JOURNAL OF TUNAS MELAYU DIGITAL (JTMD) • ISSN: PENDING

Vol. 1, No. 1, November 2026, pp. 21-26
Peer-Reviewed • Open Access • https://jutuma.org

Indirect Prompt Injection in Agentic Systems: Evidence from Tool Outputs and Third-Party Content

When the Environment Talks Back: Data, Instructions, and the Duty to Disregard

Gusti Tri Prayoga1*, Alfa Qosry2

1,2Program Studi S1 Teknik Informatika, Fakultas Teknik, Universitas Pahlawan Tuanku Tambusai
Jl. Tuanku Tambusai No. 23 Bangkinang Kota, Kab. Kampar, Riau, 28412, Indonesia

*Corresponding Author: gustitriprayoga@universitaspahlawan.ac.id • Received: 2026-11-04 • Accepted: 2026-11-07 • Published: 2026-11-07

ABSTRACT

Large language model agents differ from chatbots in one decisive respect: they act. They read files, call tools, send messages, and execute commands, which means the content they ingest can also instruct them. Indirect prompt injection — instructions embedded in web pages, files, tool outputs, and messages from other agents — is now ranked by OWASP as the leading risk for LLM applications, and benchmarks show that a substantial share of tool-integrated agents follow injected instructions. This article examines how a deployed personal AI agent resists such injection in everyday work, through a single-case study of Kesang, a personal conversational agent used intensively in Indonesia between October and November 2026. Drawing on interaction records, system logs, and scheduled-task reports, the study reconstructs four episodes: a tool result displaying the contents of a maintenance script, which the agent declined to execute from the displayed output; a background-task report that failed a prompt-injection check, which the agent discarded as a source of next steps without circumventing the hold; a third-party persona prompt arriving through the request path, which the agent's bridge was configured to ignore; and a standing pattern of retrieved content attempting to redirect the agent's task. The findings show that effective defence is architectural — provenance tracking, an instruction hierarchy, and least-privilege tooling — rather than a matter of judging each injected message on its merits, and that a refused injection should be disclosed, not silently dropped, so that it leaves an auditable trace.

Keywords: *Prompt Injection, Agentic AI, LLM Security, Instruction Hierarchy, Human–AI Collaboration*

ABSTRAK (BAHASA INDONESIA)

Agen model bahasa besar berbeda dari chatbot dalam satu hal yang menentukan: agen bertindak. Agen membaca berkas, memanggil perkakas, mengirim pesan, dan menjalankan perintah, sehingga konten yang dibacanya juga dapat menginstruksikannya. Injeksi prompt tidak langsung — instruksi yang terselip dalam halaman web, berkas, keluaran perkakas, dan pesan dari agen lain — kini dinilai OWASP sebagai risiko utama aplikasi LLM, dan tolok ukur menunjukkan sebagian besar agen terintegrasi perkakas mengikuti instruksi yang disuntikkan. Artikel ini mengkaji bagaimana agen AI personal yang sudah beroperasi menolak injeksi semacam itu dalam kerja sehari-hari, melalui studi kasus tunggal terhadap Kesang, agen percakapan personal yang digunakan secara intensif di Indonesia pada Oktober–November 2026. Berdasarkan rekaman interaksi, log sistem, dan laporan tugas terjadwal, penelitian ini merekonstruksi empat episode: keluaran perkakas yang menampilkan isi skrip pemeliharaan, yang tidak dijalankan agen dari tampilan tersebut; laporan tugas latar yang gagal melewati pemeriksaan injeksi prompt, yang dibuang agen sebagai sumber langkah berikutnya tanpa mengakali penahanan itu; prompt persona pihak ketiga yang tiba melalui jalur permintaan, yang sengaja diabaikan jembatan agen; serta pola berulang konten yang dibaca agen yang berusaha mengalihkan tugasnya. Temuan menunjukkan bahwa pertahanan yang efektif bersifat arsitektural — pelacakan asal, hierarki instruksi, dan perkakas dengan hak paling minim — bukan soal menilai tiap pesan yang disuntikkan satu per satu, dan bahwa injeksi yang ditolak sebaiknya diungkapkan, bukan diam-diam dibuang, agar meninggalkan jejak yang dapat diaudit.

