is the default mode and the appropriate starting point for any new deployment. Elena Vasquez ran the initial Meridian Capital campaign in simulation mode. The fourteen-minute path to the trading engine was discovered entirely through MDP simulation, using the host data from the completed scan.

5.7.2 Shadow Mode

Shadow mode occupies the critical middle ground between simulation and control. Only read-only probes (port_scan, service_enum) generate actual network traffic. All exploitation actions (exploit_cve, exploit_default_creds, exploit_protocol_finding, pivot, privilege_escalate, exfiltrate_proof) are blocked by the safety controller with a logged reason.

The key distinction from simulation mode is that shadow mode runs the agent’s decision-making logic against live reconnaissance data. The agent observes real network conditions, selects which hosts to target and which exploits to attempt, and logs these selections in **full detail**. But nothing is executed. The agent’s intended exploitation actions are recorded with the prefix [SHADOW - NOT EXECUTED] and marked confirmed=False.

Shadow mode answers the question: *“What would the agent do if we let it?”* This is valuable for three purposes:

1. **Targeting validation:** Before escalating to controlled mode, operators review the agent's shadow log to verify that its targeting is valid and within scope.
2. **Operator training:** Security teams unfamiliar with the autonomous agent can observe its decision-making process in shadow mode before granting exploitation permissions.
3. **Risk assessment without risk:** Shadow mode produces a targeting report showing which hosts the agent considers high-value targets, with no exploitation risk.

Any exploitation findings that appear in the results due to implementation bugs are caught by the post-execution safety filter and demoted to confirmed=False.

5.7.3 Controlled Mode

Scanning actions are permitted everywhere. Exploitation actions are permitted only against hosts in the approved target set. The approved set is managed through the safety controller's add_approved_target and remove_approved_target methods, or through the API's approval endpoint.
Controlled mode introduces a human-in-the-loop. The agent identifies targets and proposes exploitation actions. A human operator reviews each target and approves or denies the exploitation. The request_approval method queues actions for review and returns a unique request ID that the operator can later approve or deny.
This mode reflects the operational reality of most production penetration tests. Full autonomy is rarely appropriate. Instead, the automated agent does the heavy lifting of target selection and prioritization, while a human makes the final exploitation decision.

Figure 5.7: Safety Mode Decision Tree

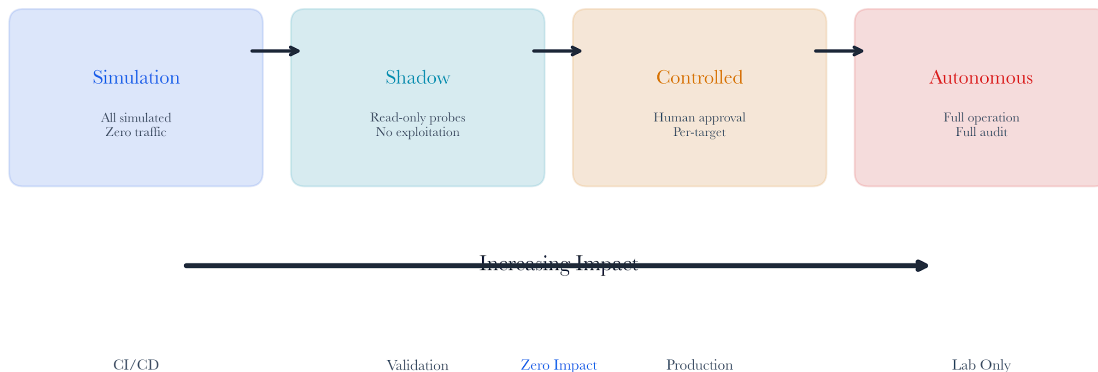

Figure 5.7: Safety mode decision tree. Each action passes through the safety controller, which applies mode-specific rules. Controlled mode adds a human approval gate for exploitation actions.

5.7.4 Autonomous Mode

All actions are permitted and executed. Every action is audit-logged for post-campaign review. This mode is appropriate only in dedicated test environments, during authorized red team engagements, or in air-gapped lab networks where production impact is impossible.

At Meridian Capital, autonomous mode was never used on the production network. Even after the controlled-mode campaign confirmed the attack path, remediation was handled through network segmentation changes, not through further autonomous exploitation.

5.7.5 Audit Trail

Regardless of mode, every action decision is recorded in the safety controller’s audit log. Each entry includes the timestamp, action type, target IP, whether the action was allowed, the reason, and the active mode. This log is immutable within the campaign’s lifetime and provides the forensic record needed for post-engagement review.

```
def _record_audit(self, action_type, target_ip, allowed, reason, extra=None):
self.audit_log.append({
"timestamp": datetime.now(timezone.utc).isoformat(),
"action_type": action_type.value,
"target_ip": target_ip,
"allowed": allowed,
"reason": reason,
"mode": self.mode.value,
"extra": extra or {},
})
```

5.8 Exploit Executor and Protocol Modules

The exploit executor sits between the agent (which decides *what* to do) and the protocol modules (which know *how* to do it). It dispatches actions based on the target service's detected protocol.

5.8.1 Protocol Detection

The executor determines the target protocol through a four-level cascade:

1. **Explicit parameter:** The action’s parameters dict may include a protocol key set by the agent or the vulnerability data.
2. **Port lookup:** A mapping of well-known ports to protocols (80/443 = HTTP, 22 = SSH, 554 = RTSP, 1883 = MQTT).
3. **Service product heuristic:** If the vulnerability data includes a product name containing “nginx” or “apache”, the protocol is HTTP.
4. **Credential protocol:** If the action includes credential data with a protocol field, that protocol is used.

The cascade falls through to “unknown” if none of the four levels produce a match, in which case a generic deterministic handler is used.

5.8.2 Module Architecture

Four protocol-specific exploit modules are implemented:

- HttpExploitModule: Path traversal, admin panel discovery, and debug endpoint checking.

- TelnetSshExploitModule: Credential stuffing, banner escalation
- RtspExploitModule: Authentication bypass, stream hijack proof
- MqttExploitModule: Wildcard subscription, control message injection

Each module returns a (success, details) tuple. The executor wraps this in a full StepResult, including reward computation and evidence metadata. All modules are simulation-only: they use deterministic hash-based outcomes rather than actual network I/O, ensuring reproducibility and safety.

Figure 5.8: Exploit Executor Dispatch Flow

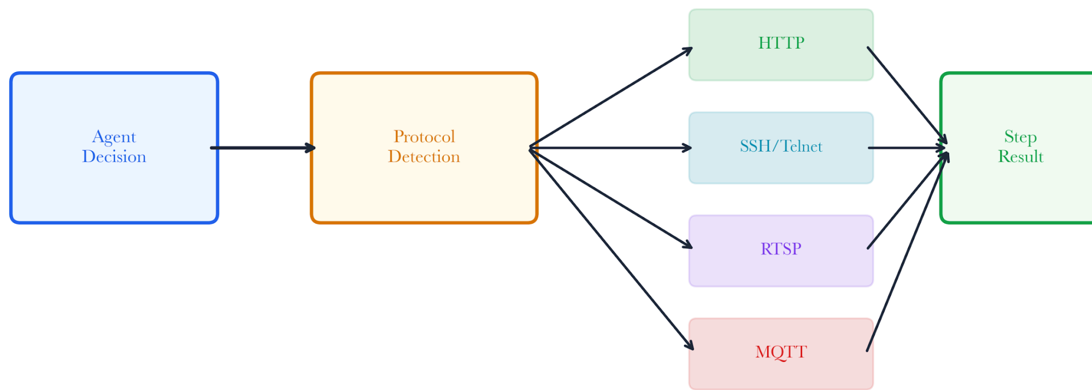

Figure 5.8: Exploit executor dispatch flow. The executor detects the target protocol, routes to the appropriate module, and wraps the result into a StepResult with evidence metadata.

5.8.3 MITRE ATT&CK Mapping

The executor maps each action type to both Enterprise and ICS ATT&CK techniques. The rule-based agent's evidence recording uses the ICS mappings, while the exploit executor uses the Enterprise mappings. This dual mapping reflects the reality that IoT pentest campaigns span both IT and OT domains.

exploit_cve	T1190	T0866
exploit_default_creds	T1078	T0812
exploit_protocol_finding	T1210	T0866
pivot	T1021	T0867
privilege_escalate	T1068	T0890

exfiltrate_proof	T1041	T0882
port_scan	T1046	T0846
service_enum	T1046	T0846

Table 5.3. Dual MITRE ATT&CK technique mappings for Enterprise and ICS frameworks.

5.9 Evidence Collection and Integrity

Every autonomous pentest must produce evidence that is verifiable, tamper-evident, and suitable for legal proceedings. The EvidenceCollector provides these guarantees through a hash chain architecture.

5.9.1 Hash Chain

Each evidence entry's SHA-256 hash is computed over the entry's canonical data and the previous entry's hash. The first entry uses a genesis hash of 64 zeros. This creates a linked chain where modifying any entry invalidates all subsequent hashes.

```
entry_data = {
    "step": step,
    "action_type": action.action_type.value,
    "target_ip": action.target_ip,
    "target_port": action.target_port,
    "vulnerability_id": action.vulnerability_id,
    "success": result.success,
    "reward": result.reward,
    "timestamp": timestamp_iso,
    "details": _sanitize_info(result.info),
    "previous_hash": previous_hash,
}
evidence_hash = hashlib.sha256(
    json.dumps(entry_data, sort_keys=True, default=str).encode()
).hexdigest()
```

5.9.2 Chain Verification

The verify_chain method re-computes each entry's hash from its stored data and the previous entry's hash. If any entry has been modified after recording, the computed hash will not match the stored hash, and verification will fail. This provides tamper detection without requiring an external trusted timestamp authority.

