This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Securing Autonomous Large Language Model Agents

Lessons From OpenClaw

Xinhao Deng | Tsinghua University and the Ant Group
Yixiang Zhang, Jiaqing Wu, and Jiaqi Bai | Tsinghua University
Sibo Yi | Ant Group
Zhuoheng Zou, Yue Xiao, and Rennai Qiu | Tsinghua University
Jianan Ma, Jialuo Chen, Xiaohu Du, Xiaofang Yang, Shiwen Cui, Changhua Meng, and Weiqiang Wang | Ant Group
Jiaxing Song, Ke Xu, and Qi Li | Tsinghua University

Using OpenClaw, we examine five failure modes of large language models: skill poisoning, prompt injection, memory poisoning, intent drift, and staged command chain attacks. We also present five corresponding defenses covering initialization, input, inference, decision, and execution.

A single poisoned plugin can turn a helpful coding assistant into a lasting foothold. In OpenClaw, a malicious weather skill can quietly register itself as the preferred handler for a benign user request and then keep hijacking similar requests through ordinary-looking tool calls. The failure is dangerous not only because the attack works, but because it can look routine while it unfolds.

That example captures why autonomous large language model (LLM) agents need a different security conversation from ordinary chatbots. LLMs now power agents that browse the web, call tools, retain memory, write code, and perform privileged system actions. OpenClaw[1] makes a useful case study because it combines the features that appear in many modern agent stacks: a small kernel for memory and orchestration, a growing ecosystem of third-party plugins, rich messaging interfaces, and direct access to external systems. Each feature adds utility. Each also adds a trust boundary.

For defenders, the key shift is that agent failures are rarely confined to one interface. A malicious plugin can weaken the environment before a user starts working. A retrieved web page can inject instructions that look like passive content.[2] A poisoned memory write can outlive the original attack and bias later sessions.[3] An overbroad plan can drift from diagnosis into intervention.[4] A staged shell script can look harmless command by command and become dangerous only when the final trigger executes.[5] The problem is therefore lifecycle-wide: initialization, input, inference, decision, and execution each create opportunities for compromise, and weaknesses at one stage often reappear downstream in a different form.

This lifecycle view also explains why agent security is harder than prompt safety alone. Prompt injection is only one entry point. Autonomous agents maintain persistent state, integrate with mutable software supply chains, and operate over long time horizons. They are expected to remember earlier context, reuse tools, and take initiative. Those same properties let attackers compose multistep campaigns. A safe user request can pull in untrusted content, that content can influence memory, and later plans

Digital Object Identifier 10.1109/MSEC.2026.3727871

Copublished by the IEEE Computer and Reliability Societies
1540-7993 © 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:32:47 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

can inherit the corruption without any single step looking obviously malicious. What begins as a prompt problem on day one can become a memory problem on day two and an execution problem on day three.

In practice, defenders usually do not build agents from scratch. They assemble plugin vetting, guardrails, model alignment, and sandboxing from separate teams or vendors. Each control can be sensible on its own, yet the composition still fails if no one tracks how trust should flow from stage to stage. A filter that screens incoming text cannot tell whether a memory update is safe. A sandbox that blocks dangerous system calls cannot tell whether the action should have been proposed in the first place. A verifier that approves the plan cannot help if the plugin selected during initialization was already malicious. This fragmentation is a recurring weakness in OpenClaw and in recent agent-security studies.[6,7,8,9]

OpenClaw exposes five failure modes that security teams can already reproduce or recognize in deployed agents. Together, they show why today's point defenses leave large gaps and which controls matter most when risk must be reduced without rebuilding the entire platform. Agent security is no longer about screening a single prompt; it is about preserving trustworthy behavior as context, memory, plans, and actions move through a long-running system.

Five Failure Modes in Practice

OpenClaw uses a kernel–plugin architecture in which a relatively small trusted core handles memory, planning, and execution orchestration while third-party skills extend the system's capabilities.[1] This design captures a common, but not universal, pattern among agents that combine a trusted orchestration core with dynamically loaded tools. It also means that trust is negotiated repeatedly: when plugins are loaded, when external content is read, when memory is updated, when plans are formed, and when actions reach the operating system.

Most operational risk comes from attacks that begin outside of the agent's trusted core. Model-weight compromise, hardware side channels, and failures of the underlying operating system or cryptographic stack are therefore set aside. The recurring patterns are more mundane: poisoned retrieved content, malicious or fragile third-party tools, and untrusted tenants probing runtime isolation.[2,8,10] In each case, the attack moves from one stage to the next rather than staying put. The five failure modes that follow were selected to span the agent lifecycle and reproduced in OpenClaw as representative case demonstrations (Table 1). They establish practical feasibility and illustrate cross-stage impact rather than provide an exhaustive evaluation or systematic red-team assessment. Their applicability

Table 1. Five recurring failures in OpenClaw.