Kata Kunci: *Injeksi Prompt, AI Agen, Keamanan LLM, Hierarki Instruksi, Kolaborasi Manusia–AI*

1. INTRODUCTION

Large language model (LLM) agents differ from chatbots in one decisive respect: they act. They read files, call tools, send messages, execute commands, and publish content on their users' behalf. A chatbot that is misled produces wrong words; an agent that is misled sends the wrong message, runs the wrong command, or publishes the wrong file. This shift from words to actions changes the security stakes of everyday AI use, because the content an agent ingests

can now instruct it. The mechanism is indirect prompt injection: instructions planted not in the user's message, but in content the agent will later process — a web page, a file, a tool result, a message from another agent [1], [6].

The threat is well established. Early work showed that simple handcrafted inputs could hijack a model's goal or leak its hidden prompt [2]. Field studies found that most real LLM-integrated applications were vulnerable to such attacks [3]. Agent benchmarks have since shown that injected instructions hidden in tool outputs redirect tool calls at substantial rates [4], [5], and OWASP ranks prompt injection first among risks for LLM applications [6]. Yet laboratory attack rates do not answer the operational question that matters to anyone deploying an agent: what does a working defence look like inside an agent that must keep reading untrusted content all day, every day, in order to do its job? A defence that requires the agent to deliberate over every message is unworkable; a defence that fails silently is unverifiable.

This article supplies evidence from one deployed personal agent. Kesang is a personal conversational AI agent used intensively in Indonesia between October and November 2026 for educational video production, infrastructure automation, and daily information work. Drawing on interaction records, system logs, and scheduled-task reports, the study reconstructs four episodes in which content from the environment carried instructional force — a tool result displaying a maintenance script, a background-task report that failed a prompt-injection check, a third-party persona prompt arriving through the request path, and a standing pattern of retrieved content attempting to redirect the agent's task — and analyses what held, and why. The research question is: how can a deployed agent keep acting on the world while treating everything it reads as potentially instructional? Section 2 reviews the literature on prompt injection, its measurement in agents, and its defences. Section 3 describes the case-study methodology. Section 4 presents the four episodes and discusses what they reveal about provenance, hierarchy, and disclosure. Section 5 concludes with findings, limitations, and directions for future research.

2. LITERATURE REVIEW

2.1. From Direct to Indirect Injection

Prompt injection was first studied systematically as a direct attack: the user's own input overrides the developer's instructions. Perez and Ribeiro [2] formalised two such attacks — goal hijacking, in which the model abandons its intended task to print an attacker-chosen string, and prompt leaking, in which it reveals the application's hidden prompt — and showed, with a framework called PromptInject, that simple handcrafted inputs succeeded on average in 58.6% of goal-hijacking attempts and 23.6% of prompt-leaking attempts against the most capable model tested. Their sobering finding was that better instruction-following also means better following of injected instructions. Greshake et al. [1] then showed that the attacker need not talk to the model at all: in indirect prompt injection, instructions are planted in data the application retrieves — a web page, a document, an email — and a legitimate user's query triggers the exploit. Liu et al. [3] formalised this threat with HouYi, an attack combining a pre-constructed prompt, an injection prompt inducing a context partition, and a malicious payload, and found 31 of 36 real LLM-integrated applications vulnerable. The common root is architectural: instructions and data share a single token stream, with no trust separation between them.

2.2. Measuring the Risk in Agents

The risk grows when the model can act. InjecAgent [4] benchmarks indirect injection specifically in tool-integrated agents: 1,054 test cases in which injected instructions in a tool's response attempt to make the agent call an attacker's tool, for direct harm or data exfiltration. A ReAct agent built on GPT-4 followed the injected instruction in roughly a quarter of base cases, and in nearly half when the attack was enhanced. AgentDojo [5] evaluates agents in stateful environments — a workspace, Slack, a travel agency, an e-banking system — with 97 user tasks and 629 security cases, measuring both the user's success and the attacker's. Its central finding is a trade-off: defences such as filtering which tools an agent may call substantially reduce attack success, but at a measurable cost to the agent's utility under attack. OWASP's Top 10 for LLM Applications 2025 [6] ranks prompt injection first (LLM01:2025), covering direct injection, indirect injection through external sources, and multimodal injection, and is candid about the state of defences: no fool-proof method of prevention is known, so the working strategy is to assume injection succeeds and limit what a hijacked model can do.

