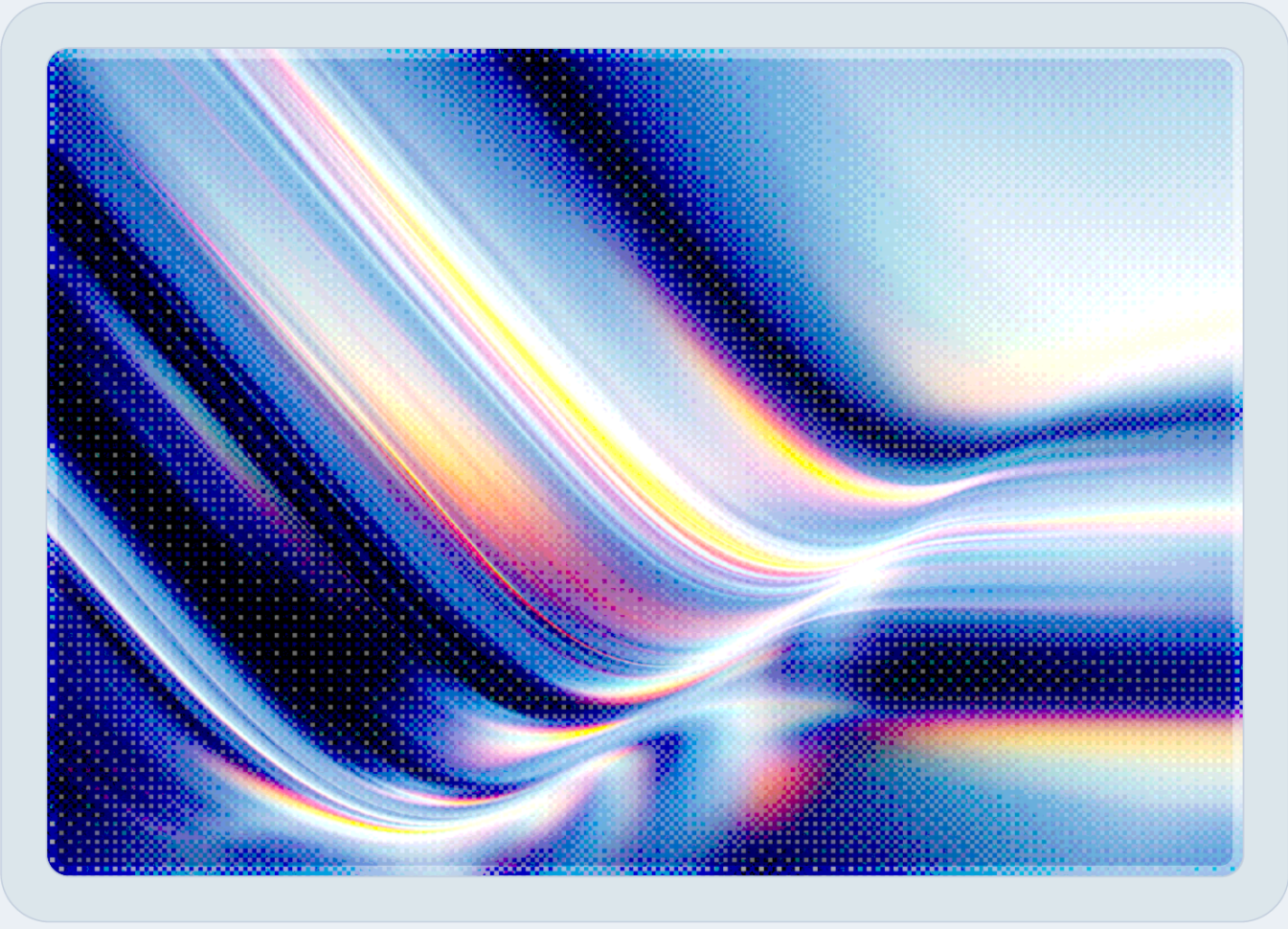
CSAI Foundation | Cloud Security Alliance