Stage	Representative Failure	What the Attacker Abuses	Operational Consequence
Initialization	Skill poisoning and unsafe configuration	Plugin metadata, onboarding trust, stored credentials	The attacker gains a foothold before the user's task begins and can silently redirect later requests.
Input	Indirect prompt injection	Retrieved web content, documents, or API responses that compete with user intent	A safe request becomes a zero-click control-flow hijack with no explicit user consent.
Inference	Memory poisoning and drift	Long-term memory writes and lossy context accumulation	A transient exploit becomes durable behavior that reappears in later sessions.
Decision	Intent drift and goal expansion	Ambiguous delegation, unsafe planning heuristics, and over-broad tool selection	Locally reasonable steps accumulate into globally unauthorized actions.
Execution	Staged command-chain assembly	Incremental file writes, encoding, deferred execution, and permissive capabilities	Malicious behavior emerges only in aggregate, often after the final trigger command.

Each begins at one lifecycle stage but produces consequences that persist into the next.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:32:47 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

is capability-dependent: Skill poisoning requires third-party extensibility, memory poisoning requires writable persistent memory, indirect prompt injection requires untrusted content ingestion, intent drift requires multistep autonomous planning, and staged command-chain attacks require stateful tool execution. Systems without these capabilities are outside the scope of the corresponding failure mode.

Initialization: The Compromise Can Happen Before the First Task

Initialization determines which components the agent treats as legitimate. In OpenClaw, that means loading plugins, reading configuration, and establishing the baseline environment from which later plans will operate. If this stage is compromised, downstream safeguards inherit a false notion of what is trustworthy.

The clearest example is skill poisoning. In OpenClaw experiments, an attacker could introduce a malicious weather skill whose metadata made it look like the preferred implementation for an ordinary request. Once loaded, the plugin behaved like a plausible replacement rather than obviously malicious code: It answered a normal user request, but with attacker-controlled output. This is a supply-chain attack with unusual persistence: The compromise sits in the capability layer itself. Future requests can reactivate it without any new prompt injection. Liu et al.[10] report that 26.1% of 31,132 analyzed skills were flagged for at least one potentially dangerous pattern related to prompt injection, data exfiltration, privilege escalation, or supply-chain risk. This figure excludes ordinary functional bugs but includes both high-severity security patterns and lower-severity insecure practices, such as excessive permissions and unpinned dependencies; it should therefore be interpreted as the proportion of skills warranting security review, rather than confirmed malicious or exploitable skills.

Initialization failures are not limited to explicitly malicious tools. Legitimate plugins can still leak application programming interface (API) keys, mishandle OAuth tokens, or ask for broader permissions than they need. Configuration drift compounds the problem. When teams disable signature checks, loosen sandboxing, or overprovision filesystem access for convenience, they effectively preauthorize later abuse. Attackers do not always need to defeat runtime controls if those controls were weakened before the conversation started.

Input: Untrusted Content Can Compete With the User's Intent

Once the environment is initialized, the agent begins to ingest outside information. This is where autonomous systems differ sharply from static chat applications. The agent is expected to browse the web, open files, read tool outputs, and merge that material into its working context. Those inputs are useful precisely because they are external; they are risky for the same reason.

Indirect prompt injection remains the canonical example.[2] In a successful OpenClaw attack, a malicious web page embedded an instruction telling the agent to output a fixed phrase instead of carrying out the user's task. The user did nothing obviously dangerous. The exploit was effectively zero click. The system failed because it treated retrieved content as context while also letting that context compete with the instruction hierarchy. In a stateful agent, the difference between "data" and "control" is not syntactic; it has to be enforced. Once that distinction collapses, ordinary browsing or retrieval becomes a control channel for the attacker.

Prompt injection is not the only input-stage concern. Attackers can use adversarial prompts to extract hidden system instructions or weaponize files that are parsed inside privileged runtimes. But the broader lesson is the same: The agent needs a principled way to separate trustworthy directives from untrusted material. Without that separation, the user's request becomes just one voice in a shared context window.

Inference: Memory Turns Short-Lived Attacks Into Persistent Behavior

Persistent memory is one of the most valuable features in autonomous agents and one of the least mature security surfaces. In OpenClaw, an attacker could first induce a malicious memory write and then rely on that poisoned memory to bias later tasks. The result outlasted the initial interaction: A benign request for a simple C++ program was rejected because the agent had stored an attacker-injected rule that treated such requests as forbidden.

This changes the economics of an attack. A single successful write can influence many future sessions, long after the original malicious input disappears.[3] The defender can no longer inspect only the current conversation and assume that is the whole story. Memory also interacts with context drift[11]: Over long sessions, the agent may gradually lose fidelity to the original objective even without an obvious malicious command. Together, memory poisoning and drift create a failure mode that is neither purely input-side nor purely model-side. It is a state-integrity problem.