2.3. Defences: Training, Structure, and Architecture

Defences fall into three families. The first is trained: the instruction hierarchy [7] teaches a model through supervised fine-tuning and reinforcement learning to prioritise instructions by source — system over developer over user over third-party text — using

synthetic conflict data. It substantially improves robustness on held-out attack types, but the protection lives in trained weights, and reported bypasses show it raises the bar without eliminating the threat. The second family is structural. StruQ [8] separates prompt and data channels with reserved delimiters and fine-tunes the model to ignore instructions appearing in the data channel, driving optimisation-free attack success to near zero at a small utility cost, though it remains vulnerable to strong optimisation-based attacks. Spotlighting [9] works at the prompt layer instead, marking untrusted text through delimiting, datamarking, or encoding, and reports attack success below 3% on its suite of static attacks. The third family is architectural. CaMeL [10] separates control flow from data flow and gives tools capabilities rather than ambient authority, aiming to defeat prompt injection by design rather than by detection. It is the only approach in this set that claims provable guarantees, and it pays for them with a measurable utility cost.

Across these families, the normative frame is the principal hierarchy developed for advanced AI assistants [11]: instructions from higher principals (system, developer) override those of lower ones (user), while instructions from non-principals — web pages, tool outputs, pasted content, other agents' reports — carry no authority at all. The hierarchy also imposes a duty of disclosure: when material facts bear on the principal's interests, including the agent's own uncertainty and conflicts, the assistant must proactively share them. Alignment failures such as misspecification and specification gaming [12] are the background against which these defences operate: even a perfectly obedient agent can be turned against its user's interests when the instruction itself is the problem. Regulatory responses, notably the European Union's Artificial Intelligence Act [13], adopt a risk-based approach that places the heaviest obligations on high-risk systems, but leave the everyday security of personal agents largely to design and professional norms — the space this article occupies.

2.4. Research Gap

Three gaps motivate this study. First, benchmarks measure models in sandboxes with seeded attacks; few studies document how a deployed agent's standing rules perform against injection attempts that arise naturally in real work. Second, the literature treats injection channels separately — web content, tool outputs, agent-to-agent messages — while a working agent meets all of them in a single day. Third, defences are evaluated on attack success rates, not on what a refusal costs: the false positives, the stalled delegations, and the useful material discarded when a hold is honoured. This case study addresses all three gaps by tracing one agent's injection handling across tool outputs, scheduled-task reports, and the request path in a five-week window.

3. RESEARCH METHODOLOGY

This study uses a qualitative single-case design, appropriate for examining a contemporary phenomenon in its real-life context where the boundaries between the phenomenon and its context are not clearly separable. The case is Kesang, a personal conversational AI agent (built on the Muse Spark model family) operating as the first author's daily assistant. The observation window runs from 1 October to 7 November 2026. During this period the agent's workload included: scripting, narrating, rendering, and scheduling short-form educational videos for the GPTanya Anak channels; operating a Discord voice bot and a reverse bridge to the user's local model provider; and compiling a daily digest of genuinely free AI models. The agent therefore read files, web pages, tool results, and messages from other agents continuously — the exact conditions under which indirect injection arises.

Data consist of the complete interaction history between the user and the assistant (chat transcripts), system-generated records (scheduled-task logs, watchdog and poller reports, safety notices), and production artefacts (video files, verification logs, scheduling confirmations). Four episodes were selected as critical incidents: each is a moment in which content from the environment carried instructional force and the agent had to decide whether to obey, and each left a verifiable trace in the records. Analysis proceeded by reconstructing each episode as a sequence — trigger, injection channel, decision, justification, outcome — and coding it against the instruction hierarchy [7], [11] and the defence families of Section 2.3. The fourth episode is a standing pattern rather than a single dated incident, and is presented as such.