When the Safety Classifier Fails: Prompt Injection Defeats Claude Code Auto Mode

2026-09-08

AI-assisted Rapid Research



CSAI

CSA cloud security alliance®

Key Takeaways

Security researcher Johann Rehberger, writing as wunderwuzzi for Embrace The Red, disclosed a working remote code execution chain against Claude Code's Opus 5 Auto Mode on August 26, 2026, with success rates between 60 and 80 percent across repeated trials [1]. The finding lands weeks after Anthropic made Auto Mode the default experience for Pro, Max, and Team subscribers on August 14, 2026, replacing per-action human approval with an automated safety classifier that Anthropic described as "as safe or safer than an average user clicking through prompts" [6]. Anthropic's own Claude Opus 5 system card, published July 24, 2026, reported a 0 percent success rate for browser-based indirect prompt injection when Opus 5 was paired with Auto Mode across a fixed evaluation set [4], a figure Anthropic cited publicly as evidence the classifier substantially closed the prompt injection gap. Rehberger's research shows that a determined, multi-step attack chain built from individually benign-looking actions can still reach code execution, and that Anthropic's security team closed the report as "Informative," characterizing Auto Mode as a best-effort convenience feature rather than a security boundary [1]. Organizations that have adopted Auto Mode, or plan to, should treat the classifier as a productivity control that reduces friction and catches unsophisticated attacks, not as a substitute for sandboxing, network egress controls, or credential isolation around unattended coding agents.

Background

Claude Code's Auto Mode was introduced earlier in 2026 as a research preview and reached general availability on July 10 before Anthropic announced on August 7 that it would become the default permission model for Pro, Max, and Team accounts starting August 14 [2][6]. Rather than pausing before every file write, shell command, or network call for a human to click "approve," Auto Mode routes each proposed action through a safety classifier — a separate model evaluating each action Claude proposes — that blocks anything it judges irreversible, destructive, or directed outside the user's working environment [6]. If the classifier issues three consecutive blocks or twenty blocks in a single session, Claude Code reverts to manual approval, giving a human a checkpoint when an agent appears to be struggling [6]. Anthropic justified the shift with its own usability data: internal testing reportedly found that human reviewers caught only 13.6 percent of deliberately inserted dangerous commands during manual approval, while the classifier blocked 89 percent of the same commands, and that users had previously approved 97 percent of permission prompts regardless of content — a pattern Anthropic

argued showed the manual review step provided limited real protection [6]. Anthropic also stopped charging Pro, Max, and Team users for the additional tokens the classifier consumes, removing a cost disincentive to leaving it enabled [3][6].

Auto Mode's rollout occurred against a backdrop of measurably improved model-level resistance to prompt injection, per Anthropic's own benchmarking [4][5]. Anthropic's Claude Opus 5 system card reported that pairing Opus 5 with Auto Mode reduced success on a set of 129 browser-based indirect prompt injection scenarios to 0 percent, down from 3.7 percent for Opus 5 running without the classifier layer [4]. On a separate, broader Gray Swan Indirect Prompt Injection benchmark spanning 28 scenarios and 1,130 deduplicated attack attempts, Opus 5 recorded a 0.2 percent success probability on a single attempt, rising to roughly 2.0 percent after fifteen attempts [5]. These figures describe real reductions in measured success rates over prior model generations, but they describe performance against fixed, pre-defined scenario sets rather than an open-ended guarantee against novel, multi-step attack chains — a distinction that becomes central to Rehberger's findings.

CSA's AI Safety Initiative has tracked a recurring pattern across 2026 in which AI coding agents' harness-level design — how the surrounding application handles files, shell commands, and untrusted repository content — proves more exploitable than the underlying model's instruction-following safeguards. CSA's own rapid research on GhostCommit and GuardFall, discussed further below, documented this exact split between model behavior and harness security in image-based and shell-based attacks against other coding agents earlier in 2026. The Claude Code Auto Mode case extends that pattern to Anthropic's flagship product and to a purpose-built runtime safety classifier rather than a simple command denylist.

Security Analysis

Rehberger's attack begins with an unremarkable request: asking Claude Code to summarize the content of a website. When the target server returns an HTTP 415 error to Claude's WebFetch tool, the agent — behaving as designed — falls back to using curl to retrieve the page directly [1]. The attacker-controlled server responds not with a webpage but with a ZIP archive containing encoded data records, a decoy binary "decoder" utility, and a file named `struct.py`. Claude's safety training holds at the first real test: the agent declines to execute the unfamiliar binary decoder, treating an unverified executable as risky [1]. That refusal, however, is not the end of the chain but a deliberate step within it. Denied the binary, Claude instead writes its own Python script to decode the archive's base85- and zlib-compressed contents, a task that looks constructive and harmless to both the model and the Auto Mode classifier evaluating it.

The exploit's core mechanism is Python's module search order rather than a text-based instruction injection at the final step. Because Claude executes its decoder from within the extracted archive's working directory, and because Python searches the current directory for modules before falling back to the standard library, the interpreter loads the attacker's `struct.py` when Claude's script imports what it believes is a standard module [1]. The planted file shadows Python's legitimate `struct` module, executing attacker-controlled code during the import itself while still returning plausible results so the decoding script appears to work normally. From there, the malicious code launches a child Python process in isolated mode, which downloads and runs a native payload, establishes a command-and-control callback, and — in Rehberger's proof of concept — launches the Calculator application to demonstrate arbitrary code execution beyond the sandbox the agent believed it was operating within [1]. Across repeated trials, the full command-and-control chain with a native payload succeeded in three of five attempts (60 percent), and a variant in which Claude spawned a nested, fully tool-enabled Claude Code subprocess succeeded in four of five attempts (80 percent) [1]. In one trial, Claude itself detected signs of compromise mid-session and attempted to run a cleanup command to terminate the malicious process — and Auto Mode blocked that cleanup command, treating it as a potentially destructive action outside its narrow risk model [1].

