Trusting Repository-Supplied Configuration

kgai maintainers

kgai.dev · team@kgai.dev · August 2026

ABSTRACT

A configuration file committed to a repository arrives on a developer's machine with git clone, authored by whoever created that repository. When a tool grants such a file authority, its keys become capabilities: a store path is a write capability, a sync destination is a network egress capability, and rules injected into a language model's context are an instruction channel. We describe the trust model kgai, a tool keeping a team's engineering decisions in a local-first knowledge graph, adopted after reproducing an attack class against its own earlier versions, in which a committed store path overwrote another repository's ignore rules and a committed sync destination exfiltrated a developer's data to an author-chosen endpoint and planted fabricated decision history in return. The resulting posture is default deny. A committed configuration decides nothing until a person explicitly approves it, approval binds to the settings the file asks for rather than to its bytes, any change asks again, and refusal can be recorded as a dismissal. We argue that this posture generalizes to agent tooling.

1. Introduction

Developer tools have long read configuration from the project directory. For most of that history the file's author and its consumer were the same person or team, so treating the file as trusted was a reasonable economy. A committed file, however, arrives on the consumer's machine with git clone, written by whoever made the repository. Configuration that travels with a clone is untrusted input, not configuration.

The governing principles are decades old. Fail-safe defaults demand that access rest on explicit permission rather than absence of objection [1], and the impossibility of fully verifying what one runs argues for trusting as little as possible implicitly [2]. What is new is the consumer. Agent tooling reads configuration at session start, acts on it through hooks without per-action review, and feeds parts of it to a language model as text.

kgai, a local-first knowledge graph that records engineering decisions from AI-assisted coding sessions, reads one of its three configuration layers from a committed file at the repository root. This report describes the threat model of that file, the attack class reproduced against versions predating the trust mechanism, the mechanism released in 1.5.0 and extended in 1.5.1, and why we consider default deny the correct posture for repository-supplied configuration in agent tooling generally.

2. Threat Model

The adversary is the author of a repository the victim clones, and the adversary's capability is complete control over its committed bytes, nothing else. No compromise of the victim's machine, network, or accounts is assumed. The victim is anyone who clones the repository and starts an instrumented coding session in it. The goals are the usual three: exfiltrate data, corrupt state outside the repository, or poison what the victim's tools subsequently read.

This is the standard supply chain position: attacks arrive through artifacts developers fetch voluntarily, and the literature documents both observed attacks [3] and a taxonomy of vectors [4]. Ecosystem studies show how much reach one author gains when tooling obeys fetched content by default [5]. A committed configuration file carries no code, making it a lower-tech channel than a malicious package, but the moment a tool grants it authority its keys become capabilities. A key that names a filesystem path is a write capability. A key that names a sync destination is a network egress capability. A key whose value reaches a language model's context is an instruction channel, and injecting instructions through retrieved data is a demonstrated attack class against LLM-integrated applications [6].

In kgai the committed layer is a file named .kgairc at the repository root, one of three layers (session, project, global). Four keys matter here: prompt (capture rules handed to the agent each session), store (where the decision log lives), remote (the sync destination), and cloud_url (the address of a hosted broker). The layer exists for a legitimate reason: a team's conventions and shared store location should arrive with the repository rather than be configured by hand. The wider scoping model those layers serve is described by the scoping report in this series (TR-2026-10).

3. The Reproduced Attack Class

Before version 1.5.0, two of these keys acted on clone-supplied authority alone. Both attacks below were reproduced against that behavior and are documented in the project's changelog.

In the first, a committed store path was obeyed as written. Store initialization then overwrote a different repository's ignore rules and scattered log files through its working tree. One route to this failure required no crafted path at all: a variable reference unset on the victim's machine expanded to the empty string and resolved to the repository's own root, with the same result. A path-valued key had become a write capability into directories the victim never chose.

In the second, a committed sync destination was honored. The background session hooks pushed the developer's recorded decisions to an endpoint the repository's author chose. With the store also pointed at the victim's home directory, the push carried files living there, including SSH keys and environment files. The same synchronization then pulled the author's fabricated team decisions back into the local log, where search returned them as genuine engineering history.