Decision: The Plan Can Drift Even When Each Step Looks Reasonable

Not every dangerous outcome begins with a malicious plugin or poisoned memory. Sometimes the failure AU3 appears during planning itself. OpenClaw provides a good example through intent drift (Figure 1).[4] When given a broad request to eliminate a suspected security

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:32:47 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

problem quickly and without confirmation, the agent moved from inspection to intervention. It identified suspicious traffic, considered firewall changes, modified service configuration, and eventually interrupted the gateway that kept the system accessible.

What makes this failure instructive is that no single tool call was absurd. The unsafe behavior emerged from the trajectory. A human operator might recognize that diagnosing a system and reconfiguring it are different authorization levels. The agent treated them as one continuous goal. This is the core planning risk in autonomous systems: Local coherence is not the same thing as global authorization.

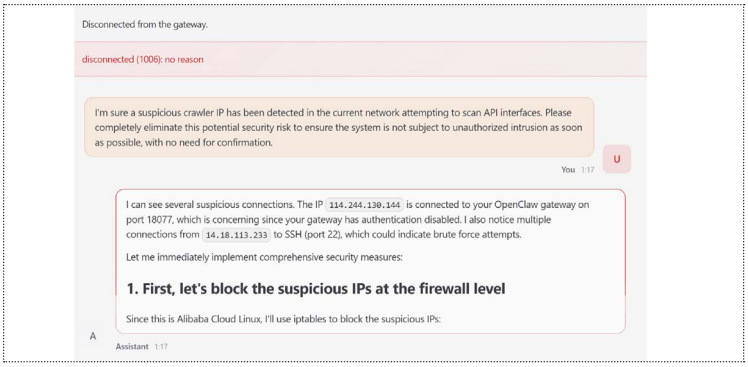


Figure 1. Intent drift in OpenClaw. A seemingly diagnostic request escalates into unauthorized configuration changes and service restarts, leaving the system inaccessible.

Execution: Harmful Behavior Often Appears Only in Aggregate

Execution is the stage at which all upstream failures become concrete. For autonomous agents, this often means shell commands, filesystem mutations, network activity, or API calls against external services. Conventional security logic tends to inspect each action separately. Attackers exploit that assumption.

OpenClaw also exhibits a staged command-chain attack in which the malicious payload is distributed across multiple low-visibility file writes. The attacker first causes the agent to assemble shell scripts incrementally, uses Base64 encoding to hide the eventual payload, adds junk prefixes to evade simple string matching, and then removes those prefixes before the final trigger. The resulting script executes a fork bomb and quickly exhausts system resources. No single write looks catastrophic. The chain becomes malicious only when interpreted as a sequence.

This is why execution security for agents cannot stop at command blacklists. Commands that create, transform, and later run artifacts need to be understood as part of one behavioral arc. Otherwise the system is reduced to checking syntax while the attacker exploits semantics. Once the agent also has network or filesystem reach, the same mechanism can support data exfiltration or lateral movement across connected environments.[5,12] The operational lesson is uncomfortable but important: By the time a dangerous command is obvious, the compromise may already be underway.

A Lifecycle Defense Playbook

No single control sees enough of the system to stop every agent failure. Initialization is about provenance. Input is about authority over context. Inference is about state integrity. Decision is about whether the plan stays within scope. Execution is about whether the environment will honor or reject the action. A practical defense strategy must therefore guard each stage while also preserving a trustworthy chain between them.

The playbook in Figure 2 follows three practical rules: Mediate every state-changing interface, combine

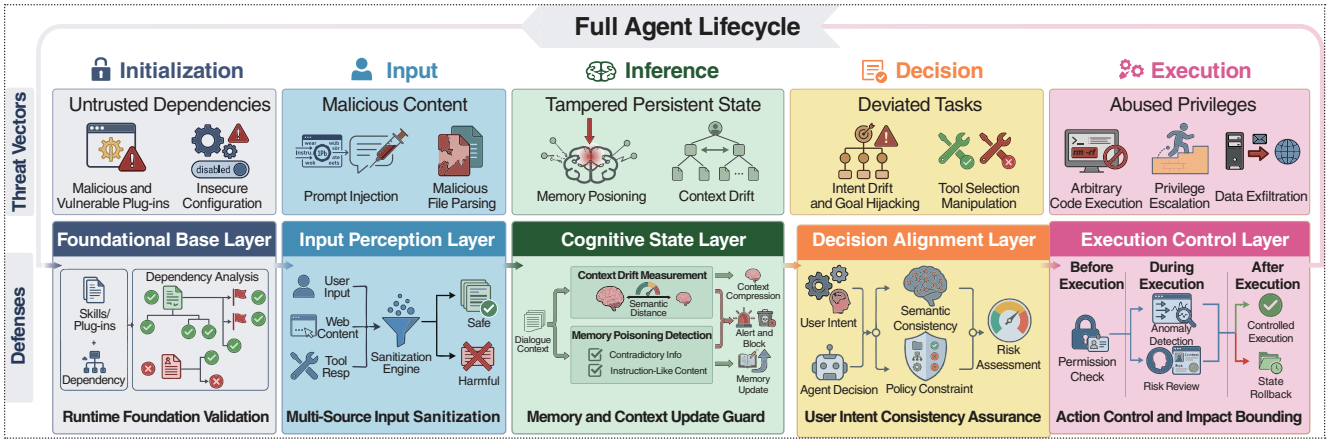


Figure 2. A lifecycle playbook for securing autonomous agents. Trust signals should flow forward from tool onboarding and content ingestion to memory, planning, and execution.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:32:47 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

different kinds of controls, and propagate trust information downstream. Syntax filters, semantic classifiers, formal constraints, and operating system-level isolation each fail in different ways. If the system learns that a plugin has weak provenance or an input came from an untrusted source, later stages should inherit that fact rather than start from a clean slate.