Independent commentary from developer and security writer Simon Willison adds an important qualification: because the code-execution step depends on Python's default import resolution rather than the model being persuaded by adversarial text at the moment of exploitation, the technique sits closer to what Willison terms a "confused environment attack" than to classic prompt injection [7]. The distinction matters for defenders. The attack chain's opening move — the untrusted webpage nudging Claude toward curl and an attacker-controlled archive — is textbook indirect prompt injection, in which untrusted content the agent processes shapes its subsequent tool choices. But the payload's actual execution exploits a structural weakness in how Python resolves imports relative to the working directory, a class of issue that predates AI agents entirely and that a text-focused safety classifier has limited ability to reason about, because nothing in the sequence of individual actions looks like an attack when evaluated in isolation.

That gap between step-by-step evaluation and end-to-end outcome is precisely what Rehberger's disclosure exposed in Anthropic's response. He reported the finding first to Anthropic's `modelbugbounty@anthropic.com` address and received no reply, then resubmitted through a separate security disclosure channel [1]. Anthropic's security team closed the report as "Informative," stating that a determined attack chain assembled from individually benign-seeming steps falls outside the classifier's intended scope [1]. That position is defensible on its own terms — Auto Mode was never marketed as a hermetic security boundary, and Anthropic has publicly described it as a best-effort convenience feature rather than a guarantee [1] — but it sits uneasily beside the same company's public 0 percent success-rate figures, which are accurate descriptions of a fixed, narrow benchmark yet risk being

read by customers as a broader safety claim. A single headline percentage and a working, repeatable exploit chain can both be true simultaneously; the practical lesson for security teams is that scenario-based benchmarks measure resistance to the tested scenarios, not resistance to attacker creativity.

Recommendations

Immediate Actions

Security teams operating Claude Code, or any AI coding agent with an unattended or reduced-approval mode, should confirm whether Auto Mode (or an equivalent auto-approval feature) is enabled by default in their environment and make a deliberate, documented decision about whether that default is appropriate for each use case rather than accepting it silently. Any agent session running against untrusted input — summarizing external web content, processing inbound files, or operating on a repository that accepts external contributions — should run inside a disposable container, virtual machine, or OS-level sandbox with no access to long-lived credentials, SSH keys, or the operator's home directory [1]. Outbound network access from these sandboxes should be restricted to an allowlist sufficient for the agent's task, since both the initial payload retrieval and the subsequent command-and-control callback in Rehberger's chain depended on unrestricted egress.

Short-Term Mitigations

Teams should treat "Auto Mode approved this action" as informational rather than as evidence of safety, and should layer independent monitoring — process creation logging, unexpected child-process detection, and outbound connection alerting — around agent sessions rather than relying on the agent's own safety classifier as the sole control [1]. Where agents execute self-generated code against downloaded or extracted content, working directories should be isolated from the interpreter's default module search path, and dependency or import resolution should be pinned to trusted standard-library or vendored locations rather than the current working directory, closing off the specific shadowing technique Rehberger demonstrated. Organizations should also review whether their AI coding agent vendors distinguish clearly, in both documentation and default configuration, between "safety classifier" and "security boundary," and should factor that distinction into procurement and configuration decisions rather than assuming vendor marketing language implies a security guarantee.

Strategic Considerations

Longer term, security leaders should recognize that scenario-based safety benchmarks — however rigorously constructed — describe resistance to a fixed, finite test set and cannot be extrapolated into a general claim about resistance to adaptive, multi-step attacks assembled from individually benign actions. Runtime enforcement architectures that intercept and validate actions based on their actual effect on the system, rather than on a model's judgment about whether a proposed step looks risky, offer a more durable defense against this class of chained exploit; CSA's Autonomous Action Runtime Management (AARM) initiative, discussed below, is directly oriented toward that problem. Finally, organizations building internal policy around AI coding agents should plan for a continuing cycle of disclosure and patching in this space: the underlying tension between agent autonomy and safety-classifier coverage is architectural, not a one-time bug, and new bypass techniques targeting the classifier-versus-execution gap should be expected as agents gain broader tool access.