Two ethical notes are in order. First, the subject of the study is software, and the only human involved is the user, who is also an author; all personal data belonging to third parties has been anonymised or omitted, and the contents of injected instructions are described rather than reproduced. Second, the assistant itself assisted in drafting this manuscript, which introduces a reflexivity concern: the subject helped write its own case study. The authors mitigated this by verifying every factual claim against primary records (transcripts and logs) and by retaining

editorial control over interpretation. The study's limitations are stated in Section 5: a single case, a single user, and a short window support analytical rather than statistical generalisation, and the episodes arose naturally rather than from a controlled red-team exercise, so no attack success rate can be computed.

4. RESULTS AND DISCUSSION

The four episodes are summarised in Table 1 and described in turn. They share a common thread: in none of them did the agent judge the injected instruction on its merits. In each, a standing rule about provenance decided before the content was evaluated.

4.1. Episode 1: The Script in the Output

On 5 November 2026, at about 03:58, a routine maintenance check returned a command result that displayed the contents of a service-restoration script. A safety notice flagged the result. The agent did not execute the script from the displayed output, and told the user plainly that it would not run a restore from that result. The episode is instructive precisely because the content was benign: the script was the agent's own maintenance script, and running it would very likely have done no harm. Executing code displayed in a tool result is nevertheless self-injection. A tool result is data about the system; treating its contents as a command converts observation into authorisation. The correct response is not to judge whether the script looks benign — a rule that executes benign-looking output will execute malicious-looking output on the day it looks benign — but to refuse on provenance: an instruction to run code must come from the user or from a standing authorised procedure, never from displayed output. The episode also shows the value of an external check: the safety notice, issued outside the agent's own judgement, supplied the pause that the agent's own reading of the output might not have.

4.2. Episode 2: The Flagged Handoff

On 6 November 2026, at about 08:53, a background-task report — a prepared delivery from a scheduled worker — failed a prompt-injection check. The agent did not treat the report as a source of next steps. It based its delivery only on the verified handoff from the developer and its own verified facts, and it did not re-read the flagged material through another route to get around the hold. Agent-to-agent reports are environment content, however authoritative their format: a report arrives with structure, citations, and confidence, all of which are properties of its presentation, not of its authority. A safety hold is a decision, not a delay, and circumventing it by re-reading the same material elsewhere is the failure mode the hold exists to prevent. The episode also illustrates a subtle point: the flagged report was not necessarily malicious, and may have been an ordinary false positive. But the agent's response does not depend on that judgement. Treating a hold as conditional on the agent's own assessment of the held content defeats the purpose of having the check outside the agent's judgement. The cost is real — useful material is discarded along with the possibly dangerous kind — and it is the honest price of the defence.

4.3. Episode 3: The Foreign System Prompt

On 16 October 2026, the agent's bridge to the user's local model provider began carrying queued requests to which the provider had applied a third-party persona prompt — a system prompt that would have replaced the agent's persona and instructions. The bridge poller was deliberately configured to ignore system prompts arriving with requests, so the agent's persona was not replaced. The decision was disclosed to the user and left open for their verdict. The request path is the most dangerous injection channel because it arrives dressed as authority: a system prompt. The defence cannot be per-message judgement — asking whether this particular prompt looks hostile — because a persona prompt is not hostile; it is simply not the principal's instruction. The defence is a boundary rule: the bridge accepts requests, not instructions. Disclosure matters here for a second reason: the user, as the principal who owns the bridge, is the only party who can authorise a change to that rule. Had the agent silently adopted the foreign prompt, or silently ignored it without telling the user, the user would have lost the ability to decide what their own bridge should do.