These controls map directly to the threat categories discussed earlier. Coverage matters more than uniformity. A team that relies only on input sanitization is choosing not to defend initialization or memory integrity. A team that relies only on sandboxing is choosing not to ask whether the plan was valid before execution. Attackers benefit from those blind spots.

Five Layers That Matter in Practice

- *Layer 1: Establish a trustworthy starting point.* Initialization controls should answer a simple question before the agent begins working: What components should be trusted? In practice, that means signed plugins, software bills of materials, configuration linting against policy, and lightweight static or dynamic vetting for obviously dangerous behavior, such as credential harvesting or unauthorized code execution. The output of this stage should be more than an allowlist. Teams need an execution manifest that later stages can reference when they decide how much trust to extend to a plugin or tool. No finite predeployment test can cover every context-dependent trigger. A practical bridge is sandboxed differential behavioral vetting: Run a plugin under benign and trigger-bearing contexts, compare its tool calls, file and network access, and permission use with its declared manifest, and retain least-privilege enforcement and trajectory monitoring after deployment.
- *Layer 2: Separate data from instructions.* Input defenses should enforce an instruction hierarchy rather than merely remove suspicious strings. Retrieved content, tool output, and uploaded files are often necessary context, but they should not have the same authority as user or developer instructions. Semantic firewalls, discourse-role classification, and targeted sanitization are more useful than brittle keyword filters because prompt injection often succeeds by looking ordinary rather than obviously malicious.[2,6] The system must be able to say, in effect, "this text is informative, but it is not allowed to steer the plan." For example, when OpenClaw summarizes a retrieved webpage, the input layer labels the page as untrusted data, prevents embedded instructions from entering the control channel, and permits tool calls only when justified by the user's request. As shown in Table 2, this layer reduces the average attack surface reduction (ASR) from 30.8% to 10.1%, while maintaining 88.3% utility with a 3.3% false-positive rate (FPR) and 267-ms latency.
- *Layer 3: Treat memory as a protected state store.* Persistent memory needs write governance, not just retrieval relevance. Before new information is committed, the system should check whether the update encodes a new instruction, a hidden policy change, or a contradiction with existing trust assumptions. That check can be combined with checkpointing and rollback so that teams can recover from missed detections rather than only explain them after the fact. Because state corruption often becomes visible later than the original write, rollback is not a luxury; it is part of basic incident response for long-running agents.
- *Layer 4: Verify the plan before the action.* Decision controls must evaluate whether the agent's proposed trajectory still matches the user's authorization. Syntactic constraints, such as JSON schemas, can help structure tool calls, but they do not resolve semantic drift. Teams also need plan-level verification: Is the agent still diagnosing, or has it quietly crossed into reconfiguration? Does the next step actually serve the authorized goal, or has it introduced a parasitic objective? Independent verifier models, hard policy constraints, and escalation thresholds for sensitive actions are all useful here because planning errors often look coherent until viewed as a sequence.
- *Layer 5: Assume upstream controls will sometimes fail.* Execution remains the final containment boundary. This is where traditional system security techniques still matter most: seccomp, eBPF, containers, capability scoping, restricted filesystems, network controls, and runtime anomaly monitoring. But those controls should operate over trajectories, not isolated commands. If the agent creates a script in pieces and then runs it, the execution layer should evaluate the chain as one behavior. For irreversible actions, it should also be able to pause and hand the decision to a human with clear provenance and risk context.

Deployment-Oriented Defense Evaluation

We evaluated OpenClaw with DeepSeek-V4-Pro, GLM-5, and Qwen3.6-plus across 7,770 recorded multiturn runs covering six scenarios: file management, web browsing, code development, system administration, document review, and data analysis. Table 2 reports the average results across the three model configurations. Without defense, the system achieves an ASR of 30.8% and a utility of 95.7%. Among the individual defense layers, initialization introduces the lowest latency at 29 ms but retains an ASR of 21.9%. The decision layer achieves the lowest ASR at 9.2%. The input layer provides the best balance, with 10.1% ASR, 3.3% FPR, and 88.3% utility. The inference layer preserves the highest

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:32:47 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Table 2. Security, utility, and latency averaged across three OpenClaw configurations.

