Claw Code (Rust)

audited.xyz
by zack.eth

Date
Apr 03, 2026 01:58 UTC

Commit
1abd951
53 files, 43,786 lines

Summary

Claw Code is a Rust-based interactive CLI agent for conversing with Claude language models, structured as a multi-crate workspace spanning approximately 22,000 lines of code across eight crates. The architecture follows a layered design: the `api` crate handles authenticated communication with LLM providers (Anthropic, xAI/Grok, OpenAI) via HTTP and SSE streaming; the `runtime` crate implements session management, sandboxed command execution, OAuth with PKCE, and a layered permission system; the `tools` crate provides a dispatcher over roughly 40 built-in tools for file operations, shell execution, web fetching, and MCP integration; the `plugins` crate manages external extensibility through hook runners and custom tool executors; and the `commands` crate parses 67 slash commands with typed enum dispatch. Supporting crates include `telemetry` for local JSONL tracing, `compat-harness` for upstream TypeScript manifest extraction, and `rusty-claude-cli` for the REPL frontend with rich terminal rendering.

The codebase demonstrates deliberate security engineering across multiple layers. Command execution consistently uses `std::process::Command::new().args()` rather than shell string interpolation, eliminating command injection as a vulnerability class in the core execution paths. The OAuth implementation enforces PKCE with S256 challenges and validates the `state` parameter against CSRF, binding its callback listener to localhost only. File path handling employs canonicalization before operations, with dedicated sanitization functions stripping directory traversal characters from plugin names and skill invocations. The permission system assigns per-tool access modes (ReadOnly, WorkspaceWrite, DangerFullAccess) and restricts sub-agent tool subsets, while the sandbox module provides Linux namespace isolation via `unshare` with container-environment detection. Atomic file writes use a temp-file-then-rename pattern, the prompt cache sanitizes path segments against traversal, and URL fetching upgrades HTTP to HTTPS for non-localhost targets.

The audit identified two findings across the entire workspace — one at low-severity and one at low-severity — reflecting a strong overall security posture for a developer tool of this complexity. Six of the eight crates produced zero findings, with the telemetry, compat-harness, CLI, api, and commands crates all demonstrating clean security profiles. The complete absence of `unsafe` code across all eight crates eliminates an entire class of memory safety concerns, and the consistent use of `Result`-based error propagation prevents silent failures throughout the workspace.

Findings

2 issues identified

1 Medium | 1 Low

M-1 | MEDIUM | Open

Project-level config can inject hook commands that bypass permission checks

`rust/crates/runtime/src/config.rs:227`

Description

The configuration system deep-merges hooks from all config sources (user, project, local). Project-level configs in cloned repositories can define PreToolUse hooks that execute arbitrary shell commands before any permission check in the conversation runtime's run_turn method. A malicious repository containing .claw/settings.json with hook definitions will execute those hooks on every tool invocation regardless of the user's permission mode.

Impact

Arbitrary command execution on the user's machine when running the CLI in a malicious repository. Exfiltration of environment variables, API keys, and tool inputs/outputs. No user confirmation required even in ReadOnly mode.

Recommendation

Only allow hooks from user-level config sources. Parse hooks per-source before merging and exclude project/local sources from hook configuration.

Fix

Restricts hook loading to user-level config entries only, preventing project-level or local-level configs from injecting shell commands that execute before permission checks.

`rust/crates/runtime/src/config.rs`

```
-         let feature_config = RuntimeFeatureConfig {
-             hooks: parse_optional_hooks_config(&merged_value)?,
-             plugins: parse_optional_plugin_config(&merged_value)?,
+         let user_hooks_value = loaded_entries
+             .iter()
+             .filter(|entry| entry.source == ConfigSource::User)
+             .filter_map(|entry| read_optional_json_object(&entry.path).ok().flatten())
+             .filter_map(|obj| obj.get("hooks").cloned())
+             .last();
+         let hooks = if let Some(hooks_val) = user_hooks_value {
+             let mut wrapper = BTreeMap::new();
+             wrapper.insert("hooks".to_string(), hooks_val);
+             parse_optional_hooks_config(&JsonValue::Object(wrapper))?
+         } else {
+             RuntimeHookConfig::default()
```

```
+        };
+        let feature_config = RuntimeFeatureConfig {
+            hooks,
+            plugins: parse_optional_plugin_config(&merged_value)?,
```