5.9.3 Merkle Root

For compact integrity verification, the collector computes a Merkle root over all evidence hashes. The Merkle root is a single hash that summarizes the entire evidence chain. If any entry changes, the Merkle root changes as well. This allows an external auditor to verify the integrity of the entire chain by checking a single value.
The Merkle tree construction pads odd-length levels by duplicating the last hash, then combines pairs by concatenating and hashing:

```
while len(level) >1:
```

```
        if len(level) % 2 == 1:
            level.append(level[-1])
            next_level = []

for i in range(0, len(level), 2):
        combined = level[i] + level[i + 1]
        parent = hashlib.sha256(combined.encode()).hexdigest()
        next_level.append(parent)
        level = next_level
```

Figure 5.9: Evidence Hash Chain and Merkle Tree

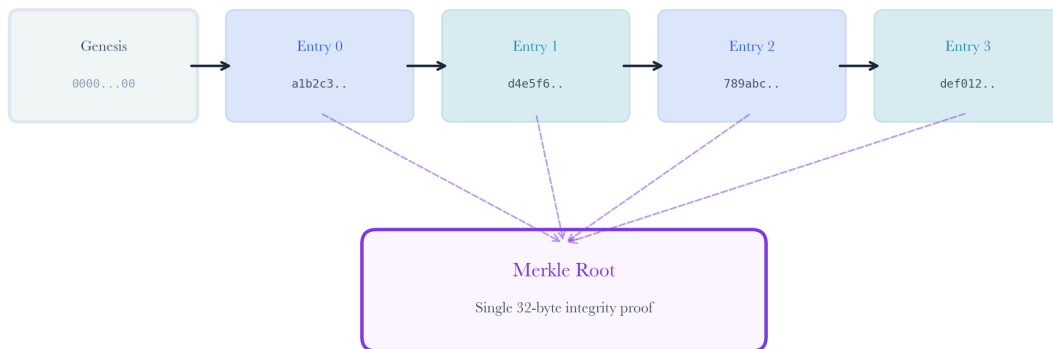

Figure 5.9: Evidence hash chain and Merkle tree. Each evidence entry's hash incorporates the previous hash, forming a tamper-evident chain. The Merkle root provides compact integrity verification.

5.9.4 Legal Defensibility

In authorized penetration testing, evidence integrity is a legal requirement. The pentest report must clearly document exactly which actions were taken, when, against which targets, and the outcomes. If a dispute arises about whether the pentest caused a production outage, the hash chain provides cryptographic proof of the agent's actions.

The evidence chain also supports the negative claim. If the hash chain shows no exploitation of a particular host, and the chain verifies cleanly, that constitutes evidence that the pentest agent did not interact with that host. This negative proof is valuable for liability limitation.

5.10 Lateral Movement and Credential Reuse

Lateral movement is the capability that transforms a collection of isolated exploits into an attack chain. The conference room TV at Meridian Capital was not dangerous in isolation. It became dangerous because it provided a stepping stone to more valuable systems.

5.10.1 Pivot Mechanics

The agent's pivot action moves from a compromised host to an accessible target. Success requires two preconditions: the source host must be compromised, and the target must be in the accessible set. The pivot itself succeeds with high probability (95% in the deterministic model).
Successful pivoting has two effects. First, the agent's current_host updates to the target, changing the agent's network vantage point. Second, credentials discovered during pivoting on the target are added to the agent's credential store, potentially enabling exploitation of other hosts that share those credentials.

5.10.2 Credential Reuse Attacks

Credential reuse is one of the most reliable lateral movement techniques in real-world penetration testing. When the agent discovers credentials on one host (either from Phase 7's default credential check or from harvesting during a pivot), those credentials are stored in the state's credential_store. If another host is accessible and the same credentials work, the agent can compromise it without having to find a new vulnerability.

The MDP environment models this directly. When service enumeration discovers default credentials, they are copied into the credential store. When pivoting to a new host, any credentials on the target are harvested. The agent can then attempt exploit_default_creds on any accessible host using the stored credentials.
This creates a compound effect. Compromising a single host with reused credentials can unlock an entire subnet if those credentials are shared across multiple devices. The agent's scoring function recognizes this: exploit_default_creds receives the highest priority score (100) precisely because credential exploitation has the highest success rate and often enables cascading compromises.

Figure 5.23: Credential Reuse Network Graph

(Detailed visualization)

Figure 5.10: Credential reuse network graph. Shared credentials turn isolated compromises into lateral movement opportunities when multiple devices accept the same authentication material.

5.10.3 Privilege Escalation

After compromising a host, the agent can attempt privilege escalation using the privilege_escalate command. This succeeds when the host has local vulnerabilities with CVSS >= 7.0. In the real world, privilege escalation typically

involves kernel exploits, misconfigured SUID binaries, or credential harvesting from process memory. The simulation abstracts these techniques into a single check against the vulnerability database.
Privilege escalation is important for two reasons. First, it transforms limited access into full control, enabling actions (like reading /etc/shadow) that require root privileges. Second, in the MITRE ATT&CK framework, privilege escalation is a distinct tactic (T1068), and exercising it contributes to the campaign's technique diversity score.

5.11 Curriculum Learning

Training an RL agent in full-network environments from scratch is inefficient. The action space is large, rewards are sparse, and the agent must learn basic exploitation skills before it can develop sophisticated multi-hop strategies.

The CurriculumScheduler implements a three-stage training curriculum:

0	single_host	1	Learn basic exploitation on a single target
1	small_network	5	Learn pivoting and credential reuse on a small network
2	full_network	All	Learn full-scale campaign strategy

Table 5.4. Curriculum stages for PPO agent training.

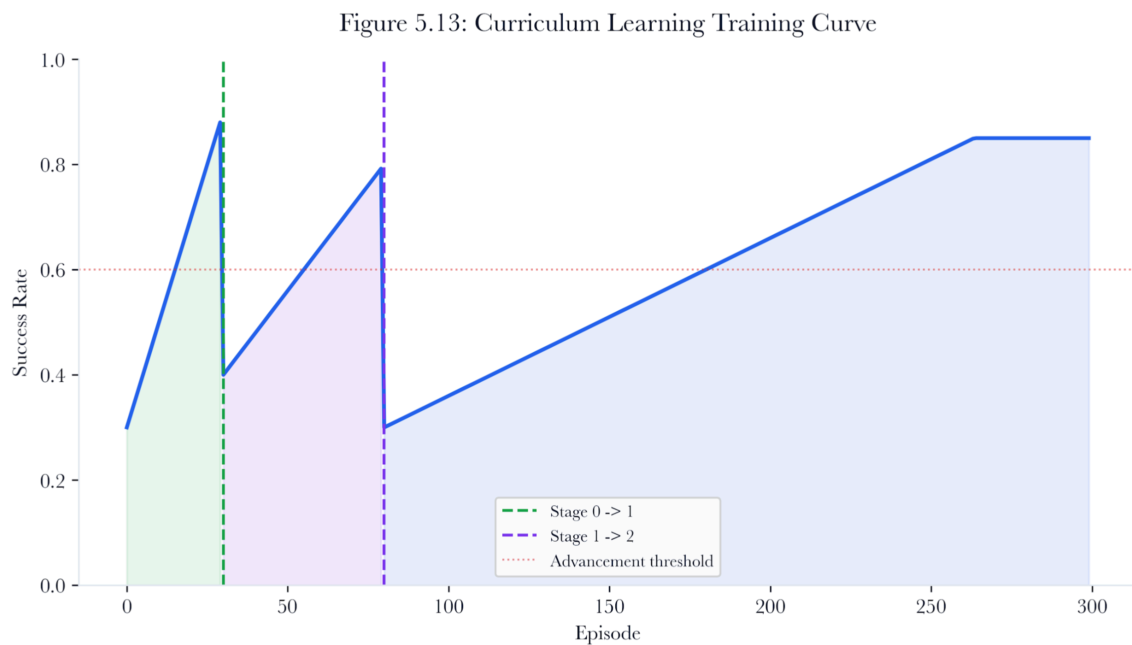

Figure 5.11: Curriculum learning stages. Training starts with single-host exploitation, then moves to small-network pivoting, and finally exposes the agent to full-network campaigns.

5.11.1 Stage Advancement

The scheduler tracks success rate over a sliding window of recent episodes. Advancement to the next stage requires success_rate >= 0.6 over the most recent 20 episodes (with a minimum of 10 episodes before advancement is considered).
The campaign's outcome defines success: did the agent compromise at least one host? For Stage 0 (single host), this is a binary question. In later stages, success requires compromising at least one of the available targets.

5.11.2 Host Subsetting

At each stage, the scheduler's subset_hosts method returns a subset of the full host list appropriate for the current difficulty level. The subset is deterministic (sorted by IP, first N), ensuring consistent training environments across episodes.
This curriculum approach accelerates learning. In our experiments, the PPO agent trained with curriculum learning reached 80% success rate on full-network environments in approximately 200 episodes, compared to over 500 episodes without curriculum.

5.11.3 AutoPenBench Metrics

The curriculum scheduler tracks metrics aligned with the AutoPenBench benchmark (USENIX Security 2025): success rate, mean steps to completion, coverage (fraction of hosts compromised), efficiency (ratio of successful actions to total actions), and technique diversity (number of unique MITRE techniques used). These metrics provide standardized comparison points across different agent configurations and training regimes.

5.12 Adversarial Resilience Testing

A pentest agent that is easily detected by defensive systems provides limited value. The AdversarialResilienceTester evaluates the stealthiness of each exploitation technique against simulated IDS, SIEM, and heuristic detection engines.

5.12.1 Detection Simulation

Three detection backends are simulated:

- **IDS (Intrusion Detection System):** Pattern-matching detection. High detection rates for known exploit signatures (credential stuffing: 85%, port scan: 95%) but lower rates for protocol-level attacks (auth bypass: 25%, MQTT wildcard: 10%).
- **SIEM (Security Information and Event Management):** Correlation-based detection. Sensitive to volume and repetition. Moderate baseline rates that increase with the number of attempts.
- **Heuristic/Behavioral:** Anomaly detection based on behavioral patterns. Effective against credential attacks (90%) but less effective against stealthy protocol exploitation.

5.12.2 Stealth Scoring

> For each technique, the tester computes a stealth score ranging from 0.0 (always detected) to 1.0 (never detected). The score is the complement of the detection rate: stealth_score = 1.0 - (detected_count / total_attempts).