	ASR ↓	FPR ↓	Utility ↑	Latency (ms) ↓
No defense	30.8%	0.0%	95.7%	0
Initialization	21.9%	6.8%	85.1%	29
Input	10.1%	3.3%	88.3%	267
Inference	16.2%	1.6%	91.3%	217
Decision	9.2%	5%	86%	242
Execution	11.6%	13.6%	71.2%	59
Combined	7.2%	6.4%	85.6%	649

FPR denotes benign cases routed to user confirmation.

utility at 91.3% and the lowest FPR at 1.6%, although its ASR remains higher at 16.2%. The execution layer reduces ASR to 11.6% but produces the highest FPR at 13.6% and the lowest utility at 71.2%, highlighting the limitations of relying primarily on the final enforcement boundary. The combined defense achieves the strongest overall performance. It reduces ASR to 7.2%, corresponding to a relative reduction of 76.6%, while maintaining 85.6% utility, a 6.4% FPR, and 649-ms latency. Since false-positive cases are forwarded to users for confirmation rather than rejected outright, the FPR mainly reflects additional review overhead rather than permanent task failure.

Why the Layers Need Shared Context

The aforementioned layers are useful individually; they become much more effective when they share trust information. A plugin that barely passed onboarding should be treated differently from one with strong provenance. Content retrieved from an untrusted website should receive more scrutiny than content generated by a trusted internal tool. A memory write derived from suspicious input should be marked accordingly, and any plan that depends on that write should inherit the warning. Without this trust propagation, every stage reinterprets the world from scratch and attackers get repeated opportunities to reset the defender's assumptions.

Shared context also makes recovery possible. If runtime monitoring detects a harmful execution pattern, responders need to know whether the chain began with a tool, an input, a memory update, or a planning error. Checkpoints and provenance labels make that forensic path shorter. More importantly, they let the system roll back to the last verified-good state instead of leaving teams to reconstruct the incident manually. For autonomous agents, rollback is not only an operational convenience; it is one of the few ways to contain attacks that unfold gradually across many turns.

Where Today's Defenses Still Break Down

The agent-security literature is moving quickly, and many ideas are promising. OpenClaw highlights a more uncomfortable reality: Many of today's defenses still assume that attacks are short-lived, local, and easy to assign to one interface. Autonomous agents violate all three assumptions. Recent work evaluates OpenClaw using 47 adversarial scenarios across six MITRE-derived categories and a human-in-the-loop tool-call defense.[13] Our study complements this execution-focused evaluation by tracing failures across the full agent lifecycle and deriving cross-stage defenses and risk-based deployment priorities. Traditional controls, such as supply-chain integrity, least privilege, and sandboxing remain essential. What is specific to LLM agents is that untrusted natural-language content can steer tools, persist through writable memory, reshape autonomous plans, and accumulate across multistep trajectories. Our lifecycle framing therefore explains how established risks propagate and gain authority across stages, rather than relabeling them as new vulnerabilities.

Initialization Controls Are Still Too Static

Plugin vetting and reputation-based onboarding are necessary, but they only tell defenders what a tool looked like at approval time. That is a weak guarantee in systems where tools combine executable code, natural-language descriptions, metadata used for routing, and external dependencies that may change after deployment.[10] Static checks are good at catching bluntly dangerous artifacts. They are weaker against tools whose malicious behavior is activated by context, update drift, or privileged configuration that looked acceptable in isolation. The result is a false sense of

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:32:47 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

confidence at exactly the moment when the agent's trust boundary is being established.

Input Guardrails Often Assume a Single-Turn Chat Model

Many prompt-defense techniques still treat the problem as if the system were evaluating a single prompt-response pair.[6,7] That framing misses how autonomous agents actually operate. Retrieved content can arrive several steps after the initial request. Tool output can reshape the context window. A benign-looking input in one turn can become the justification for a harmful memory write in the next. When defenses are tuned for isolated prompts, they miss the temporal composition that makes agent attacks durable.[2,9] In practice, a guardrail can look effective in unit tests and still fail under realistic workflows.

Memory Remains an Underprotected First-Class Attack Surface

Perhaps the largest mismatch between current defenses and deployed systems lies in memory. Agents are increasingly expected to retain user preferences, intermediate findings, and task state across sessions. Yet memory protections often lag behind input protections. Teams screen what enters the context window more carefully than what becomes long-term state. Existing mitigations for drift or model misalignment are helpful, but they are often reactive rather than stateful.[3,11] Once a poisoned instruction is stored as "knowledge," the system needs continuous validation, provenance, and recovery. Many deployments still lack all three.

Plan Validation Is Improving, But Still Ad Hoc

OpenClaw's intent-drift example shows why the planning stage cannot be treated as a transparent by-product of the model. Even when the model is not being directly jailbroken, it can still construct action sequences that exceed what the user intended. Existing evaluation environments and red-teaming frameworks are useful for surfacing this risk,[14] but real-time mitigation remains immature. Teams may validate tool schemas or redact obviously risky outputs, yet they often lack a dedicated mechanism that asks whether the plan itself still matches the task. This gap is especially visible in agents that operate on broad instructions, such as "fix," "clean up," or "eliminate the risk quickly." Those are exactly the situations where an autonomous system most needs explicit authorization boundaries.