L-1 LOW Open

Pipe buffer deadlock in hook execution

`</> rust/crates/plugins/src/hooks.rs`

Description

The output_with_stdin method writes the entire JSON payload to the child process stdin synchronously via write_all, then calls wait_with_output to read stdout/stderr. When stdin, stdout, and stderr are all piped, this creates a classic pipe buffer deadlock: the parent blocks on write_all if the OS pipe buffer fills (~64KB), while the child blocks writing to stdout/stderr because the parent has not started reading those pipes. The hook payload includes tool_input and tool_output which can easily exceed pipe buffer limits.

Impact

The hook runner is invoked on every tool use. A deadlock hangs the entire CLI agent session with no automatic recovery. The user must manually kill the process, losing in-progress conversation state. Reliably triggerable when hooks produce output and the combined payload exceeds the pipe buffer size.

Recommendation

Write stdin in a background thread so the parent can proceed to read stdout/stderr concurrently via wait_with_output.

Fix

Moves the stdin write to a background thread, allowing wait_with_output to read stdout/stderr concurrently. The child_stdin handle is moved into the thread and dropped after writing, closing the pipe so the child sees EOF.

`rust/crates/plugins/src/hooks.rs`

```
-    fn output_with_stdin(&mut self, stdin: &[u8])→ std::io::Result<std::process::Output> {
-        let mut child = self.command.spawn()?;
-        if let Some(mut child_stdin) = child.stdin.take() {
-            use std::io::Write as _;
-            child_stdin.write_all(stdin)?;
-        }
-        child.wait_with_output()
-    }
+    fn output_with_stdin(&mut self, stdin_data: &[u8])→ std::io::Result<std::process::Output> {
+        let mut child = self.command.spawn()?;
+        if let Some(child_stdin) = child.stdin.take() {
+            let stdin_data = stdin_data.to_vec();
+            std::thread::spawn(move || {
+                use std::io::Write as _;
```

```
+                 let mut stdin = child_stdin;
+                 let _ = stdin.write_all(&stdin_data);
+             });
+         }
+         child.wait_with_output()
+     }
```

Conclusion

The Claw Code workspace exhibits a high level of security maturity for a Rust CLI application. The codebase leverages Rust's type system effectively — typed enums for message roles and content blocks prevent type confusion, and the complete absence of `unsafe` code across all crates is notable for a project of this size. Test coverage is thorough, with unit tests spanning argument parsing, session lifecycle, SSE edge cases, cache expiry, and plugin manifest validation. The trait-based provider abstraction in the API crate, the layered configuration system in the runtime, and the dispatcher pattern in the tools crate all reflect careful architectural separation of concerns.

The low-severity finding [H-1] represents the most pressing security concern, as project-level configuration files can inject hook commands that bypass the permission checks enforced on standard tool execution paths. Since hooks execute with the full privileges of the host process, a malicious or compromised project configuration could escalate access beyond what the permission model intends to allow. The low-severity pipe buffer deadlock [M-1] in the hook execution path affects the critical execution path of every tool invocation when hooks are enabled — a single hook producing output larger than the OS pipe buffer (typically 64KB) will render the CLI session unresponsive. Both findings are localized to the `plugins` crate and have straightforward, non-breaking fixes.

The workspace is suitable for deployment after addressing the two identified findings. The hook command injection bypass [H-1] should be resolved before any deployment where the permission model is relied upon as a security boundary, by validating hook configurations against the same permission checks applied to tool execution. The pipe deadlock fix [M-1] requires minimal code changes — replacing synchronous stdin writes with threaded writes — and should be applied before enabling the plugin hook system in production. Beyond these remediations, no architectural changes are required; the codebase's security foundations — its permission model, credential handling, input validation, and execution isolation — are well-engineered and appropriate for a local developer tool operating in single-user environments.