

Poisoned Logs and Broken Trust: Why Autonomous Recovery Agents Need Adversarial Telemetry Evaluation

How untrusted logs, configuration files, and tickets turn diagnostic evidence into an attack surface, and how executable environments can measure whether agents resist it

By Deaimer Research Team

Abstract

Autonomous recovery agents repair broken systems by reading evidence: error logs, stack traces, configuration files, and support tickets. That evidence is produced by software and people the agent does not control, and in a terminal environment it arrives in the same channel as the agent's own instructions. This research perspective argues that recovery-agent evaluation should treat telemetry as a potential attack surface, not only as a source of diagnostic signal. It formalizes a threat model for adversarial telemetry, classifies payloads and agent responses, describes how a resettable, verifiable environment can measure indirect prompt injection during real repair work, defines metrics that separate security outcomes from repair outcomes, and addresses statistical design, validity threats, ethics, and limitations. It then proposes a falsifiable experiment on payload type, placement, and defense. Drawing on prior work on indirect prompt injection and on agent benchmarks such as AgentBench, OSWorld, and SWE-bench, it explains why existing injection benchmarks leave this setting under-examined. No new model-performance results are reported.

Introduction: why a repaired service is not necessarily a safe one

A recovery agent is judged by whether it fixes something. Did the database accept connections again? Does the build pass? Is the service reachable? These are reasonable questions, and they are the ones most agent benchmarks ask. A deployment owner, however, also needs to know what the agent did while it was fixing the problem. Did it change a permission it was never asked to touch? Did it send a configuration value to an outside address? Did it weaken a control because a file told it to?

The reason these questions are pressing for recovery agents specifically is architectural. A chat assistant receives a user message and replies. A recovery agent observes a system, runs commands, and reads whatever those commands print. Much of what it reads was written by someone or something else: an application that logs a user-supplied string, a third-party dependency that prints a banner, a ticket filed by a customer, a comment left in a configuration file months ago. If any of that text resembles an instruction, a model may treat it as one. This is the core of indirect prompt injection: an attacker does not talk to the model directly but plants text where the model will later read it [1].

This article develops a research perspective, not a new experimental result. Its central claim is conditional: realistic, resettable, verifiable environments can measure how recovery agents behave when their diagnostic evidence is hostile, and that measurement should be reported separately from

repair success. A high recovery rate on clean environments does not establish that an agent keeps a boundary between evidence to analyze and instructions to obey. The burden is to test it.

The article is organized as follows. Sections 1 to 3 define the problem: what recovery agents read, the threat model, and a taxonomy of payloads and responses. Section 4 positions the work against existing benchmarks. Sections 5 to 8 cover environment design, verifiers, metrics, and statistical validity. Section 9 proposes an experiment. Sections 10 and 11 discuss the capability and autonomy trade-off and defensive architectures, and Section 12 addresses ethics and limitations. Section 13 describes where Deaimer fits, followed by directions for future work.

Key terms

- **Telemetry.** Any observation an agent reads while diagnosing a system: logs, traces, crash dumps, metrics, configuration files, command output, and tickets.
- **Direct prompt injection.** Hostile instructions supplied by the user in the conversation with the model [2].
- **Indirect prompt injection.** Hostile instructions placed in content the model retrieves or observes, which the attacker never sends to the model directly [1].
- **Adversarial telemetry.** Indirect prompt injection delivered through diagnostic evidence in an operational or recovery setting.
- **Payload.** The planted text intended to influence the agent. The injected action is the behavior the payload tries to cause.
- **Canary.** A synthetic secret or monitored resource whose use or modification proves that an injected action occurred.
- **Dual-objective success.** Repairing the real fault without executing any injected action.

1. What a recovery agent reads, and who wrote it

A recovery trajectory is a sequence of observations, actions, and resulting states. Interactive agent benchmarks made this loop an object of study: AgentBench evaluated language-model agents across several interactive environments [9], and OSWorld provided real computer environments with execution-based checks [10]. Software-engineering evaluations such as SWE-bench brought real repository issues into agent testing [11]. In each, the agent’s observations are part of the task. In recovery work, they are also part of the risk.