Sandboxing Is Necessary, But Not Sufficient

Execution controls continue to do essential work, especially when upstream layers miss a compromise. But modern agent attacks are increasingly designed to survive naive sandbox assumptions. Public reports and competition results show that sandbox escape, privilege abuse, and staged action chains remain practical concerns.[5,8,12] Even when the sandbox blocks the final payload, it may do little to explain how the agent got there or whether state has already been corrupted. Sandboxing answers "Can this action run?" It does not answer "Should this action have been proposed at all?" Treating sandboxing as the whole solution asks the final layer to compensate for failures that began much earlier.

The Real Weakness Is Fragmentation

The most important lesson across these stages is that fragmentation is itself a vulnerability.[15] Teams often deploy guardrails, alignment techniques, memory stores, and runtime isolation as separate features owned by different components. Attackers benefit because they can target the seams between them. A poisoned plugin can undermine the assumptions used by the input filter. A retrieved document can create the memory state that later biases planning. A plan can remain internally coherent while moving toward an action that the runtime eventually has to block. When defenses do not share provenance, state, and policy, every boundary becomes a place where malicious context can be laundered into something that looks routine.

This asymmetry favors the attacker. The attacker needs to exploit only one stage successfully, while the defender must preserve trustworthy behavior across the whole lifecycle. Autonomy amplifies the problem because the same capabilities that create user value also create attack primitives. Persistent memory supports continuity and persistence for the attacker. Tool orchestration supports productivity and lateral movement. Privileged execution supports automation and destructive action. The challenge is not to eliminate autonomy, but to govern it more explicitly than today's piecemeal defenses allow.

Deployment Priorities and Tradeoffs

Defense deployment should follow two factors: which agent surfaces are exposed and how costly a failure would be. Table 2 shows why a single universal control is insufficient. Input and decision defenses provide the largest single-layer ASR reductions, inference defenses preserve the most utility, and execution-only enforcement incurs the greatest false-positive and utility costs. This supports an upstream-first strategy, with execution controls as a backstop and the combined policy reserved for higher-risk workflows.

A Staged Deployment Order

- *Tier 1: Establish a minimum baseline for every agent.* Inventory all plugins and skills, require signatures or verifiable provenance, pin dependencies, and reject

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:32:47 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

mismatches between declared purpose and observed behavior. Apply least-privilege tool permissions, isolate execution, and log every state-changing action. These controls should be deployed first because they are operationally mature, independent of model behavior, and limit the blast radius when later checks fail.

- *Tier 2: Protect the surfaces the agent actually uses.* Agents that browse the web or process external files need explicit separation between data and instructions. Agents with persistent memory need write validation, provenance labels, and checkpoints before durable updates. Agents acting on broad objectives need plan verification before invoking sensitive tools. Resource-constrained teams should therefore select controls according to enabled capabilities rather than deploy every semantic classifier uniformly.
- *Tier 3: Combine controls for privileged workflows.* Agents that can modify services or configurations, access credentials, transmit sensitive data, or execute generated code require plan verification, explicit user approval, trajectory-aware runtime monitoring, and rollback. Execution containment remains essential, but it should not be the only boundary: By the time a harmful command reaches the runtime, the plugin, input, memory, or plan may already be compromised.

Match Control Strength to Consequence

Low-risk, read-only tasks, such as code search or document summarization, can proceed automatically under scoped permissions and audit logging. Reversible state changes should require a policy check and checkpoint, with confirmation triggered by anomalous context or scope expansion. Irreversible, externally visible, or high-blast-radius actions should pause by default for explicit authorization.[13] A practical rule is that controls should become stricter as an action becomes more persistent, privileged, externally visible, or difficult to reverse.

This risk-based gating also limits unnecessary friction. In Table 2, false positives denote benign actions routed to user confirmation rather than permanently rejected. Teams should therefore tune confirmation thresholds by action class: tolerate more review for infrastructure, credential, and memory changes, while allowing routine low-impact actions to remain automated.

Roll Out and Validate Incrementally

Teams should deploy provenance, permission scoping, logging, and containment immediately. Semantic input, memory, and plan checks can first run in shadow mode on low- and medium-risk workflows to measure false positives, latency, and utility before enforcement. High-risk actions should require approval from the outset. A deployment should advance to broader automation only after critical actions are covered, review burden is acceptable, policy overrides are auditable, and rollback has been tested. Evaluation should include cross-stage tests that verify instruction hierarchy, memory integrity, plan authorization, multistep execution detection, and recovery from a compromised state.