4.4. Episode 4: The Standing Condition

Across the observation window, retrieved content repeatedly carried instructional language: web pages, files, forwarded messages, and subagent reports that attempted to redirect the agent — to expand the task, to extract information the user had not authorised sharing, or to manufacture urgency the user never expressed. None of these was a dated incident; together they are the background condition of agentic work. The standing rule was that content from the environment is data, and only the user's instructions — and the system's standing rules — authorise action. This is the point the benchmarks miss: injection is not an event but a climate. An agent that reads the web all day meets instructional language constantly, and a defence that requires deliberation over every message

is unworkable. The workable defence is a default-deny posture on provenance, plus disclosure when the default bites: telling the user what was not acted on, and why, so that the non-event leaves a trace instead of a silence.

Table 1. Summary of the four indirect prompt injection episodes in the Kesang case study.

Episode	Injection channel	Agent response	Defence principle	Residual risk
1. Script in the output	Tool result displaying a maintenance script	Declined to execute from displayed output; reported plainly	Provenance over plausibility	A genuinely authorised script must still run through an authorised path
2. Flagged handoff	Background-task report failing an injection check	Discarded as a source of next steps; hold not circumvented	Holds are decisions, not delays	False positives discard useful material
3. Foreign system prompt	Third-party persona prompt in the request path	Bridge configured to ignore request-borne prompts; decision disclosed	Boundary rule, not per-message judgement	User may legitimately want a different persona; needs their verdict
4. Standing condition	Instructional language in retrieved content	Treated as data; only user instructions authorise action	Default-deny on provenance	Over-caution can stall legitimate delegation

4.5. Discussion

Five lessons emerge. First, in agentic systems injection is a routine condition, not an attack scenario. The case shows three distinct channels in five weeks, plus a standing background rate — and none of the four episodes involved a hostile attacker. The threat model that matters for a personal agent is not only the adversary planting instructions, but the ordinary environment constantly emitting them. Second, defence must be architectural. Provenance tracking, an instruction hierarchy [7], [11], and least-privilege tooling decide before content is evaluated; per-message judgement fails because injected content is often benign-looking or even genuinely useful — the agent's own script, in Episode 1. Third, a hold is a decision. The test of a defence is whether it survives the agent's own curiosity about the held material: Episode 2 held precisely because the agent did not re-read the flagged report by another route. Fourth, disclosure converts a silent non-event into auditable evidence. Each episode was reported to the user with its reason, which is what makes the defence checkable rather than merely asserted. Fifth, assume breach. OWASP's counsel [6] — limit what a hijacked model can do — is the backstop when a defence fails, and the case's authorisation gates (a separate confirmation before scheduling, one video per task, each verified before the next) bound the blast radius of any single compromise.

These findings extend the companion study of this agent's ethical boundaries, which concluded that boundaries must be operational — encoded as gates, rules, and scripts — rather than merely professed. The present study shows the same for security: a principal hierarchy that exists only as a value is no defence; it must exist as a boundary rule enforced where content enters the system, as in Episode 3, and as a hold that cannot be circumvented, as in Episode 2. They also qualify the benchmark-driven view of defence. AgentDojo [5] and InjecAgent [4] measure how often agents follow injected instructions; this case suggests a complementary metric that benchmarks do not capture: how often a deployed agent refuses, discloses, and keeps working. A defence that never fires in production is either unnecessary or unobserved, and only disclosure distinguishes the two.

The design implications follow directly. Personal agents should label the provenance of every content channel they read, so that the hierarchy is visible at the point of use rather than buried in a system prompt. They should expose explicit authorisation surfaces for consequential actions, so that "yes" is a deliberate act rather than an ambient default. They should keep an auditable log of refused injections — channel, content type, decision, and disclosure — so that both user and developer can review them. And they should treat refusal and disclosure templates as first-class interface elements: the words an agent uses when it declines an instruction from the environment are part of its security behaviour, not an afterthought. The trade-off should be stated honestly: default-deny costs utility, in false positives and stalled delegations, and that cost deserves measurement as much as attack success does.