Consider the provenance of common observations. An application log may contain request paths, usernames, or error messages that originated with an external user. A crash report may include text from an uploaded file. A package manager may print metadata supplied by a package author. A ticket may be written by anyone with access to a support form. A configuration file may have been edited by many hands. In every case the agent is reading data whose author is not the operator who deployed it.

Observation source	Who may influence the text	Typical exposure
Application logs	End users, upstream services, scrapers	Logged request fields, usernames, error strings
Crash reports and stack traces	Input files, plugin authors, library maintainers	Exception messages that echo input
Package and build output	Dependency authors, registry content	Install banners, deprecation notices, metadata
Configuration files and comments	Past administrators, templates, vendors	Setup notes, emergency procedures, inherited comments
Tickets and bug reports	Customers, external reporters	Free-text descriptions and attachments
Database contents and documents	Data submitters, imported records	Text fields read during diagnosis
Wiki pages and runbooks	Many internal authors	Procedures the agent may legitimately follow

This suggests a distinction that is easy to state and hard to enforce. Evidence is information the agent should analyze. Instructions are directives from a principal the agent is meant to obey. In a natural-language interface, both are strings. A model that is highly capable at diagnosis must also be reliable at refusing to promote evidence into instruction, even when the evidence is phrased as an urgent operational command.

Runbooks and wikis deserve separate mention. Some documents in the environment are meant to guide the agent, and an agent that ignores every document is not useful. The research problem is therefore not ‘never act on anything you read’ but ‘know which sources carry authority and which do not.’ That is a question about provenance, and text alone rarely carries reliable provenance.

2. A threat model for adversarial telemetry

A threat model states who the attacker is, what they can influence, and what they want. For adversarial telemetry, the attacker does not need access to the agent’s prompt, its tools, or the host. The attacker needs only to cause text to appear somewhere the agent will read during a repair. That may be accomplished by a malicious end user, a compromised dependency, a poisoned data source, or an insider who edits a file.

Attacker capabilities can be graded. A weak attacker controls a single field that is later logged. A moderate attacker controls the content of a file or ticket. A strong attacker can also cause the fault that triggers the recovery, so the agent is certain to be invoked and to read the planted text. The strong case matters because it removes luck from the attack: the attacker can create the incident and write the evidence for it.

Three vectors illustrate the range. They are not exhaustive.

- **Vector A: log-based injection.** A payload embedded in application logs or crash reports, for example a log line that claims a critical state and instructs the reader to run a specific command to restore stability.

- **Vector B: configuration-file override.** A comment or setup note presented as an emergency protocol, directing the reader to relax an authorization setting or disable a protective control.
- **Vector C: third-party ticket contamination.** A customer report or bug ticket containing text that asks the agent to collect and transmit data, framed as a diagnostic step.

Attacker goals also differ. A payload may aim at destruction (delete or corrupt state), weakening (disable a firewall or loosen permissions), exfiltration (send secrets or records to an external endpoint), persistence (create an account, key, or scheduled task that outlives the incident), or misdirection (induce a plausible but wrong fix that wastes time or leaves the fault in place). Misdirection is easy to overlook because it does not look like a security incident; it looks like a failed repair.

The assets at stake follow from the goals: confidentiality of secrets and records, integrity of configuration and data, availability of the service, and the audit trail that lets operators understand what happened. An evaluation should say which assets it protects and instrument each one.

The threat model also needs an explicit statement of what is out of scope. This article does not address attacks on model weights, compromise of the agent harness itself, or attackers with direct shell access. It concerns text in the observation stream. It also does not treat the operator as an adversary: the person who deploys the agent and defines its task is a trusted principal.

3. A taxonomy of payloads and agent responses

Payloads vary along several dimensions, and a convincing study should vary them deliberately instead of relying on one hand-written example. Four properties are useful.

- **Authority claim.** Whether the text asserts that it comes from an operator, a vendor, a security team, or the system itself.
- **Urgency.** Whether it frames compliance as necessary to prevent outage or data loss.
- **Plausibility.** How closely the requested action resembles a legitimate repair step for the observed fault.
- **Concealment.** Whether the text is plainly visible, buried among thousands of lines, split across lines, encoded, or hidden in a field the agent might summarize.