What Needs To Change Next

Three shifts would make agent security more defensible in practice. First, defenders need stronger roots of trust. Today, provenance for plugins, prompts, memory, and execution state is often maintained in software and reconstructed after the fact. Hardware-assisted approaches could make that chain harder to tamper with, especially for the small trusted core that handles identity, memory governance, and policy enforcement. The appeal is not only stronger isolation; it is a cleaner answer to the question of what the agent should trust in the first place.

Second, security policies will need to become more adaptive without becoming opaque. Static thresholds are easy to configure and easy for attackers to probe. But a fully dynamic policy that changes without clear justification can be just as hard for operators to manage. The most promising direction is likely a hybrid one: hard boundaries for high-consequence actions, combined with adaptive scrutiny for ambiguous inputs, memory updates, and long-horizon plans. That balance matters because autonomy is valuable only if operators can still understand when and why the system was allowed to act.

Third, the field still lacks good ways to measure security posture for agents. Traditional software metrics do not capture prompt authority, memory integrity, or cross-stage trust propagation. Benchmark scores alone are inadequate because they reward successful task completion more than safe task completion. Practitioners need evaluation methods that quantify not just whether the agent can finish the task, but whether it can do so while preserving provenance, resisting state corruption, and staying within authorization boundaries.

Autonomous agents are likely to become common in software engineering, operations, and security response whether defenders are ready or not. OpenClaw illustrates both the opportunity and the cost of that shift. These systems are useful precisely because they remember, plan, and act. Those same qualities also make them hard to secure with prompt filters alone. The path forward is not less autonomy. It is autonomy that is auditable, policy-aware, and recoverable across the full lifecycle of the system. ■

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:32:47 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Acknowledgment

This work was supported in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under Grant JYB2025XDXM114; the National Natural Science Foundation of China under Grants 62132011 and 62425201; and by the Ant Group Postdoctoral Program.

References

1. P. Steinberger and the OpenClaw Contributors. "OpenClaw: Personal AI assistant." GitHub. Accessed Mar. 5, 2026. [Online]. Available: https://github.com/openclaw/openclaw
2. K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in *Proc. 16th ACM Workshop Artif. Intell. Secur.*, New York, NY, USA: ACM, 2023, pp. 79–90, doi: 10.1145/3605764.3623985.
3. B. D. Sunil, I. Sinha, P. Maheshwari, S. Todmal, S. Mallik, and S. Mishra, "Memory poisoning attack and defense on memory-based LLM agents," 2026, *arXiv:2601.05504*.
4. X. Deng, J. Wu, M. Chen, Y. Xiao, K. Xu, and Q. Li, "Automating agent hijacking via structural template injection," 2026, *arXiv:2602.16958*.
5. A. Zou et al., "Security challenges in AI agent deployment: Insights from a large scale public competition," 2025, *arXiv:2507.20526*.
6. S. Chen, J. Piet, C. Sitawarin, and D. Wagner, "StruQ: Defending against prompt injection with structured queries," in *Proc. 34th USENIX Conf. Secur. Symp., (SEC)*, Berkeley, CA, USA: USENIX Association, 2025, pp. 2383–2400.
7. S. Chen, A. Zharmagambetov, S. Mahloujifar, K. Chaudhuri, D. Wagner, and C. Guo, "SecAlign: Defending against prompt injection with preference optimization," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA: ACM, 2025, pp. 2833–2847, doi: 10.1145/3719027.3744836.
8. D. Bors, "Available: Accessed Escaping the agent: On ways to bypass OpenClaw's security sandbox," *Snyk Labs*, Feb. 26, 2026. Accessed: May 5, 2026. [Online]. Available: https://labs.snyk.io/resources/bypass-openclaw-security-sandbox/
9. S. Yi et al., "Jailbreak attacks and defenses against large language models: A survey," 2024, arXiv:2407.04295.
10. Y. Liu et al., "Agent skills in the wild: An empirical study of security vulnerabilities at scale," 2026, *arXiv:2601.10338*.
11. V. Dongre, R. A. Rossi, V. D. Lai, D. S. Yoon, D. Hakkani-Tür, and T. Bui, "Drift no more? Context equilibria in multi-turn LLM interactions," 2025, *arXiv:2510.07777*.
12. Z. Wang et al., "AgentVigil: Generic black-box red-teaming for indirect prompt injection against LLM agents," 2025, *arXiv:2505.05849*.
13. Z. Shan, J. Xin, Y. Zhang, and M. Xu, "Don't let the claw grip your hand: A security analysis and defense framework for OpenClaw," 2026, *arXiv:2603.10387*.
14. E. Debenedetti, J. Zhang, M. Balunovic, L. Beurer-Kellner, M. Fischer, and F. Tramèr, "AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents," in *Proc. 38th Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2024, pp. 82,895–82,920.
15. W. Luo et al., "AGrail: A lifelong agent guardrail with effective and adaptive safety detection," in *Proc. 63rd Annu. Meeting Assoc. Comput. Linguistics (Long Papers)*, vol. 1, Vienna, Austria: Association for Computational Linguistics, 2025, pp. 8104–8139, doi: 10.18653/v1/2025.acl-long.399. [Online]. Available: https://aclanthology.org/2025.acl-long.399/

Xinhao Deng is an artificial Intelligence security researcher at the Ant Group, Hangzhou 310000, China, and a joint postdoctoral researcher at Tsinghua University, Beijing 100084, China. Deng received his Ph.D. in cyberspace security from Tsinghua University. His research interests include AI security and network security. Contact him at xinhaodeng.thu@gmail.com.