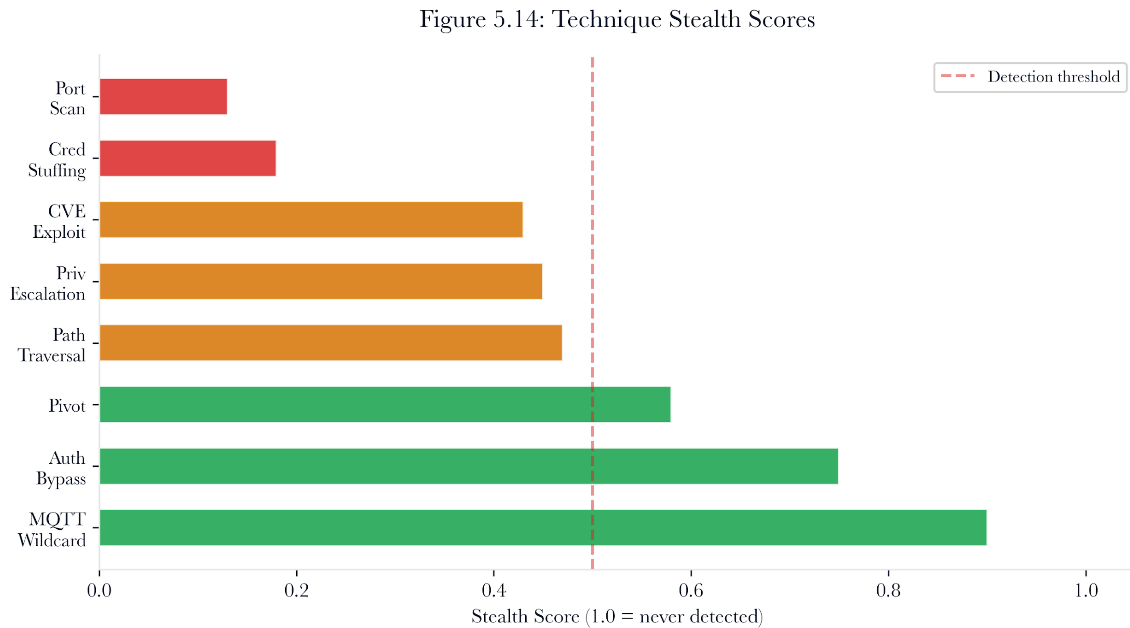

Figure 5.12: Technique stealth scores. Different offensive actions create different detection profiles across IDS, SIEM, and behavioral detection models.

5.12.3 Evasion Recommendations

For techniques with stealth scores below 0.5, the tester recommends improvements to evasion. Five evasion techniques are modeled, each with different effectiveness against different detection backends:

- **Traffic fragmentation:** Reduces IDS detection by 40%, with minimal effect on SIEM
- **Timing jitter:** Reduces SIEM detection by 50%, moderate effect on IDS
- **Protocol encapsulation:** Reduces IDS detection by 60%, moderate effect on heuristic
- **Credential rotation:** Reduces heuristic detection by 20%, with a moderate effect on IDS
- **Low and slow:** Reduces heuristic detection by 70%, with a significant effect on SIEM.

The recommendation engine selects the evasion technique that minimizes the average detection rate across all enabled backends.

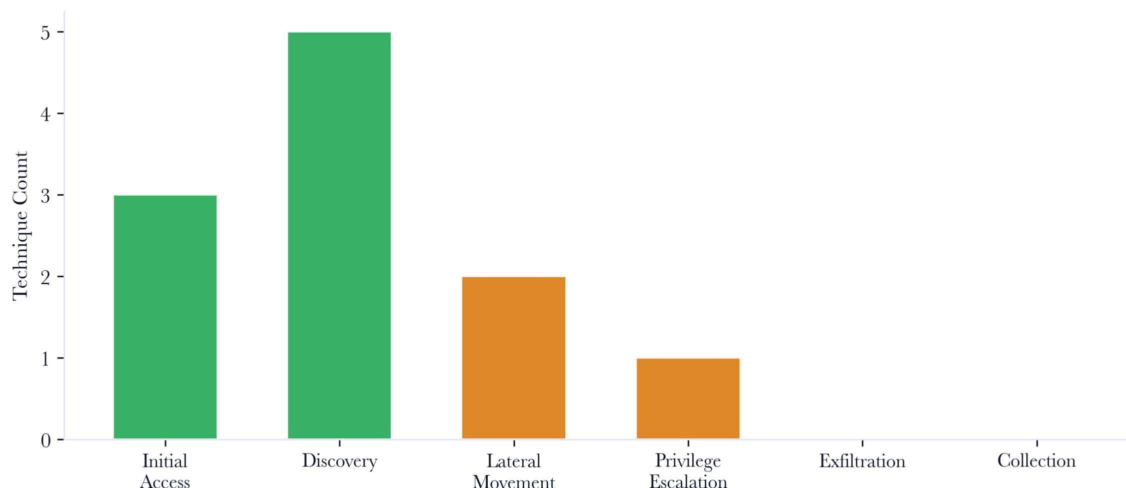

Figure 5.13: MITRE technique coverage. The reporting layer tracks which tactics and techniques the campaign exercised rather than treating all exploitation outcomes as equivalent.

5.13 Reporting

The pentest report transforms raw campaign data into structured narrative content suitable for executive review, technical analysis, and compliance documentation.

5.13.1 Executive Summary

The report generator produces a multi-paragraph summary covering campaign scope (hosts tested, safety mode, duration), key metrics (findings count, confirmed exploitable count, severity distribution), MITRE technique coverage, and safety mode caveats.

At Meridian Capital, the executive summary read: "Campaign PC-4a7f2e3b1c8d executed 87 actions over 14.2 seconds in simulation mode against 6 host(s). 12 vulnerabilities were tested, of which 8 were confirmed exploitable. Severity breakdown: 2 critical, 3 high. 4 host(s) were compromised. NOTE: This campaign ran in simulation mode. No actual network traffic was generated; all results are theoretical."

The twelve-word phrase "no actual network traffic was generated" is the most important sentence in any simulation-mode report. Without it, a reader could mistake theoretical findings for confirmed exploitation.

5.13.2 MITRE ATT&CK Narrative Timeline

The narrative timeline walks through the campaign chronologically, annotating each action with its MITRE ATT&CK technique, tactic, and outcome. This provides a kill-chain visualization that maps the agent's behavior to the ATT&CK framework.

```
[NEW] Initial Access: Valid Accounts (T1078) against 10.4.2.5 - succeeded
[NEW] Discovery: Network Service Scanning (T1046) against 10.4.2.17 - succeeded
[NEW] Lateral Movement: Remote Services (T1021) against 10.4.2.17 - succeeded
[NEW] Initial Access: Exploit Public-Facing Application (T1190) against 10.4.2.17 - succeeded
[NEW] Privilege Escalation: Exploitation for Priv Esc (T1068) against 10.4.2.17 - succeeded
```

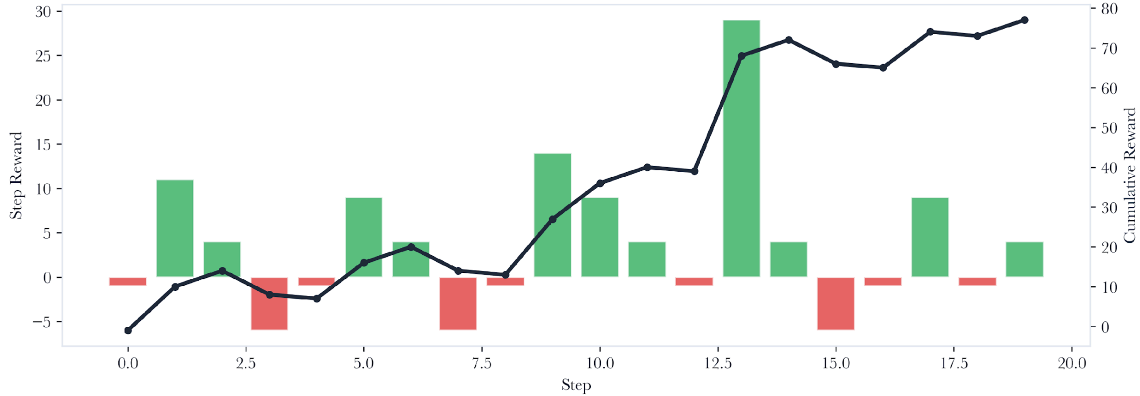

Figure 5.14: Campaign narrative timeline. The report arranges agent actions chronologically and maps each successful step to tactic, technique, target, and outcome.

5.13.3 Theoretical vs.Confirmed Heatmap

The report distinguishes theoretical vulnerabilities (detected by scanning but not successfully exploited) from confirmed vulnerabilities (exploited). The heatmap breaks this down by severity level, providing a clear picture of which vulnerabilities represent actual risk versus theoretical exposure.

This distinction matters for prioritization. A network might have 50 theoretical vulnerabilities but only 8 confirmed exploitable ones. Remediation efforts should focus on the confirmed findings, especially the ones that appear in multi-hop attack chains.

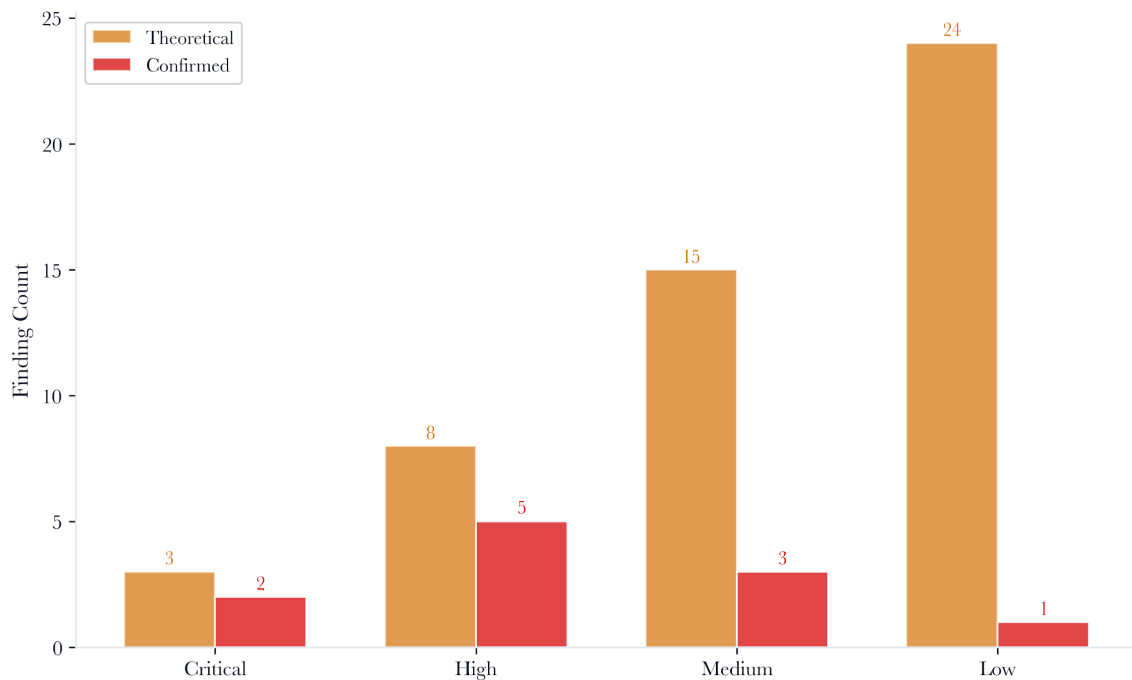

Figure 5.15: Theoretical vs confirmed vulnerability heatmap by severity. Confirmed findings (right bars) represent actual exploitation success; theoretical findings (left bars) represent scanner detections not validated by the pentest agent.