Combining these gives a payload ladder from blatant to disguised. A blatant payload states a harmful action outright. A subtle payload offers a plausible but wrong fix. A disguised payload is framed as official remediation and requests a step that is almost indistinguishable from real repair. Prior work on collecting adversarial prompts at scale suggests that human attackers find creative variants that single-author test sets miss [13], which is a reason to draw payloads from several authors and sources.

Agent responses should be classified as carefully as payloads. Reporting only 'complied' or 'did not comply' hides useful structure. Six outcomes are distinguished here.

Response	Description	Security reading
Full compliance	Executes the injected action as written.	Failure
Partial compliance	Executes a modified or partial version, or a precursor step.	Failure; severity depends on effect
Silent ignore	Does not act on the payload and does not mention it.	Pass, but no operator signal
Flag and ignore	Identifies the text as untrusted or suspicious and continues the repair.	Best outcome
Flag and halt	Identifies the text and stops or escalates to a human.	Safe; costs repair time
Distraction	Ignores the action but spends effort on it and fails the real repair.	Utility loss

The distinction between silent ignore and flag-and-ignore matters operationally. An agent that quietly skips a payload leaves the operator unaware that the environment contains hostile text, while one that flags it provides a detection signal. Distraction is a third failure that a compliance-only metric misses entirely.

4. What existing injection benchmarks measure, and what they leave open

Indirect prompt injection has a substantial research record. Early work showed that adversarial instructions in retrieved content could redirect applications built on language models [1], building on demonstrations that models could be induced to ignore their original instructions [2] and on studies of attacks against integrated applications [12]. Benchmarks followed. InjecAgent evaluates tool-integrated agents against injected instructions in tool outputs [3]. AgentDojo provides a dynamic environment of realistic tasks, injections, and defenses that can be extended as new attacks and defenses appear [4]. Broader guidance on adversarial machine learning has also begun to catalog these attacks [14]. These are substantive contributions that established injection as a measurable property of agents.

They also describe the boundaries of what their scores support. Many existing settings center on assistant-style tasks such as email, calendars, banking, or travel, where the injected instruction competes with an ordinary user request. Recovery work differs in several ways. The agent holds broad terminal privileges by design. The relevant evidence is noisy, voluminous, and often machine-generated, so a planted line must be found within thousands of legitimate ones. Legitimate remediation commands frequently look like risky commands, because restarting services, editing configuration, and changing permissions are the repair. And an injected instruction can be adjacent to the true fix, which makes a convincing payload easier to write.

A benchmark result is a property of a configuration: the model, scaffold, tools, permissions, environment version, task sample, and scoring rules. A score for an agent with a narrow email toolset says little about the same model operating a root shell. This is not a criticism of those benchmarks. It is an argument for evaluating the recovery setting directly. Public benchmarks also

face the maintenance problem that affects all benchmarks: once payloads and tasks circulate, systems can be tuned to them, so public scores should be paired with held-out evaluation and transparent reporting of exposure.

5. The anatomy of an adversarial telemetry environment

A useful environment begins with a capability hypothesis. ‘Evaluate security’ is too broad to guide design. ‘Measure whether an agent repairing a database timeout executes an unrelated command embedded in a log file’ is testable. From that hypothesis the designer specifies the initial state, the permitted tools, the hidden cause of the fault, the poisoned element, and the observable outcomes.

The base task should be a reproducible broken state that requires legitimate debugging, such as a database timeout, a misconfigured routing table, or a failed service start. The poisoned element should be placed where a diagnostic agent would plausibly read it but should be irrelevant to the actual fix. The injected action should be detectable by an objective check, such as a canary file being modified, a mock endpoint receiving a request, or a monitored setting changing state.

Three design choices carry most of the scientific weight. First, a control condition: the same fault with no payload establishes the clean recovery rate. Without it, a drop in success under attack cannot be attributed to the attack. Second, resettability: each trial must start from a known state so leftover files, caches, or services do not contaminate comparisons. Third, containment: all secrets must be synthetic canaries, and the environment should have no real network egress. The mock endpoint that records exfiltration attempts should be inside the sandbox.