The second attack is the more instructive because it composes exfiltration with poisoning. The decision log is a retrieval corpus for a language model, and an agent that consults it treats recalled decisions as authoritative prior context, so writing into a victim's log is indirect prompt injection with persistence [6]. The corpus a team trusts most, its own recorded history, is exactly the one an adversary most wants to write to.

The response, released in 1.5.0, has two parts: hard limits that hold whether or not anything is approved, and an approval gate for the authority that remains.

4. Hard Limits: What No Committed File May Do

Not every question is delegated to user judgment. Some capabilities are removed from the committed layer outright, applying least privilege to the layer itself [1].

The remote and cloud_url keys are never taken from a committed file. A write to the wrong layer is refused, and a value found in the wrong layer is ignored on read. The read side is the load-bearing half, because a committed file never went through the tool's own write path. Ignored keys are listed in the configuration report, so a committed sync destination is visibly ignored rather than mysteriously ineffective. The rationale is partly architectural, since synchronization belongs to the store rather than to one repository, and partly a security boundary: a cloned file cannot choose where a developer's decisions are uploaded, nor point an install-local credential at a foreign broker.

A store value that would damage something is refused, with the reason. An unresolved variable is an error naming the variable. The repository root, the home directory, and any directory already holding someone else's files are refused. Symlinks are resolved before these checks, so a link cannot aim the checks at one directory while writes land in another. The store's scaffold refuses to overwrite an ignore file it did not write. A broken or conflicted committed file, a routine state for anything under version control, stops the command instead of silently resolving to a freshly minted store nobody reads.

Finally, capture rules reach the model framed as data rather than instructions. They are injected inside a delimiter carrying a per-session random tag, so text inside the rules cannot close the block and pose as instructions, the boundary is restated after the data, and the value is capped at 8,000 bytes, enforced on read because a hand-written file never passes through the writing code.

5. The Approval Gate

What remains, the capture rules and the store location, is real authority that a team legitimately wants to ship with its repositories. For these, the mechanism is explicit approval with a defined pending state.

Until a person approves, a committed configuration decides nothing. Its store value is ignored, its rules are not injected into the session, and the pending state is reported redundantly: by the configuration and

prompt commands, by the session-start hook, and by the installer's own status line in every session. Nothing is blocked meanwhile. Work continues with the repository's settings simply absent. The local per-project store is deliberately not created while the decision is pending, so a later approval of a team store leaves no stray store behind.

Approval is a human act. A show command prints the store path and rules the file asks for and approves nothing. In an agent session the approval is a conversation: the agent displays what the file asks for, waits for the user's answer, and is forbidden from approving on its own initiative. An unapproved file has no channel into the conversation through which to argue for itself.

What is approved is the settings the file asks for, not its bytes. The fingerprint is a hash of the values of the keys the layer may set, canonically ordered. Reformatting the file or reordering keys asks no new question. Changing a rule or the store path asks again, whether the user edited the file or a teammate's commit did. Adding a sync destination changes nothing, because that key is ignored regardless. Because the fingerprint is the configuration, one approval covers every repository asking for exactly the same thing, an organization's standard configuration is accepted once per machine, and an inherited approval is announced once rather than passing silently.

Refusal is recorded as deliberately as consent. A dismissal, added in 1.5.1, is bound to the same fingerprint. The configuration continues to decide nothing, the per-session reminder stops, a later change to the file asks again, and a revocation re-arms the prompt.

Approvals live in a per-machine, per-user file, never synchronized and never in the repository. The exclusion is load-bearing. An approval stored in the committed file would travel to everyone who clones it, pre-approving it for exactly the people the step protects, and a hash stored inside the file would change what the file hashes.

The pattern has precedents. direnv refuses to load a cloned environment file until the user allows it and asks again when the file changes [7], and git refuses to read repository configuration owned by another user unless the path is explicitly listed as safe [8]. kgai's contribution is to bind the approval to the semantic content of the requested settings rather than to a file path or byte hash, which makes reformatting free and organizational reuse a single question.