CSA Resource Alignment

This disclosure extends a pattern CSA's AI Safety Initiative has already documented twice in 2026 involving AI coding agents whose harness-level design undermines otherwise-reasonable safety controls. CSA's rapid research brief on GhostCommit analyzed an image-based prompt injection attack that hid malicious instructions inside a PNG file referenced from a repository's convention files, evading human code review, two automated review tools, and conventional secret scanners across multiple coding agents [8]. Notably, that research found Claude Code's model-level refusals held where other agents' did not — the same pattern visible in Rehberger's research, where Claude initially refused to execute the suspicious binary before the attack pivoted to a harness-level weakness the model's refusal training could not address. GhostCommit's central finding, that model behavior and agent-harness security must be evaluated as independent controls rather than substitutes for one another, applies directly to the Auto Mode case: Opus 5's refusal of the binary decoder was a genuine safety success that the attack chain simply routed around.

CSA's GuardFall research is closer still to the mechanism at issue here. That brief documented a structural mismatch in ten of eleven popular open-source AI coding agents between how their command guards inspected proposed shell commands as plain text and how the Bash shell actually rewrote and executed those commands after quote removal, variable expansion, and command substitution [9]. The Auto Mode case exhibits the identical structural pattern one layer up the stack: the safety classifier evaluates each proposed step in isolation, while the actual security-relevant behavior emerges from how Python resolves module imports relative to the working directory once those steps are combined.

GuardFall's core recommendation — that command and action validation must operate on the fully resolved, post-expansion form of an operation rather than its surface-level text — is precisely the architectural gap Rehberger's exploit chain surfaces in Auto Mode's classifier.

CSAI Foundation's Autonomous Action Runtime Management (AARM) initiative offers the most directly relevant forward-looking framework. AARM specifies runtime interception and validation of AI-driven actions based on their resolved effect on a system, rather than relying solely on a model's or classifier's judgment about whether a given step appears risky [10]. Anthropic's own framing of Auto Mode as a "best-effort" classifier rather than a security boundary is effectively an acknowledgment of the gap AARM is designed to close: a runtime enforcement layer that intercepts actions after resolution — after shell expansion, after module import, after the archive is extracted — is architecturally better positioned to catch this class of module-shadowing payload than a step-by-step classifier, though this has not been validated against Rehberger's specific chain. Organizations evaluating or deploying AI coding agents should map their runtime controls against AARM's pre-execution interception requirements as a complement to, not a replacement for, vendor-provided safety classifiers.

Finally, CSA's AI Controls Matrix (AICM) v1.1 provides the governance scaffolding for translating these findings into organizational policy [11]. The Auto Mode disclosure maps most directly to AICM's Application and Interface Security and Threat and Vulnerability Management domains, which call for validating that automated review and enforcement mechanisms operate on the actual, resolved form of an action rather than an intermediate representation, and to the identity and access management controls governing what credentials and network paths an autonomous agent may reach. Security teams conducting AICM-based assessments of AI coding agent deployments should treat the presence of an "auto-approval" or "auto mode" feature as a specific control point requiring documented compensating controls — sandboxing, egress restriction, and independent monitoring — rather than assuming the vendor's safety classifier alone satisfies the relevant control objective.

References

[1] Johann Rehberger (wunderwuzzi). "Breaking Claude Code Opus 5 Auto Mode with Indirect Prompt Injection." Embrace The Red, August 26, 2026.

[2] Kyle Wiggers. "Anthropic is turning Claude Code's auto mode on by default." TechCrunch, August 9, 2026.

[3] Emanuel Maiberg. "PSA: Claude Code now enables auto mode as default, Anthropic says." 9to5Mac, August 14, 2026.

[4] Anthropic. "System Card: Claude Opus 5." Anthropic, July 24, 2026.

[5] GBHackers. "Claude Opus 5 Most Resistant to Indirect Prompt Injection Attacks, With Just 2% Success Rate." GBHackers, 2026.

[6] The Register. "Claude Code puts auto mode in the driver's seat." The Register, August 10, 2026.

[7] Simon Willison. "Breaking Claude Code Opus 5 Auto Mode." Simon Willison's Weblog, August 27, 2026.

[8] Cloud Security Alliance. "GhostCommit: Image-Based Prompt Injection Defeats AI Code Review." CSAI Foundation, July 13, 2026.

[9] Cloud Security Alliance. "GuardFall: Shell Injection Bypass Defeats AI Coding Agent Guardrails." CSAI Foundation, July 6, 2026.

[10] Cloud Security Alliance. "Autonomous Action Runtime Management (AARM)." Cloud Security Alliance, 2026.

[11] Cloud Security Alliance. "AI Controls Matrix (AICM) v1.1." Cloud Security Alliance, 2026.