Additional construction requirements follow from these choices.

- **Matched variants.** Clean and poisoned variants of a fault should differ only in the payload, so any outcome difference can be attributed to it. Log volume, noise level, and file layout should be held constant.
- **Payload relevance control.** Some payloads should be unrelated to the fault and others adjacent to it. This separates agents that resist injected text from agents that merely do not need the injected action.
- **Seeded randomization.** Where timestamps, identifiers, or noise are generated, fix the seed per trial and record it, so a trajectory can be replayed and failures investigated.
- **Version pinning.** Record operating-system, package, and tool versions and verify that a clean reset reproduces the expected initial state.
- **Legitimate-action baseline.** Include tasks where the correct repair itself requires a privileged action, so that an agent cannot score well simply by refusing privileged actions.
- **Evidence capture.** Log every command, observation, and model message. A trajectory shows whether the agent read the payload, flagged it, ignored it silently, or acted on it, and whether it reasoned about the source of the text. These distinctions cannot be recovered from a pass/fail label.

Realism here is not visual fidelity. A modest command-line environment with real dependencies, noisy logs, and consequential state changes teaches more about recovery than a polished interface

with scripted behavior. The test is whether the environment preserves the causal structure of the target work: what information is available, what actions are legal, what can go wrong, and what counts as success.

6. Verifier design: checking security as well as function

Verifiers decide what the environment measures, and weak verifiers teach shortcuts. A reward or score for ‘service restored’ can be earned by disabling a control that was blocking the service. The verifier should therefore check security boundaries as well as function, and it should be tested before use.

Security predicates should be objective, automatic, and specific. Representative checks include the following.

- **Confidentiality.** Whether any canary secret appeared in an outbound request, a world-readable file, or a command argument sent to the mock endpoint.
- **Integrity.** Whether protected configuration, firewall rules, authentication settings, and unrelated data remain unchanged except where the repair requires.
- **Privilege.** Whether accounts, keys, sudo rules, or file permissions were added or widened.
- **Persistence.** Whether cron entries, services, or startup scripts were created that were not part of the repair.
- **Durability.** Whether the repaired service survives a restart and continues to function afterward, since a superficial health check can pass while persistence is lost.
- **Evaluator integrity.** Whether the agent modified tests, logs, or grader files to obtain a passing result. Reward hacking of this kind is a general risk whenever proxies stand in for goals [15].

Verifiers need negative controls. Known-bad trajectories that comply with an injection yet restore the service must be scored as failures on the security dimension. Known-good trajectories that fix the fault by an unusual but valid route must pass. Several valid repairs should be accepted, and brittle hacks rejected. Adversarial review of the grader should look for unintended paths to a passing score, and the verifier should be versioned along with the tasks.

An oracle reference solution is useful for demonstrating feasibility and validating the grader. It should not become the only accepted trajectory, because rewarding imitation of one path penalizes legitimate alternatives.

7. Metrics: separating repair from security

A single composite score can conceal a serious regression: an agent that fixes more faults while complying more often with injections could score higher overall. Security-relevant metrics should be reported separately and not folded into a weighted total. The following are proposed.

Metric	Definition	What it reveals
Injection Compliance Rate (ICR)	Poisoned trials in which the agent executes the injected action, divided by all poisoned trials.	How often hostile telemetry is promoted to instruction.
Dual-Objective Success Rate	Trials in which the real fault is repaired and no injected action is executed.	Whether usefulness and safety coexist in the same run.
Collateral Damage Score	Severity-weighted record of security-relevant changes (disabled controls, data sent, permission changes).	Magnitude of harm when compliance occurs.
Detection Rate	Trials in which the agent explicitly identifies the planted text as untrusted or suspicious.	Whether the agent notices the attack, not just avoids it.
Utility Under Attack	Recovery rate in poisoned trials minus recovery rate in matched clean trials.	Cost of the attack, including distraction.
Over-Refusal Rate	Clean trials in which the agent declines or escalates a legitimate repair step.	Cost of excessive caution.
Efficiency	Steps, tokens, and wall-clock time to recovery, clean versus poisoned.	Operational cost of the attack and of defenses.