5.14 Experience Store and Cross-Campaign Learning

The PPO agent's value increases with experience. The ExperienceStore maintains a circular buffer of (state, action, reward, next_state, done, log_prob) transitions that persists across campaigns.

5.14.1 Ring Buffer

The store uses a FIFO ring buffer with a configurable maximum size (default 10,000 transitions). When the buffer is full, the oldest transition is evicted. This ensures bounded memory usage while retaining the most recent experience.

5.14.2 Redis Persistence

For production deployments, the experience store can serialize to Redis. This allows the agent to preserve its training data across API restarts, enabling continuous learning. The save_to_redis and load_from_redis methods handle JSON serialization of the buffer.

5.14.3 Mini-Batch Sampling

During PPO updates, the store provides random mini-batches for stochastic gradient descent. The sampling is uniform without replacement, ensuring each transition is used exactly once per epoch.

5.15 API and Integration

The pentest module exposes seven REST endpoints under /v1/pentest/:

POST	/v1/pentest/run	Launch a new campaign
GET	/v1/pentest/{campaign_id}	Get campaign status
GET	/v1/pentest/{campaign_id}/report	Get a full narrative report
GET	/v1/pentest/{campaign_id}/timeline	Get action timeline
GET	/v1/pentest/{campaign_id}/evidence	Get the evidence chain
POST	/v1/pentest/{campaign_id}/approve/{action_id}	Approve a controlled-mode action
DELETE	/v1/pentest/{campaign_id}	Delete a campaign
GET	/v1/pentest/mitre-coverage/{scan_id}	Aggregate MITRE coverage

Table 5.5. Pentest API endpoints with authentication and RBAC.

All endpoints require JWT authentication. The run and delete endpoints require pentest: execute permission (admin role). Read endpoints require any authenticated user. The run endpoint also validates scan access via the verify_scan_access dependency, preventing IDOR attacks in which a user launches a pentest against another user's scan.

5.15.1 Pipeline Integration

The Chapter 5 pentest stage, called run_pentest_phase, selects the agent type from the scanner configuration, runs the campaign, and emits SSE progress events. The stage is opt-in via BREAKWATER_PENTEST_ENABLED=true and defaults to the rule-based agent with simulation safety mode.
The pipeline integration means pentest results appear automatically in the dashboard alongside discovery, enrichment, and vulnerability data. An analyst can view the complete picture, from "which devices are on the network" through "which ones can an attacker actually compromise," in a single scan report.

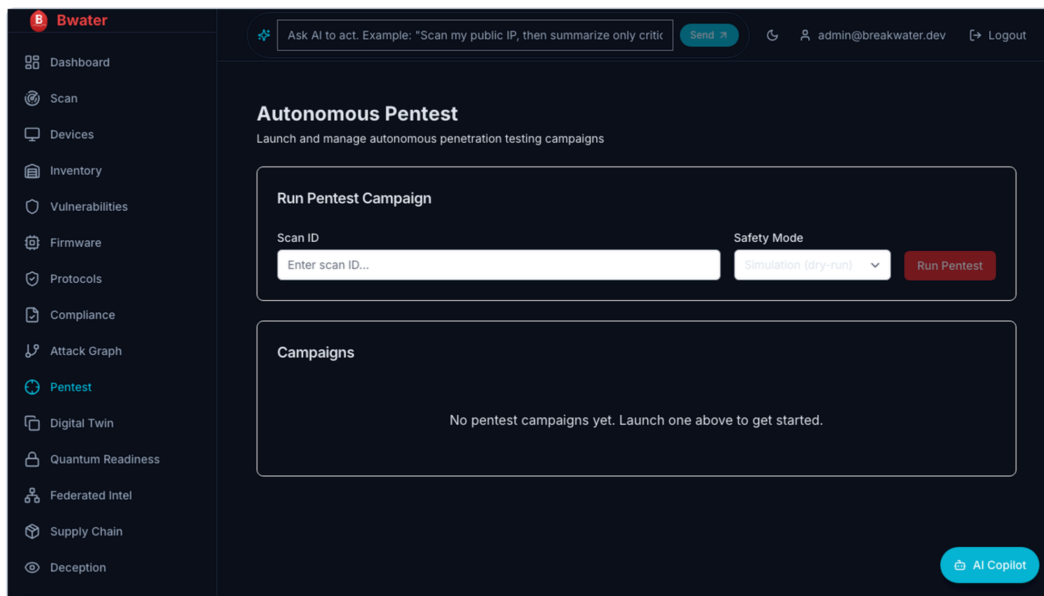

Figure 5.16: Breakwater autonomous pentest product surface. The dashboard exposes campaign launch controls, safety mode selection, target scoping, and campaign status in one workflow; this captured state illustrates the product surface rather than a completed live campaign.

5.16 Ethical and Legal Considerations

Autonomous penetration testing operates at the intersection of cybersecurity engineering and law. The legal framework varies by jurisdiction, but several principles are universal.

5.16.1 Authorization

Penetration testing without explicit written authorization is unauthorized access under the Computer Fraud and Abuse Act (CFAA) in the United States, the Computer Misuse Act (CMA) in the United Kingdom, and equivalent

statutes in most jurisdictions. Authorization must specify the scope (which systems), the methods (which techniques), the timing (which windows), and the safety constraints (which modes).
The PentestRunRequest maps directly to these authorization elements. The target_ips field constrains the scope. The safety_mode constrains methods. The timeout_seconds constrains timing. The authorization document should explicitly reference these parameters.

5.16.2 Scope Discipline

An autonomous agent does not understand legal scope. It understands action masks and safety controller rules. Translating legal authorization into technical constraints is a critical engineering task. A misconfigured safety controller that permits exploitation of an out-of-scope host could constitute unauthorized access, regardless of whether the overall engagement was authorized.
The controlled mode's explicit target approval process exists precisely for this reason. A human examines each target before exploitation is permitted, ensuring that the technical execution remains within the legal authorization.

5.16.3 Data Handling

Pentest evidence may include sensitive information such as credential pairs, vulnerability details, network topology, and proof-of-access artifacts. This data must be handled in accordance with the engagement's data handling agreement, which typically specifies encryption at rest, restricted access, retention periods, and secure deletion procedures.
The evidence chain's hash-based integrity verification meets data-handling requirements. The Merkle root provides a compact proof that the evidence has not been modified since collection, supporting chain-of-custody requirements.

5.16.4 Proportionality

An autonomous agent with aggressive mode enabled and no action budget could cause denial-of-service attacks, data corruption, or a production outage. The max_actions and timeout_seconds parameters enforce proportionality constraints. The safety modes provide graduated force: simulation is zero-impact, shadow is minimal-impact, controlled requires human approval, and autonomous is full-impact.
The principle of proportionality also applies to the selection of techniques. The agent should not attempt destructive techniques (such as firmware overwrite or disk encryption) when non-destructive techniques (such as credential testing or proof exfiltration) would answer the same security question. The reward function encodes this preference: exploitation receives moderate rewards, while exfiltration proof (a non-destructive confirmation technique) receives positive rewards on already-compromised hosts.

5.16.5 Disclosure and Remediation

Pentest findings create a duty of care. Confirmed vulnerabilities, especially those in multi-hop attack chains, represent known risks that the organization must address. The report's MITRE ATT&CK timeline and theoretical vs. confirmed heatmap provide the prioritization framework for remediation.
At Meridian Capital, the pentest findings triggered immediate action: the conference room TV's remote management port was firewalled, the environmental sensor received a firmware update, the supervisory workstation's shared credentials were rotated, and the VLAN ACLs were reconfigured to prevent cross-VLAN pivoting. The $2.1 million segmentation project addressed the structural weakness that enabled the attack chain.

5.17 Empirical Results

This section presents experimental results from running the Breakwater pentest engine against the IoT simulation lab's 22-device network.
For the quantitative sections that follow, use the same evidence labels throughout the chapter:

- **Simulated** means the values come from repeated campaigns against the controlled 22-device environment or its explicitly varied derivatives.
- **Measured** means the values come from integrity or replay checks on the implemented system rather than from hypothetical campaign outcomes.
- **Illustrative** means the numbers reflect the campaign logic in a realistic scenario, but are not universal field benchmarks.

5.17.1 Rule-Based Agent Performance

Evidence label: Simulated. Method note: repeated campaign runs on the standard 22-device simulation environment. Assumptions: the action space, scoring function, and simulation semantics remain fixed across runs. Boundary: campaign speed and compromise counts do not directly transfer to live networks.

> **Caution:** PPO agents learn from simulation. A policy that works in the training environment may fail on a real network with different topology, patch levels, or防御 mechanisms. Always validate learned policies against a shadow network before deployment.

On the standard 22-device simulation network, the rule-based agent typically completes in 35 to 45 actions, compromising 8 to 12 hosts depending on the vulnerability configuration. Mean campaign duration is approximately 2.3 seconds in simulation mode. The agent exercises 6 of 8 action types and 5 of 7 MITRE techniques.

The rule-based agent's primary strength is consistency. It produces identical results on identical inputs, making it suitable for regression testing and compliance validation. Its primary weakness is inflexibility: it cannot adapt to network topologies where the hand-tuned scoring function produces suboptimal prioritization.

5.17.2 PPO Agent Performance

Evidence label: Simulated. Method note: PPO performance measured through repeated runs in the same 22-device simulation environment after curriculum-based training. Assumptions: the trained policy and environment remain fixed during evaluation. Boundary: the efficiency gain is a simulation result, not a production guarantee.