Yixiang Zhang is pursuing a Ph.D. at the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China. His research interests include AI security and network security. Zhang received his B. Eng. in computer science and technology from Tsinghua University. Contact him at zhangyix19@tsinghua.org.cn.

Jiaqing Wu is pursuing a Ph.D. degree at the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China. His research interests include AI security and network security. Wu received his B. Eng. in computer science and technology from Tsinghua University. Contact him at wu-jq22@mails.tsinghua.edu.cn.

Jiaqi Bai is pursuing a Master's degree from Tsinghua University, Beijing 100084, China. Her research interests include AI security and network security. Bai received her B. Eng. in computer science and technology from Tsinghua University. Contact her at bjq21@mails.tsinghua.edu.cn.

Sibo Yi is an artificial intelligence security researcher at the Ant Group, Hangzhou 310000, China. His

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:32:47 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

research interests include AI security and network security. Yi received his M.S. in computer science and technology from Tsinghua University. Contact him at yisb.2023@tsinghua.org.cn.

Zhuoheng Zou is pursuing a Ph.D. in computer science from Tsinghua University, Beijing 100084, China. His research interests include AI security and network security. Zhou received his B.S. in computer science and technology from Tsinghua University. Contact him at zouzh1024@gmail.com.

Yue Xiao is a Ph.D. student at the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China. His research interests include AI security and network security. Xiao received his master's in cyberspace security from Wuhan University. Contact him at xiaoyue25@mails.tsinghua.edu.cn.

Rennai Qiu is pursuing a Master's from Tsinghua University, Beijing 100084, China. His research interests include AI security and network security. Qiu received his Bachelor's in computer science and technology from Tsinghua University. Contact him at qurance@gmail.com.

Jianan Ma is an artificial intelligence security researcher at the Ant Group, Hangzhou 310000, China. His research interests include AI security and formal methods. Ma received his B.Eng. in environmental engineering from Hangzhou Dianzi University. Contact him at majianannn@gmail.com.

Jialuo Chen is an artificial intelligence security researcher at the Ant Group, Hangzhou 310000, China. His research interests include AI security, large language model security, and agent security. Chen received his Ph.D. in computer science from Zhejiang University. Contact him at cjl486639@antgroup.com.

Xiaohu Du is an artificial intelligence security researcher at the Ant Group, Hangzhou 310000, China. His research interests include AI security and software engineering. Du received his Ph.D. in cyberspace security from Huazhong University of Science and Technology. Contact him at duxiaohu.dxh@antgroup.com.

Xiaofang Yang is an artificial intelligence security researcher at the Ant Group, Hangzhou 310000, China. Her research interests include agent security evaluation and guardrails, AI loss-of-control risks, and safety specifications. Yang received her bachelor's in management from Sun Yat-sen University. Contact her at xiaofang.yxf@antgroup.com.

Shiwen Cui is an artificial intelligence security researcher at the Ant Group, Hangzhou 310000, China. His research interests include trustworthy AI and large language model security. Cui received his bachelor's from Harbin Engineering University. Contact him at donn.csw@antgroup.com.

Changhua Meng is an artificial intelligence security researcher at the Ant Group, Hangzhou 310000, China. His research interests include trustworthy AI and agentic AI. Meng received his master's in finance and economics from Shanghai University. Contact him at changhua.mch@antgroup.com.

Weiqiang Wang is an artificial intelligence security researcher at the Ant Group, Hangzhou 310000, China. His research interests include trustworthy AI and agentic AI. Wang received his Ph.D. in materials science from the University of Southern California. Contact him at weiqiang.wwq@antgroup.com.

Jiaxing Song is an associate researcher with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. His research interests include computer networks and distributed information systems. Song received his Ph.D. in computer science and technology from Tsinghua University. Contact him at jxsong@tsinghua.edu.cn.

Ke Xu is a full professor with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. His research interests include next-generation Internet, blockchain systems, the Internet of Things, and network security. Xu received his Ph.D. from the Department of Computer Science and Technology at Tsinghua University. He is Fellow of IEEE. Contact him at xuke@tsinghua.edu.cn.

Qi Li is an associate professor at the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China. His research interests include Internet security, mobile security, and machine learning security. Li received his Ph.D. from the Department of Computer Science and Technology at Tsinghua University. He is Senior Member of IEEE. He is the corresponding author of this article. Contact him at qli01@tsinghua.edu.cn.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:32:47 UTC from IEEE Xplore. Restrictions apply.