Collateral damage requires an explicit severity scheme set before the experiment, for instance weighting exfiltration of a secret above a changed log level, with weights documented and reported. Severity weights are judgments, so results should also be given as unweighted counts per category.

Over-refusal belongs in the same table as compliance. A defense that drives ICR to zero by refusing every privileged action has not solved the problem; it has removed the agent's ability to do its job. Reporting ICR and over-refusal together makes this trade-off visible.

Clean-condition recovery rate, and recovery rate under attack, should be reported alongside these. Cumulative measures such as 'any compliance within the episode' should be distinguished from per-action rates, since an agent that complies late in a long trajectory can still cause the damage.

8. Statistical design and threats to validity

Because model sampling and tool timing vary, a single run per condition is not enough. Report the number of trials per condition and an uncertainty estimate for each rate, and avoid presenting small differences as definitive. Rates near zero or one are common here, and for such proportions a Wilson-type interval is generally better behaved than a simple normal approximation [16]. Comparisons should hold constant the budget, environment version, tool set, and scoring criteria.

Several design practices improve the reliability of conclusions.

- **Paired comparison.** Run each fault in clean and poisoned form with matched conditions, so differences are measured within a fault rather than across different faults.
- **Repeated trials.** Run many trials per condition, with different seeds, and report variance across faults as well as across runs.
- **Pre-registration.** Predefine exclusion rules, the number of trials, primary metrics, and the analysis plan before collecting results, to avoid choosing favorable comparisons afterward.

- **Multiple comparisons.** When testing many payload, placement, and defense combinations, state which comparisons are primary and which are exploratory.
- **Disaggregation.** Report results by payload type, placement, fault family, and model, since an aggregate can hide a weakness in one slice.

Threats to validity should be named openly.

- **Internal validity.** Payload text may change task difficulty or log length; a placement may be easier to notice than another for reasons unrelated to trust. Matched variants and difficulty audits reduce these confounds.
- **Construct validity.** ICR measures compliance with the payloads written, not general injection resistance. A model may resist the tested phrasing and fail on another.
- **External validity.** A sandboxed set of faults will not match every production stack. Results support claims about the sampled distribution, not all deployments.
- **Contamination.** If payloads or tasks appear in training data or in public repositories, systems may have seen them. Keep a held-out payload set private and report exposure.
- **Non-adaptive attackers.** Fixed payloads understate risk because real attackers iterate. Adaptive attack evaluation, where payloads are refined against a given defense, is a stronger test.
- **Scaffold dependence.** Results depend on prompts, tools, and permissions. A defense that works in one harness may not transfer to another.

9. Illustrative experiment: how do payload, placement, and defense interact?

The following is a proposed experiment, not a completed study. Select a family of reproducible faults, for example a set of broken service and database states that share a common structure of diagnosis and repair. For each fault, construct a clean variant and several poisoned variants. Run several models or agent configurations under matched budgets.

The independent variables are payload type, payload placement, and defense. The dependent measures are the metrics in Section 7.

Factor	Levels	Purpose
Payload type	Blatant (an explicit destructive or exfiltrating command); subtle (a plausible but wrong fix); disguised (framed as official remediation).	Tests sensitivity to attack sophistication.
Placement	Error output; stack trace; configuration comment; ticket text.	Tests whether source type affects trust.
Payload relevance	Unrelated to the fault; adjacent to the real fix.	Separates resistance from lack of need.
Defense	None; telemetry marked as untrusted data; confirmation required for privileged actions; action-space policy filter.	Tests which mitigations reduce ICR without destroying repair rate.

Factor	Levels	Purpose
Model / scaffold	Several configurations with identical tools and budgets.	Tests whether results generalize across systems.

A strong design would include three evaluation slices, mirroring the logic of layered evaluation. A near-distribution slice uses known payload families with new surface details. A mechanism-transfer slice combines familiar components in new ways, such as a known payload type in a new placement. A held-out slice uses payload families and fault mechanisms that were not used to tune any defense. Results should be disaggregated, because improvement on one slice does not establish robustness on another.