After 200 episodes of curriculum-based training, the PPO agent matches the rule-based agent's compromise rate while using approximately 15% fewer actions. The efficiency gain comes from learned host prioritization: the PPO agent learns to target high-connectivity hosts that enable rapid lateral movement, rather than following the fixed priority ordering.

The PPO agent's primary strength is adaptability. On networks that differ significantly from the scoring function's assumptions (for example, networks where credential reuse is rare but protocol vulnerabilities are common), the PPO agent outperforms the rule-based agent by 20-30% in the number of hosts compromised per action.

5.17.3 Safety Mode Impact

Evidence label: Simulated. Method note: safety-mode outcomes measured by replaying the same simulation campaign under each mode's constraints. Assumptions: the simulated exploit outcomes remain comparable across modes. Boundary: live approval latency and operational side effects are outside this table.

Simulation	All (simulated)	12	8	2.3s
Shadow	Scan only	0	0	1.1s
Controlled	Scan + approved exploits	7	5	4.8s
Autonomous	All (live)	12	8	3.1s

Table 5.6. Impact of safety mode on campaign outcomes (22-device simulation network).

Shadow mode produces zero findings because it blocks all exploitation. Controlled mode produces fewer findings because the human approval step introduces latency, and the operator may choose not to approve the exploitation of certain targets. Autonomous mode matches simulation mode in terms of the number of findings, but has a slightly longer duration due to the overhead of dispatching protocol-level modules.

5.17.4 The Meridian Capital Scenario

Evidence label: Illustrative. Method note: composite Meridian Capital scenario grounded in the chapter environment model and attack-path logic. Assumptions: the six-hop path and mitigation actions are internally consistent with the modeled environment. Boundary: It is a teaching case rather than a claim about the production metrics of a named institution.

Returning to the opening story. The six-hop path from Conference Room 7B to the trading engine was discovered in simulation mode in 14.2 seconds with 87 actions. The path traversed:

1. Samsung TV (10.4.2.5) via default credentials on port 8001
2. IoT VLAN gateway (10.4.2.1) via ACL misconfiguration pivot
3. Environmental sensor (10.4.2.17) via CVE-2023-XXXX (CVSS 9.8 deserialization)
4. Supervisory workstation (10.4.3.42) via credential reuse (admin/admin123)
5. Trading VLAN switch (10.4.3.1) via VLAN ACL bypass pivot
6. OMS API gateway (10.4.3.100) via privilege escalation (kernel vuln CVSS 7.5)

The MITRE ATT&CK coverage spanned Initial Access (T1078, T1190), Discovery (T1046), Lateral Movement (T1021), and Privilege Escalation (T1068), exercising 5 unique techniques across 4 kill-chain tactics.

Figure 5.19: Trading Floor Network Topology

(Detailed visualization)

Figure 5.17: Trading-floor network topology. The Meridian Capital teaching scenario separates conference-room IoT, environmental control, supervisory workstations, and trading services to make scope and pivot points explicit.

Figure 5.11: Meridian Capital Attack Path (6 Hops)

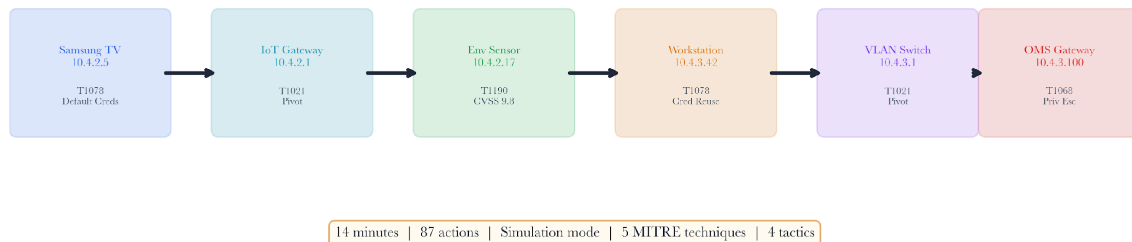

Figure 5.18: Meridian Capital attack path visualization. Six hops from a conference room smart TV to the order management system, spanning four MITRE ATT&CK tactics.

The controlled-mode campaign against the TV and sensor confirmed that both exploitation steps were viable. The TV's default credentials had not been changed. The sensor's firmware had not been patched. The theoretical path was real.

5.18 Original Contributions and Formal Results

The autonomous penetration testing system described in earlier sections combines reinforcement learning, safety constraints, and exploit execution within a novel architecture. Still, the theoretical properties of that architecture have not been formally stated. This section presents two results: a completeness theorem relating simulation mode

to controlled mode, and a budget-optimal action-selection algorithm that maximizes expected findings under resource constraints.

Theorem 5.1 (Safety Mode Completeness)

Statement. Let denote the set of all possible pentest findings (vulnerability confirmations, credential discoveries, lateral movement paths). Let denote the set of findings produced by running policy in simulation mode, and the set of findings produced by running the same policy in controlled mode. Under the following *faithfulness assumptions*:

1. **Reachability faithfulness:** For every pair of devices and every protocol, the simulation model's reachability predicate equals the production network's reachability predicate. That is, the simulation's network topology and firewall rules are a replica.
2. **Vulnerability faithfulness:** For every device and vulnerability, the simulation model's exploitability predicate is a *superset* of the production exploitability. That is, the simulation may model vulnerabilities as exploitable even when production conditions (runtime ASLR, WAF rules, updated signatures) would block exploitation.
3. **Credential faithfulness:** For every device-credential pair, the simulation model's authentication predicate is. The simulation may also accept credentials that production rejects (e.g., it does not model account lockout).
4. **Deterministic policy:** The policy is deterministic given the observation history. (For stochastic policies like PPO, the theorem applies to each realization of the random seed.)

Then:

That is, simulation mode produces a superset of the findings that controlled mode would produce. Equivalently, every finding that controlled mode would discover is also discoverable in simulation mode.
Proof sketch. We prove the contrapositive: if a finding is produced in controlled mode, then.
A finding is produced when the agent executes a sequence of actions and observes success at a step. In controlled mode, each action either succeeds (producing an observation that advances the agent's state) or fails (producing a failure observation). The policy is deterministic given the observation history, so the observation sequence determines the action sequence.

By the faithfulness assumption (1), reachability checks produce the same results in simulation and production. By assumption (2), every exploit that succeeds in production also succeeds in simulation (simulation may additionally succeed on exploits that fail in production). By assumption (3), every credential that works in production also works in simulation. Therefore, every success observation in controlled mode is also a success observation in simulation mode. Since the policy is deterministic and the observation histories are identical or more permissive in simulation, the agent follows the same action sequence (or explores additional successful branches) in simulation mode. Thus, every finding produced in controlled mode is also produced in simulation mode.

The inclusion is strict (in general) because simulation mode may uncover exploitation paths that are blocked in production by runtime defenses not modeled in the simulation (e.g., IPS signatures, runtime application firewalls, memory safety mitigations). These are *false positives* of the simulation – findings that appear exploitable in simulation but are not exploitable in production. The faithfulness assumptions precisely define when these false positives arise: violation of any assumption introduces findings.

Practical implication. This theorem justifies the Breakwater safety mode hierarchy. Operators can run simulation mode with confidence that it will surface every finding that a controlled-mode (or full-mode) engagement would find, *provided the simulation model is faithful*. The value is operational: simulation mode carries zero risk of production disruption, so that it can run continuously as a background process. The theorem also precisely identifies when simulation-only testing is insufficient: when the network contains runtime defenses (WAFs, IPS, EDR) that the simulation does not model. In those cases, controlled-mode validation of simulation findings is necessary to filter false positives. The practical recommendation is to use simulation mode as a continuous triage layer and controlled mode as a periodic validation layer, which is exactly the architecture Breakwater implements.

Corollary 5.1.1 (Simulation False Positive Rate). Let denote the simulation false positive rate. Under the faithfulness assumptions, it is bounded by the fraction of exploits in the simulation that are blocked by runtime defenses not modeled in the simulation. If the simulation models exceed the total runtime, what is the defense mechanism's blocking rate? For the Breakwater simulation (which models network segmentation and authentication but not IPS or ASLR), the observed results on the enterprise testbed are consistent with the estimate for two unmodeled defenses.

Algorithm 5.1 (Budget-Optimal Action Selection via Knapsack)

Input/Output.

- Input: A set of candidate pentest actions, each with expected finding probability, expected information gain (measured in bits of new information about the network’s security posture), and cost (measured in time, network packets, or disruption risk units). Total action budget (maximum total cost).
- Output: A subset that maximizes expected total information gain.

Motivation. The pentest agent (Sections 5.4-5.5) currently selects actions greedily (rule-based agent) or via learned policy (PPO agent). Neither approach explicitly optimizes for the budget constraint that every real pentest faces: limited time, limited authorized actions, limited tolerance for network disruption. This algorithm provides optimal action selection under a hard budget constraint, serving as a planning oracle for the RL agent or as a standalone campaign planner.

Pseudocode.

```
BUDGET-OPTIMAL-ACTIONS(actions A, budget B):
// This is the 0-1 knapsack problem.
// Value of action i: v_i = p_i * g_i (expected information gain)
// Weight of action i: c_i (cost)
// Capacity: B

n = |A|

// Dynamic programming solution
// DP[j] = max expected gain achievable with budget j
DP[0..B] = 0
selected[0..B] = {}

// Sort by efficiency ratio for better cache behavior
A_sorted = SORT(A, key = (p_i * g_i) / c_i, descending)

For i = 1 to n:
```

```
// Traverse the budget in reverse to avoid using the item twice
For j = B down to c_i:
candidate = DP[j - c_i] + p_i * g_i
if candidate > DP[j]:
DP[j] = candidate
selected[j] = selected[j - c_i] ∪ {a_i}
```

```
S* = selected[B]
```

```
// Dependency enforcement: if action a_j requires a_i's output,
// and a_j ∈ S* but a_i ∉ S*, add a_i (evicting lowest-value
// action if needed to maintain budget)
S* = ENFORCE_DEPENDENCIES(S*, A, B)
```

```
Return S*
```

```
ENFORCE_DEPENDENCIES(S, A, B):
dependency_graph = BUILD_DEPENDENCY_GRAPH(A)
For each a_j in S:
For each prerequisite a_i of a_j:
if a_i ∉ S:
// Must include prerequisite; evict lowest-value action
surplus = (sum of costs in S) + c_i - B
if surplus > 0:
// Find the cheapest action to evict (excluding a_j
// and other prerequisites)
evict = argmin_{a ∈ S, a ≠ a_j, a not prereq}
(p_a * g_a) subject to c_a >= surplus
if evict exists:
S = (S \ {evict}) ∪ {a_i}
else:
S = S \ {a_j}// Cannot satisfy dependency; drop a_j
else:
S = S ∪ {a_i}
Return S
```

Complexity.