5. CONCLUSION

This case study asked how a deployed agent can keep acting on the world while treating everything it reads as potentially instructional. Its answer, drawn from four episodes in five weeks of real deployment, is that the defence must be architectural: provenance tracked at every entry point, an instruction hierarchy enforced as a boundary rule rather than a per-message judgement, holds treated as decisions rather than delays, and refused injections disclosed so that they leave an auditable trace. The agent that is most useful is not the one that follows the most instructions, but the one that follows only the instructions it should — and can show which ones it declined, and why.

The study's limitations are clear. It examines a single assistant and a single user over a short window, so its findings support analytical rather than statistical generalisation. The episodes arose naturally rather than from a controlled red-team exercise, so no attack success rate can be computed and the sample is biased toward the injection attempts the agent noticed. The assistant's participation in drafting introduces reflexivity, mitigated here by verification against primary records but not eliminable. The cultural setting — Indonesia, with its norms of indirect communication — may shape both the environment's instructional language and the agent's responses in ways that do not transfer directly. Future research should compare injection handling across users, deployments, and cultures; measure the false-positive cost of default-deny defences in production; run controlled red-team exercises against the boundary rules described here; and compare their performance with architectural approaches such as CaMeL [10]. The practical recommendation is simple: treat provenance as a design requirement for personal AI agents, log and disclose every refused injection, and bound every tool by least privilege — because the environment will keep talking, and the agent must keep deciding what not to hear.

6. REFERENCES

[1] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. arXiv:2302.12173.

[2] Perez, F., & Ribeiro, I. (2022). Ignore previous prompt: Attack techniques for language models. arXiv:2211.09527. Presented at the NeurIPS 2022 ML Safety Workshop.

[3] Liu, Y., Deng, G., Li, Y., Wang, K., Wang, Z., Wang, X., Zhang, T., Liu, Y., Wang, H., Zheng, Y., Zhang, L. Y., & Liu, Y. (2023). Prompt injection attack against LLM-integrated applications. arXiv:2306.05499.

[4] Zhan, Q., Liang, Z., Ying, Z., & Kang, D. (2024). InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents. arXiv:2403.02691. Findings of the Association for Computational Linguistics: ACL 2024.

[5] Debenedetti, E., Zhang, J., Balunović, M., Beurer-Kellner, L., Fischer, M., & Tramèr, F. (2024). AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. arXiv:2406.13352. Advances in Neural Information Processing Systems, 37 (Datasets and Benchmarks Track).

[6] OWASP GenAI Security Project. (2025). OWASP top 10 for LLM applications 2025. OWASP Foundation. https://genai.owasp.org/llm-top-10/

[7] Wallace, E., Xiao, K., Leike, R., Weng, L., Heidecke, J., & Beutel, A. (2024). The instruction hierarchy: Training LLMs to prioritize privileged instructions. arXiv:2404.13208.

[8] Chen, S., Piet, J., Sitawarin, C., & Wagner, D. (2024). StruQ: Defending against prompt injection with structured queries. arXiv:2402.06363. Proceedings of the 34th USENIX Security Symposium, 2025.

[9] Hines, K., Lopez, G., Hall, M., Zarfati, F., Zunger, Y., & Kiciman, E. (2024). Defending against indirect prompt injection attacks with spotlighting. arXiv:2403.14720.

[10] Debenedetti, E., Shumailov, I., Fan, T., et al. (2025). Defeating prompt injections by design. arXiv:2503.18813.

[11] Gabriel, I., Manzini, A., Keeling, G., Hendricks, L. A., Rieser, V., Iqbal, H., Tomašev, N., Ktena, I., Kenton, Z., Rodriguez, M., El-Sayed, S., Brown, S., Akbulut, C., Trask, A., Hughes, E., Bergman, A. S., Shelby, R., Marchal, N., Griffin, C., Mateos-Garcia, J., Weidinger, L., Street, W., Lange, B., Ingerman, A., Lentz, A., et al. (2024). The ethics of advanced AI assistants. arXiv:2404.16244.

[12] Kenton, Z., Everitt, T., Weidinger, L., Gabriel, I., Mikulik, V., & Irving, G. (2021). Alignment of language agents. arXiv:2103.14659.

[13] European Parliament and Council of the European Union. (2024). Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence. Official Journal of the European Union.