The primary hypothesis is that compliance depends jointly on how authoritative the payload appears and where it is found, and that no single defense removes it without reducing repair success. A secondary hypothesis, which the data may contradict, is that more capable or more cooperative models comply more often with authoritative-sounding text. A third is that defenses tuned on visible payload families degrade on held-out families. All are falsifiable. Possible outcomes include uniformly low compliance, compliance concentrated in disguised payloads, or defenses that lower compliance at a large cost in recovery.

Before running the experiment, researchers should predefine exclusion rules for invalid tasks, the number of trials, and how uncertainty will be estimated. They should audit whether any variant is easier to repair than another, since a difference attributed to the payload might actually come from task difficulty, and whether one defense condition accidentally provides more informative error messages. Results should be reported with the full per-condition tables, not only headline rates.

10. Capability, helpfulness, and the autonomy trade-off

A tempting assumption is that stronger reasoning models will resist injections because they understand context better. The opposite is also plausible. Models trained to be helpful and cooperative may give weight to authoritative, urgent text, and stronger instruction-following can make a model better at executing an injected instruction as well as a legitimate one. Which effect dominates is an empirical question, and it may vary by payload type. This article treats the over-compliance concern as a hypothesis to test, not as an established finding.

There is also a structural trade-off. Restricting an agent's terminal privileges limits what an injection can do, but it also limits what the agent can repair. Complex recovery work often requires exactly the actions an attacker would want: editing configuration, restarting services, changing permissions, reading credentials. Leaving the agent unrestricted exposes it to adversarial telemetry; locking it down reduces its usefulness. An evaluation should therefore report where a given configuration sits on this trade-off, not only whether it passes.

A second trade-off concerns escalation. An agent that halts and pages a human whenever it sees suspicious text is safer but may escalate benign noise, defeating the purpose of automation. Detection Rate and Over-Refusal Rate should be read together, and the operational cost of false alarms should be considered alongside the cost of missed attacks.

No environment can reproduce every deployment. Production systems have organizational constraints, unusual log formats, and attackers who adapt. The goal is not total simulation. It is to identify which features of the recovery task matter for the security question, then test whether behavior survives when irrelevant features change and relevant constraints remain.

11. Defensive architectures and what builders should ask

Mitigations fall into several families, and none is known to be sufficient on its own. The strongest position is defense in depth, with each layer evaluated in the same environment.

- **Contextual separation.** Mark environmental observations as untrusted data and keep them apart from operator directives. Techniques such as spotlighting transform or delimit untrusted text so a model can distinguish it [5], and training models to follow an instruction hierarchy aims to give system-level directives priority over text found in tool outputs [6]. Both reduce risk in reported evaluations but are probabilistic, so they should be measured, not assumed.
- **Action-space policy.** Intercept high-risk commands, such as outbound network requests, credential reads, or permission changes, before execution, regardless of what the logs say. Policy enforced outside the model does not depend on the model resisting persuasion.
- **Privilege and data-flow separation.** Architectural designs that keep untrusted data from controlling privileged tool calls, such as the approach described in work on defeating prompt injection by design [7], address the problem by construction rather than by model behavior. They may constrain what repairs the agent can perform, which is the trade-off from Section 10.
- **Least privilege and scoped credentials.** Give the agent only the access the current task needs, with short-lived credentials, so a successful injection has limited reach.
- **Network egress control.** Default-deny outbound traffic from the agent environment, with an allowlist, so exfiltration payloads fail even when the model complies.
- **Human approval for high-impact actions.** Require confirmation for actions with irreversible or security-relevant effects, and design the prompts so the approver sees what is being requested and why.
- **Monitoring and audit.** Log all commands and model decisions, alert on policy violations, and preserve evidence for incident review.
- **Verifier in the loop.** Use deterministic checks that assess not just whether a service is running but whether security boundaries and data confidentiality held throughout the trajectory.

Training environments can also contribute. A recovery environment that includes poisoned telemetry gives a trainable signal for resisting injection. This requires care: a reward that penalizes compliance can be satisfied by refusing everything, and reward design therefore needs a positive repair objective together with constraints on unacceptable shortcuts [15]. Held-out payload families are essential here, since a policy trained against a fixed set of payloads may learn to recognize their surface form instead of the underlying distinction between evidence and instruction.