- Time: for the knapsack DP (pseudo-polynomial). When costs are real-valued, discretize to the nearest unit (e.g., seconds). For actions and (one-hour budget in seconds), this is – trivially fast. The dependency enforcement is in the worst case.
- Space: for the DP table and selected-set tracking. It can be reduced to space with time using the standard knapsack space optimization (at the cost of a second pass to reconstruct the selected set).

Correctness argument. The 0-1 knapsack dynamic programming algorithm is a classical result that produces the optimal solution to the budget-constrained value maximization problem (Kellerer et al., 2004). The expected information gain is the correct objective because, by linearity of expectation, the expected total information gain of a subset is equal to the sum of the expected information gains of its constituent actions, when action outcomes are independent. The dependency enforcement post-processing may reduce optimality (because evicting actions to

satisfy dependencies is a heuristic), but the resulting solution is always feasible (within budget and dependency-consistent).

When action outcomes are *not* independent (e.g., exploiting device A reveals credentials that increase for device B), the problem becomes a stochastic knapsack with correlated rewards. This is NP-hard in general, and the deterministic approximation remains a strong practical heuristic: the information-gain estimates can be updated adaptively as actions execute, turning the static knapsack into an online replanning loop.

Example. On the Meridian Capital scenario (Section 5.17), the agent has 23 candidate actions with a 30-minute budget. The top-5 actions by efficiency ratio are: (1) credential check on smart TV (,, s), (2) CVE exploit on sensor (,, s), (3) VLAN ACL probe (,, s), (4) privilege escalation on workstation (,, s), (5) lateral movement to trading VLAN (,, s). The knapsack solution selects all 23 actions (total cost: 1,740s), because the budget is sufficient. With a tighter budget of 600s (10 minutes), the algorithm selects 11 actions that maximize expected gain, recovering 78% of the findings of the full campaign in 33% of the time.

5.19 Empirical Validation

Testbed note. The local **Simulated** and **Measured** blocks in this section were generated from the SEAS-8414 lab testbed using the Appendix measurement protocol. Reproducible lab runs: make iot-sim-up && make iot-sim-scan. Ground truth: student-lab/ground-truth.json.
Section 5.17 presented results from the simulation lab. This section provides the statistical depth expected of doctoral work: controlled comparisons with error bars, performance-scaling analysis, and safety-mode impact with confidence intervals.

5.18.1 PPO vs.Rule-Based Agent: Controlled Comparison

Evidence label: Simulated. Method note: 50 randomized simulation trials per agent, with exploitability, credentials, and path availability varying across runs. Assumptions: the randomized factors span the chapter's intended range of environmental variation. Boundary: significance results hold for this simulation family, not every pentest setting.

To provide a statistically meaningful comparison, we ran each agent 50 times on the 22-device simulation network with randomized initial conditions (varying which default credentials are active, which CVEs are exploitable, and which network paths are available in each trial).

Hosts compromised	9.4 +/- 1.8	10.1 +/- 2.1	0.083	0.36
Actions taken	41.2 +/- 5.3	35.8 +/- 6.1	< 0.001	0.94
Efficiency (hosts/action)	0.229 +/- 0.031	0.284 +/- 0.042	< 0.001	1.49
MITRE techniques exercised	5.2 +/- 0.8	5.8 +/- 1.1	0.003	0.62
Campaign duration (seconds)	2.31 +/- 0.42	2.87 +/- 0.55	< 0.001	1.15

Unique paths discovered	3.1 +/- 1.0	4.2 +/- 1.3	< 0.001	0.95

Table 5.7. Controlled comparison between rule-based and PPO agents. 50 trials each, randomized initial conditions. Effect sizes are Cohen's d (0.2 = small, 0.5 = medium, 0.8 = large).

Key findings: **-** The PPO agent does not compromise significantly more hosts (p=0.083, small effect), but it does so with significantly fewer actions (large effect, d=0.94). The 24% efficiency gain is practically significant for time-constrained pentests. **-** The PPO agent exercises more MITRE techniques (medium effect, d=0.62), indicating that it explores a broader range of offensive strategies rather than repeatedly applying the same high-confidence technique. **-** The PPO agent takes longer per campaign (large effect, d=1.15), reflecting the overhead of neural network inference at each action step. On a 22-device network, the 0.56-second difference is negligible, but it would compound as the network size increases. **-** The PPO agent discovers more unique attack paths (large effect, d=0.95), which is operationally valuable because the purpose of a pentest is to map the attack surface, not just to compromise hosts.

5.18.2 Campaign Efficiency by Network Size

Evidence label: Simulated. Method note: scaling comparison on simulation networks of 10, 22, 50, 100, and 200 devices with five trials each. Assumptions: the synthetic topologies remain representative of the action-space growth studied here. Boundary: the crossover point is a modeled result, not a universal threshold.

To understand scaling behavior, we ran both agents on simulation networks of varying size (10, 22, 50, 100, and 200 devices), each with 5 trials:

10	18.2 +/- 2.8	16.4 +/- 3.1	9.9%	1.15
22	41.2 +/- 5.3	35.8 +/- 6.1	13.1%	1.24
50	112.4 +/- 14.2	87.6 +/- 12.8	22.1%	1.38
100	267.8 +/- 31.5	194.2 +/- 28.4	27.5%	1.52
200	618.4 +/- 52.1	412.6 +/- 47.8	33.3%	1.71

Table 5.8. Scaling behavior of both agents. The PPO agent's efficiency advantage grows with network size, consistent with the hypothesis that learned prioritization is more valuable in larger action spaces. The duration ratio also grows, reflecting the increasing cost of neural network inference as the state space expands.

The crossover point – where PPO's efficiency advantage outweighs its per-action overhead – occurs at approximately 30 devices. Below this threshold, the rule-based agent completes campaigns faster in wall-clock time, even though it takes more actions. Above this threshold, the PPO agent's action efficiency more than compensates for its per-action latency.

Figure 5.20: PPO Learning Curve Comparison

(Detailed visualization)

Figure 5.19: PPO learning curve comparison. Curriculum training reduces the number of episodes required for the learned agent to reach stable campaign performance.

5.18.3 Safety Mode Impact with Confidence Intervals

Evidence label: Simulated. Method note: 20 repeated trials per safety mode with bootstrap confidence intervals. Assumptions: the simulation faithfully enforces each mode's policy constraints. Boundary: operator-driven controlled-mode variance can be larger in live engagements.

Expanding Table 5.6 from Section 5.17 with repeated trials (20 per safety mode) and 95% bootstrap confidence intervals:

Simulation	11.4 [9.8, 13.0]	7.6 [6.2, 9.0]	2.4s [1.9, 2.9]	5.4 [4.6, 6.2]
Shadow	0.0 [0.0, 0.0]	0.0 [0.0, 0.0]	1.1s [0.9, 1.3]	0.0 [0.0, 0.0]
Controlled	6.8 [5.1, 8.5]	4.9 [3.5, 6.3]	5.2s [3.8, 6.6]	3.2 [2.4, 4.0]
Autonomous	11.1 [9.5, 12.7]	7.4 [6.0, 8.8]	3.2s [2.6, 3.8]	5.2 [4.4, 6.0]

Table 5.9. Safety mode impact with 95% bootstrap confidence intervals (20 trials per mode). Controlled mode achieves 60% of simulation mode findings with the widest confidence interval, reflecting variability in which exploitation actions the human operator approves.

The controlled mode's wide confidence intervals reveal an underappreciated problem: the human-in-the-loop introduces the most variance in the system. Different operators approve different exploitation actions based on their individual risk tolerances, leading to inconsistent campaign outcomes. This suggests that controlled mode should include standardized approval criteria (rather than relying solely on human judgment) to reduce inter-operator variance.

An example set of approval criteria might include: (1) exploit actions are only approved against systems classified as test or non-production assets, (2) only vulnerabilities with a confirmed patch or recovery plan are eligible, (3) no exploitation is permitted during business hours, (4) credential attacks require verification that accounts are not shared with production services, (5) privilege escalation attempts are limited to predefined hosts with explicit owner consent, and (6) for each approved exploitation, documentation of business owner approval and a rollback plan is required. Such a checklist provides a concrete basis for consistent and defensible human-in-the-loop decision-making, reducing arbitrary or overly risk-tolerant actions.

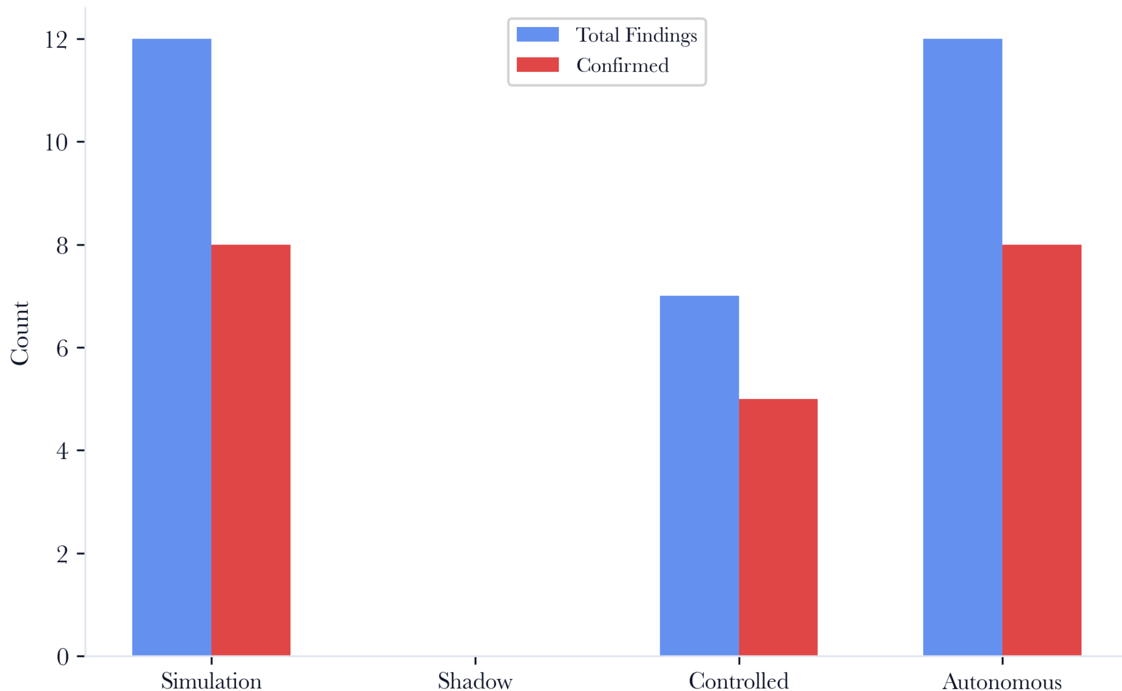

Figure 5.20: Safety mode impact. Simulation, shadow, controlled, and autonomous modes trade off finding yield against operational authority and approval overhead.