6. Default Deny as a General Posture for Agent Tooling

Fail-safe defaults, basing access decisions on explicit permission rather than exclusion, is the oldest principle in the catalogue [1], and repository-supplied configuration is an input crossing a trust boundary in the plainest sense. Three properties of agent tooling make the default-allow alternative untenable.

First, agent tooling acts at session start, through hooks, without per-action review, so a configuration key can take effect before any person has read the file. That removes the informal safeguard that made

trusting project files survivable in interactive tools.

Second, the configuration surface of agent tools is unusually capable, routinely naming filesystem locations, network destinations, and credentials, so a default-allow read hands each of those capabilities to whoever authored the repository.

Third, some configuration is destined for the model's context, and any text an untrusted party can place in front of a language model is a potential instruction channel [6]. A committed rules key is an injection surface rather than a preference, to be framed as data with a boundary its content cannot forge.

The attacks of Section 3 are ordinary consequences of ignoring these properties. The resulting posture for any agent tool that reads committed configuration: partition keys by layer and never accept egress destinations from the committed layer, refuse values that can damage state before approval is even considered, make the pending state visible rather than silent, bind approval to semantic content and ask again on change, allow refusal to be recorded so that safety does not nag, and frame committed text as data wherever it reaches the model.

7. Residual Risk

The mechanism narrows trust to one explicit question, and the honest statement is that the question itself remains fallible.

Approved rules are still text a model reads. The framing is strong, but it is not a sandbox. Rules a user approves can shape what gets recorded, because that is their purpose, so the approval is only as good as the reading that preceded it.

One approval covers identical configurations. If an organization's standard committed configuration is public, a third party can copy it into their own repository, and cloning that repository enrolls it without a new question, since it asks for exactly what was already approved. The copy gains nothing beyond what that configuration was granted, but the difference from per-file approval is real. The documented mitigations are keeping organizational store paths out of public repositories or including something repository-specific in the rules.

The human can delegate the gate away. The agent is forbidden from approving on its own initiative, but it will run the approval if the user tells it to. A user who says approve without reading has reduced the mechanism to a formality.

Finally, the first session on a new machine can miss the pending notice when the reporting hook races the engine installation it depends on. Subsequent sessions report it.

8. Conclusion

A committed configuration file is the cheapest supply chain vector there is: no malicious package, no compromised account, no code, only a tool willing to obey what arrived with a clone. The attack class we reproduced against kgai's own earlier versions shows what obedience costs in an agent setting: a write capability into unrelated repositories, exfiltration of a developer's files to an author-chosen endpoint, and fabricated history planted in the corpus a model treats as ground truth. The remedy is neither novel nor complicated, and that is the point. Remove the dangerous capabilities from the committed layer entirely, and gate the legitimate remainder behind an explicit human approval that is bound to what the file asks for, asks again when that changes, and can be declined on the record. We adopted it after demonstrating the attacks against our own tool, and we see no reason it should not be the default wherever configuration arrives with a clone.

References

[1] J. H. Saltzer and M. D. Schroeder. 1975. The Protection of Information in Computer Systems. Proceedings of the IEEE 63(9), 1278-1308.

[2] K. Thompson. 1984. Reflections on Trusting Trust. Communications of the ACM 27(8), 761-763.

[3] M. Ohm, H. Plate, A. Sykosch, and M. Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2020), Springer, 23-43.

[4] P. Ladisa, H. Plate, M. Martinez, and O. Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. 2023 IEEE Symposium on Security and Privacy.

[5] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. 28th USENIX Security Symposium, 995-1010.

[6] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. 2023. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec 2023).

[7] direnv maintainers. 2026. direnv documentation, the allow mechanism. https://direnv.net/

[8] Git project. 2022. git-config documentation, the safe.directory setting, introduced in Git 2.35.2 in response to CVE-2022-24765. https://git-scm.com/docs/git-config