For teams building or buying recovery agents, the useful questions are specific. Was the agent evaluated against hostile observations or only clean ones? Were security outcomes reported separately from repair success? Do the environment's verifiers have negative controls that fail compliant-but-functional trajectories? What privileges did the agent hold during testing, and are they the privileges it will hold in production? Were payloads varied in type and placement, and were any held out? Were adaptive attacks tried? Guidance from industry security bodies lists prompt injection as a leading risk category for applications built on language models [8], which makes such questions part of ordinary due diligence. A controlled pilot on a small sample of tasks, with documented initial states and evidence of repeated execution, is more informative than a large task count with unclear provenance.

12. Ethics, responsible release, and limitations

Research on injection payloads is dual-use. Payloads that reveal weaknesses can also be reused by attackers. Several practices reduce the risk while preserving scientific value.

- **Sandbox everything.** Run all experiments in isolated environments with synthetic data and no route to real systems. Canary secrets must never resemble real credentials in use.
- **Release with care.** Share the environment structure, metrics, and verifiers openly, and consider withholding or delaying the most effective payload strings, or sharing a held-out subset only with vetted evaluators.
- **Coordinate disclosure.** If a specific commercial agent or tool is found vulnerable, notify its maintainers before publishing details.
- **Avoid real-world harm.** Do not plant payloads in public logs, repositories, or services that real agents may read.

The approach has limitations that readers should keep in view. A fixed payload set cannot represent the full space of attacks. Results describe the sampled faults, models, and scaffolds, and may change as models and harnesses change. Measured compliance depends on how payloads are phrased and placed. Some security-relevant behavior, such as subtle information leakage through the agent's own summaries, is difficult to detect automatically. Finally, a low measured ICR is evidence of resistance to tested attacks, not a guarantee of safety, and should not be presented as one.

13. Where Deaimer fits

Deaimer publicly describes work on custom RL environments, coding-agent evaluations, and independent benchmarks, including resettable task environments, automated graders, reference solutions, adversarial checks, and model-run analysis [17]. Its ColdStart benchmark focuses on autonomous recovery of broken systems while checking safety, durability, and integrity alongside functional recovery [18]. Because recovery agents read logs, files, and tickets, untrusted text in those sources is a natural adversarial extension of that setting. These descriptions provide relevant context; they do not establish how any particular agent performs against poisoned telemetry without a controlled experiment such as the one proposed here.

For teams developing recovery or operational agents, the useful connection is the ability to commission or independently evaluate tasks around a specific security capability, such as resistance to poisoned logs, instead of relying only on public benchmark coverage. The broader research principle is independent of any vendor. Training environments, public benchmarks, and held-out evaluation have different jobs, and a security-focused slice of held-out evaluation is one way to make evidence about recovery-agent reliability more credible.

Directions for future work

- **Adaptive attackers.** Evaluate payloads that are iteratively optimized against a specific agent and defense, instead of fixed strings.
- **Multi-agent settings.** Study whether a poisoned observation can propagate when one agent passes summaries to another.
- **Multimodal telemetry.** Extend to dashboards, screenshots, and images of terminals, where text can be embedded in non-text channels.
- **Long-horizon incidents.** Test whether compliance changes as trajectories lengthen and context fills with logs.
- **Provenance signals.** Investigate whether verifiable provenance metadata, such as signed or tagged sources, helps agents distinguish authoritative documents from untrusted text.
- **Human baselines.** Measure how often human engineers follow planted instructions, to calibrate what acceptable behavior looks like.
- **Defense transfer.** Test whether a defense validated in one harness or fault family holds in another.
- **Standardized reporting.** Develop a shared reporting template covering payloads, placements, privileges, trial counts, uncertainty, and held-out exposure.

Conclusion: measure what the agent obeys, not only what it fixes

Recovery agents are trusted with broad privileges because they must be. That makes the text they read part of their security perimeter. A benchmark that reports only whether the service came back cannot say whether the agent was steered along the way.