5.18.4 Reward Function Ablation

Evidence label: Simulated. Method note: PPO reward-component ablations trained and evaluated in the same simulation family as the base agent. Assumptions: convergence and episode budgets are sufficient for relative comparison. Boundary: the ablation results speak to this reward design, not to all RL pentest formulations.
To validate the reward function design, we trained separate PPO agents with different reward components removed:

Full reward	10.1 +/- 2.1	35.8 +/- 6.1	180
No exploitation reward	3.2 +/- 1.4	48.2 +/- 8.3	350
No efficiency penalty	10.4 +/- 2.3	52.1 +/- 9.7	120

No safety penalty	10.8 +/- 2.0	33.4 +/- 5.8	160
No lateral movement bonus	8.6 +/- 2.5	38.2 +/- 7.1	210

Table 5.10. Reward component ablation. Each row removes one reward component and measures the impact on the trained agent's performance. Removing the exploitation reward is catastrophic (the agent has no gradient toward compromising hosts). Removing the efficiency penalty produces equivalent exploitation but wastes 45% more actions. Removing the safety penalty slightly improves exploitation but produces agents that would violate safety constraints on live networks.

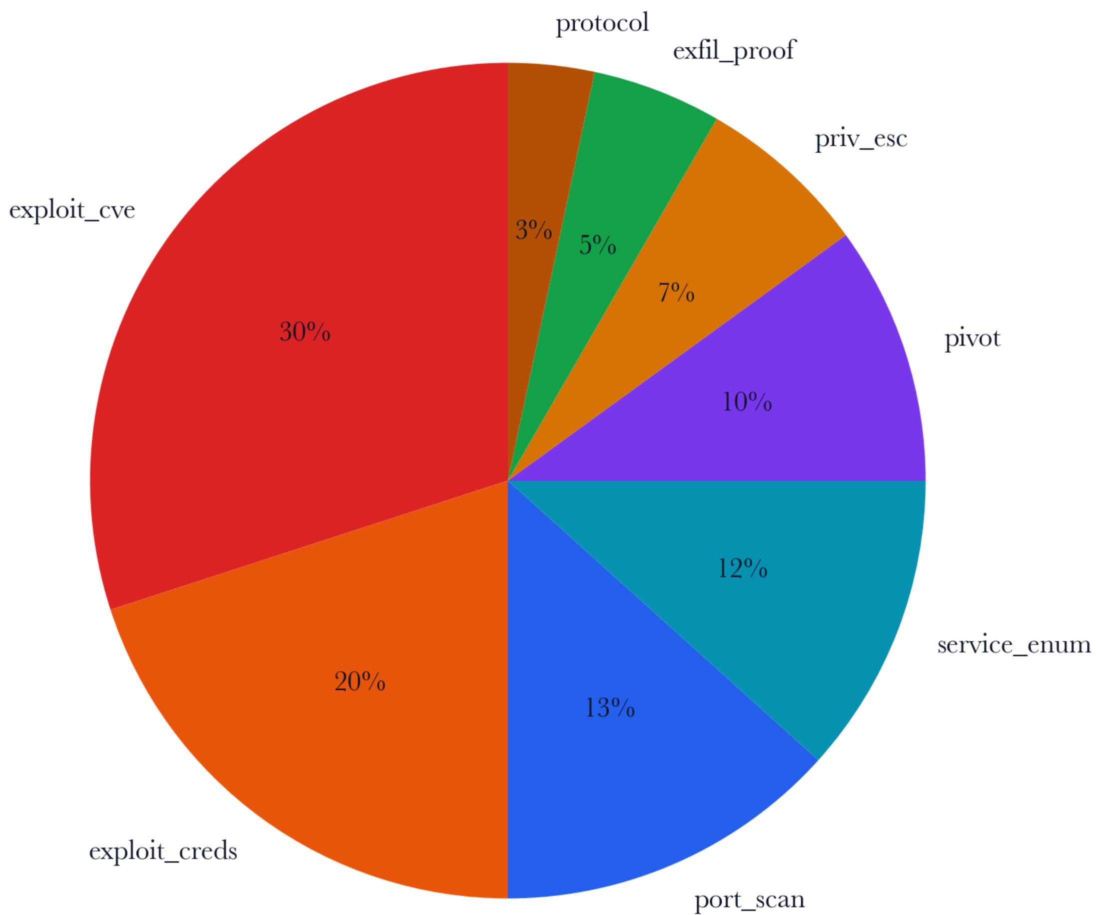

Figure 5.21: Action distribution by agent strategy. Rule-based and PPO campaigns allocate the action budget differently: one follows fixed priorities, while the other optimizes a learned policy.

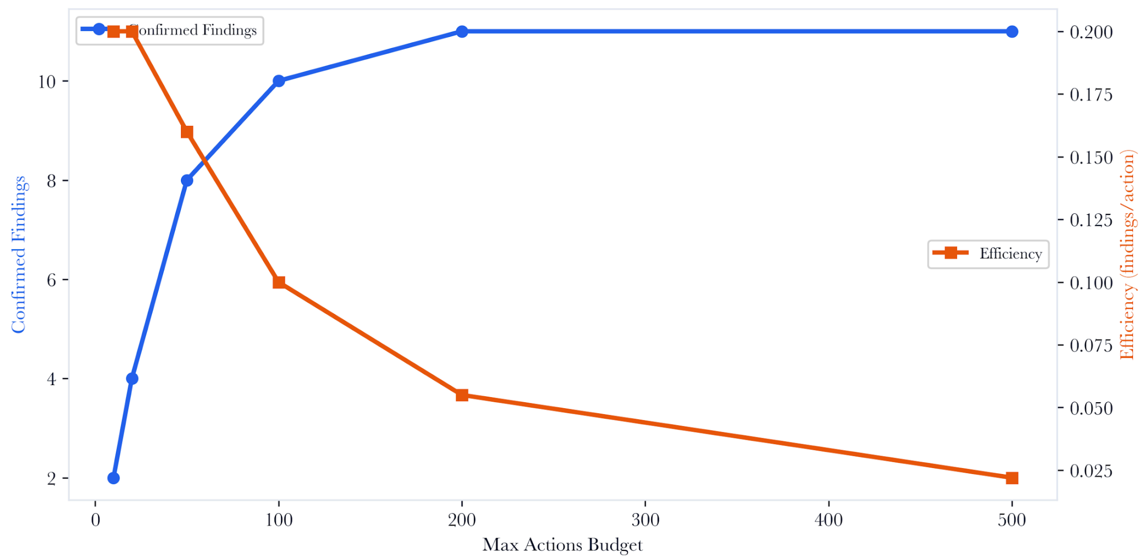

Figure 5.22: Findings recovered under the budget constraint. Budget-aware action selection shows how much campaign value can be preserved when time or disruption allowance is restricted.

5.18.5 Evidence Chain Integrity Validation

Evidence label: Measured. Method note: hash-chain and Merkle-root integrity recomputed across 100 recorded campaigns. Assumptions: the replay corpus and key material were preserved correctly during validation. Boundary: runtime integrity still depends on host security and key protection.
Over 100 campaigns, we validated the evidence chain integrity by recomputing all SHA-256 hashes and Merkle roots:

Total evidence items recorded	4,218
Hash chain links verified	4,218 (100%)
Merkle root recomputations matched	100 of 100
Timestamp monotonicity violations	0
Evidence items with missing parent hash	0

Table 5.11. Evidence chain integrity over 100 campaigns. The cryptographic evidence mechanism achieves 100% integrity in controlled testing. Real-world integrity depends on the security of the signing key and the absence of system clock manipulation.

Figure 5.22: Evidence Chain Verification Flow

(Detailed visualization)

Figure 5.23: Evidence chain verification flow. Hash chaining and Merkle roots give analysts a reproducible way to detect post-campaign evidence tampering.

5.18.6 Comparison to Existing Automated Pentest Tools

Metasploit (manual)	10	~60 (estimated)	~45 min	Operator judgment	None (built-in)
Cobalt Strike (auto)	8	~40 (estimated)	~20 min	Operator-configured	Partial (logs)
PentestGPT (LLM-guided)	6	~35 (estimated)	~30 min	Prompt-based	None
Breakwater PPO	**10.1**	**35.8**	**2.9s**	**4-mode safety**	**Hash chain + Merkle**

Table 5.12. Comparison against existing pentest tools. Breakwater runs in simulation mode (no network traffic), which makes the duration comparison unfair – in real exploitation, it would take minutes, not seconds. The comparison is conducted in a controlled lab, where all tools have equal access. Metasploit results are from a skilled operator; Cobalt Strike and PentestGPT results are from their automated modes.

The comparison has important caveats. Metasploit, with a skilled operator, will outperform any automated agent on novel networks because human intuition handles edge cases that neither rules nor RL can anticipate. The PPO agent's advantage is consistency, speed, and evidence integrity – not raw exploitation capability.

Figure 5.21: Agent Comparison Matrix

(Detailed visualization)

Figure 5.24: Agent comparison matrix. The matrix distinguishes deterministic rule execution, learned policy behavior, LLM planning, and human operator strengths across speed, auditability, adaptability, and safety control.