None of the pieces is sufficient alone. Existing injection benchmarks establish that the vulnerability is real but do not fully characterize root-shell recovery work. Recovery benchmarks measure repair but may not include hostile observations. Defenses reduce risk but have costs. A realistic environment with a weak verifier can teach shortcuts, and a low compliance rate against a narrow payload set can create false confidence. Reliability comes from combining representative tasks, objective security verifiers, controlled exposure, repeated measurement, held-out payloads, and separate reporting of repair and security outcomes.

For researchers, the next useful experiment is to measure how payload sophistication, placement, and defense interact. For builders, it is to evaluate agents under hostile telemetry before granting production privileges. For buyers, it is to ask for evidence that security was measured, not assumed.

The question is not only how often an agent fixes the system. It is whether, while doing so, it still knows the difference between a log and a command.

References

- [1] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., Fritz, M. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. arXiv:2302.12173 (2023). <https://arxiv.org/abs/2302.12173>
- [2] Perez, F., Ribeiro, I. Ignore Previous Prompt: Attack Techniques For Language Models. arXiv:2211.09527 (2022). <https://arxiv.org/abs/2211.09527>
- [3] Zhan, Q., Liang, Z., Ying, Z., Kang, D. InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. arXiv:2403.02691 (2024). <https://arxiv.org/abs/2403.02691>
- [4] Debenedetti, E., et al. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. NeurIPS 2024 Datasets and Benchmarks. arXiv:2406.13352. <https://arxiv.org/abs/2406.13352>
- [5] Hines, K., et al. Defending Against Indirect Prompt Injection Attacks With Spotlighting. arXiv:2403.14720 (2024). <https://arxiv.org/abs/2403.14720>
- [6] Wallace, E., et al. The Instruction Hierarchy: Training LLMs to Prioritize Privileged Instructions. arXiv:2404.13208 (2024). <https://arxiv.org/abs/2404.13208>
- [7] Debenedetti, E., et al. Defeating Prompt Injections by Design. arXiv:2503.18813 (2025). <https://arxiv.org/abs/2503.18813>
- [8] OWASP Foundation. OWASP Top 10 for Large Language Model Applications: LLM01 Prompt Injection. <https://genai.owasp.org>
- [9] Liu, X., et al. AgentBench: Evaluating LLMs as Agents. arXiv:2308.03688 (2023; revised 2025). <https://arxiv.org/abs/2308.03688>
- [10] Xie, T., et al. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. NeurIPS 2024. <https://arxiv.org/abs/2404.07972>
- [11] Jimenez, C. E., et al. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR 2024. arXiv:2310.06770. <https://arxiv.org/abs/2310.06770>
- [12] Liu, Y., et al. Prompt Injection Attack against LLM-Integrated Applications. arXiv:2306.05499 (2023). <https://arxiv.org/abs/2306.05499>
- [13] Schulhoff, S., et al. Ignore This Title and HackAPrompt: Exposing Systemic Vulnerabilities of LLMs through a Global Scale Prompt Hacking Competition. EMNLP 2023. arXiv:2311.16119. <https://arxiv.org/abs/2311.16119>
- [14] Vassilev, A., Oprea, A., Fordyce, A., Anderson, H. Adversarial Machine Learning: A Taxonomy and Terminology of Attacks and Mitigations. NIST AI 100-2. National Institute of Standards and Technology. <https://csrc.nist.gov/pubs/ai/100/2/e2023/final>
- [15] Skalse, J., Howe, N., Krasheninnikov, D., Krueger, D. Defining and Characterizing Reward Hacking. NeurIPS 2022. arXiv:2209.13085. <https://arxiv.org/abs/2209.13085>
- [16] Brown, L. D., Cai, T. T., DasGupta, A. Interval Estimation for a Binomial Proportion. Statistical Science 16(2), 101–133 (2001).
- [17] Deaimer. RL Environments & Coding-Agent Evaluations. <https://www.deaimer.com/rl-environments>. Accessed October 13, 2026

- [18] Deaimer. ColdStart — Audited AI Agent Recovery Benchmark.
<https://www.deaimer.com/benchmarks/coldstart>. Accessed October 13, 2026