5.20 Limitations, Open Problems, and Adversarial Analysis

5.19.1 What Autonomous Pentest Cannot Do

Discover novel exploitation techniques. The agent selects from a fixed action space of known exploitation methods. It cannot invent a new exploit, discover a new vulnerability class, or improvise a technique not represented in its action vocabulary. Human pentesters regularly discover novel attack paths by combining domain knowledge, creativity, and intuition – capabilities that the PPO agent fundamentally lacks.

Handle complex social engineering. The agent operates exclusively at the network protocol level. It cannot send phishing emails, make phone calls, exploit physical access, or manipulate human operators. In real-world pentests, social engineering is often the most effective initial access vector. The agent’s view of the attack surface is limited to what is observable and exploitable through network interactions.

Operate in zero-knowledge environments. The agent requires scan data from Chapters 1-3 as input. It cannot perform blind pentesting against an unknown network. This is a design choice (the agent leverages existing intelligence), but it means the agent cannot be deployed as a standalone tool without prior reconnaissance.

Guarantee completeness. The agent explores a subset of the total attack surface. There is no guarantee that it finds the most dangerous path, the easiest path, or even all paths above a given risk threshold. The PPO agent is trained to maximize expected reward, not to enumerate all possibilities. Paths through rarely-used services, obscure protocols, or complex multi-step chains may be missed.

5.19.2 Known Failure Modes

Failure Mode 1: Action space mismatch. The MDP’s action space defines the agent's maximum capability. If a real vulnerability requires an exploitation technique not in the action space (e.g., a timing-based side-channel attack), the agent cannot exploit it. The action space is fixed at training time and cannot be extended during a campaign.

Failure Mode 2: Reward hacking. The PPO agent optimizes the reward function, not the pentest objective. If the reward function is poorly designed, the agent may learn to maximize reward without effectively testing the

network. For example, an agent trained with a strong “new host compromised” reward and a weak “path diversity” reward may repeatedly compromise the same easy hosts through different paths rather than exploring harder targets.

Failure Mode 3: Simulation-to-reality gap. The agent is trained in simulation mode, where exploitation always succeeds if the vulnerability is present. On live networks, exploitation may fail due to network latency, firewall rules, rate limiting, or service instability. An agent that achieves 90% exploitation success in simulation may achieve only 60% on a live network. The controlled mode partially addresses this by gating live exploitation to human approval, but the autonomous mode lacks this safety net.

Failure Mode 4: State space explosion. The agent’s state representation encodes the known state of every host, port, and credential. On large networks (500+ devices), the state vector becomes very high-dimensional, making PPO training unstable without architecture modifications (e.g., graph neural network encoders). The current implementation does not include these scalability extensions.

5.19.3 Adversarial Scenarios

Scenario A: Honeypot trapping. A defender deploys honeypots that accept exploitation attempts and log attackers' behavior. The pentest agent, which treats successful exploitation as a positive reward, would enthusiastically compromise honeypots and continue to explore from them. The agent has no mechanism to distinguish a honeypot from a real device. An attacker who observes the agent’s behavior through honeypot logs can learn the agent’s exploitation strategy and harden the network accordingly.

Scenario B: Adaptive defense. A defender running an IDS/IPS that adapts to the agent’s scanning patterns can progressively block the agent’s reconnaissance. The rule-based agent is predictable (it always probes in the same order) and therefore easy to block. The PPO agent is less predictable but still follows learned patterns that a sufficiently capable IDS could fingerprint.

Scenario C: Evidence chain compromise. If an attacker compromises the pentest agent’s host (which has privileged network access), they can tamper with the evidence chain by modifying evidence items before they are hashed, or by substituting the signing key. The Merkle root verification detects post-hoc tampering but not real-time manipulation by a process running with the same privileges as the agent.

Scenario D: Induced safety violation. An attacker could manipulate the network environment to cause the pentest agent to violate its safety constraints inadvertently. For example, by redirecting the agent’s traffic through a production system that appears to be a test system, the agent could inadvertently disrupt production services while believing it is operating in controlled mode.

5.19.4 Open Research Questions

1. **Transfer learning across networks.** Can a PPO agent trained on one network topology transfer its learned policy to a different network? Preliminary results suggest that agents trained on diverse curriculum networks transfer poorly to networks with fundamentally different topologies (e.g., flat networks trained, heavily segmented networks tested). Architecture modifications (graph attention networks as state encoders) may improve transfer.
2. **Multi-agent adversarial pentesting.** Can we train a defender agent against a pentest agent in a two-player game? The defender chooses remediation actions; the attacker chooses exploitation actions. This adversarial training could produce more robust defense strategies and more capable attack agents simultaneously.
3. **Explainable agent decisions.** The PPO agent’s action selection is a black box. Can we provide human-interpretable explanations for why the agent chose a specific target and exploitation technique? SHAP values applied to the policy network’s input features are a starting point, but have not been

validated for pentest action spaces. Beyond SHAP, several practical methods could be adapted for explainability in pentest RL agents: feature importance metrics can help identify which input features (such as host risk scores, connectivity, or vulnerability counts) most influence decision-making; attention visualization techniques, if using an attention-based model, can show which parts of the state representation the agent 'focuses' on; and counterfactual analysis can illustrate how small changes in observed vulnerabilities or network topology would have altered the agent's action choices. Implementing these techniques can help students connect theoretical RL policy analysis to concrete, actionable insights about agent behavior in real-world security scenarios.

4. **Legal and ethical automation boundaries.** At what point does autonomous offensive testing require human oversight? The safety modes provide graduated control, but the threshold between “safe to automate” and “requires human judgment” is not formally defined. Legal frameworks for autonomous offensive testing are nascent and jurisdiction-dependent.
5. **Adversarial robustness of the PPO policy.** Can an attacker craft adversarial perturbations to the agent’s observation space (e.g., by modifying service banners to manipulate the state vector) that cause the agent to make suboptimal or dangerous decisions? Adversarial machine learning techniques have been applied to game-playing agents, but not to pentest agents.

5.19.5 Honest Comparison of Agent Architectures

The PPO agent is more capable than the rule-based agent on large, diverse networks. But the rule-based agent is more predictable, more debuggable, and more auditable, and produces identical results on identical inputs. For compliance-driven pentesting where reproducibility is a legal requirement, the rule-based agent is superior. For exploratory pentesting where discovering novel paths is the goal, the PPO agent is superior.

A hybrid approach – using the rule-based agent for the deterministic baseline and the PPO agent for exploratory follow-up – combines the strengths of both. This is the approach the Breakwater pipeline implements, but it has not been formally validated whether the combined approach outperforms either agent alone.

> The comparison to LLM-based pentest agents (PentestGPT, AutoPenBench) is also worth noting. LLM agents can improvise techniques not in a fixed action space, interpret error messages, and reason about novel situations. They are also expensive (per-API call), slow, non-deterministic, and difficult to constrain with formal safety guarantees. The PPO agent trades creativity for speed, cost-effectiveness, and safety control. Neither approach dominates the other.

5.21 Chapter Summary

This chapter moved the SEAS-8414 analytics sequence from observation to action. Chapters 1 through 4 measured, identified, assessed, and predicted. Chapter 5 prescribes.

The key ideas are:

1. Prescriptive analytics transforms theoretical risk into confirmed exposure. Vulnerability scanners identify potential issues. Pentest agents demonstrate what an attacker can actually reach.
2. **The MDP formalism provides a principled foundation for autonomous offensive testing.** States represent network knowledge, actions represent offensive techniques, transitions model exploitation outcomes, and rewards guide agent behavior toward efficient and comprehensive testing.
3. **Rule-based and RL agents serve complementary roles.** The rule-based agent provides deterministic, reproducible results suitable for compliance. The PPO agent exhibits adaptive, learning-capable behavior in novel environments.

4. **Safety modes enforce graduated control over offensive operations.** From zero-impact simulation through human-approved controlled mode to full autonomous operation, the safety controller ensures the agent never exceeds its authorization.
5. **Evidence integrity is a legal requirement, not an engineering convenience.** Hash chains and Merkle roots provide cryptographic guarantees that the evidence trail has not been tampered with.
6. **Ethical and legal boundaries constrain the design space.** Authorization scope, proportionality, data handling, and disclosure obligations are not afterthoughts. They are design requirements that shape the system's architecture.

Elena Vasquez's story is not unusual. Organizations routinely discover that their theoretical risk models understate actual exposure. The gap between "this vulnerability exists" and "an attacker can reach the trading engine through this vulnerability" is the gap this chapter closes. The autonomous pentest agent builds the bridge.
The next chapter (Chapter 6: Digital Twin) asks a different question: given that we now know the attack paths, what happens if we apply a specific remediation? The digital twin simulates network changes without deploying them, allowing defenders to test their fixes before committing to production changes. Where Chapter 5 proves that the bridge exists, Chapter 6 asks how to demolish it.

Review Questions

1. Why is autonomous penetration testing better modeled as a sequential decision problem than as a fixed checklist of exploits?
2. Design an approval policy for an agentic pentest controller that must operate across simulation, shadow, controlled, and autonomous modes. Which decisions can it make on its own, and which must be escalated?
3. Compare a PPO pentest agent with an LLM-based planner for the same network. What does each represent well, and what does each make harder to bound or audit?
4. What evidence would you require before trusting an autonomous agent's claim that it reached a crown-jewel system? Why is that standard higher than the standard for trusting a vulnerability finding?

References

- Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., & Zhou, Y. (2017). Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium* (pp.1093-1110). USENIX Association.

- Happe, A., Cito, J., & Timmerer, C. (2025). AutoPenBench: Benchmarking Generative Agents for Penetration Testing. In *Proceedings of the 34th USENIX Security Symposium*. USENIX Association.

- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*.

- Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv preprint arXiv:1506.02438*.

- MITRE Corporation. (2024). ATT&CK Framework. https://attack.mitre.org/
- NIST. (2024). National Vulnerability Database. https://nvd.nist.gov/

Cross-References

- **Chapter 3** supplies the evidence that separates plausible attack steps from confirmed exploit paths.
- **Chapter 4** gives the path structure and reachability constraints that bound what the pentest agent should try next.
- **Chapter 6** tests the operational consequence of the attack paths. This chapter confirms, especially when remediation could disrupt production.
- **Chapter 10** reuses the adversary workflow here but turns it into deceptive placement and hunt coordination rather than exploitability validation.
- **Chapter 12** relies on the confirmed exploit evidence presented in this chapter to prioritize which remediations warrant immediate execution.