ClaURST

audited.xyz
by zack.eth

Date
Apr 03, 2026 00:35 UTC

Commit
d024edc
139 files, 77,236 lines

Summary

ClaURST is a Rust-based AI assistant platform structured as a modular workspace of ten crates spanning a CLI interface, TUI frontend, API layer, MCP (Model Context Protocol) server, bridge and buddy coordination layers, a query engine, command dispatcher, tooling subsystem, and a shared core library. The architecture follows a clean separation of concerns: the `core` crate provides shared types and utilities consumed by higher-level crates, while `commands` and `tools` encapsulate discrete operations that the `cli`, `tui`, and `api` crates orchestrate into user-facing workflows. The `mcp` crate exposes the assistant's capabilities as a standardized protocol server, and the `bridge` crate mediates between internal representations and external AI provider APIs. This multi-crate design enables independent compilation and testing of each subsystem, which is a sound architectural choice for a project of this scope.

The codebase demonstrates several positive security patterns consistent with Rust's ownership model and type system. Memory safety is largely enforced at compile time, and the use of strong typing across crate boundaries reduces the surface area for type-confusion vulnerabilities. The API and bridge layers employ structured request building rather than raw string concatenation for most external communications, and the `query` crate appears to parameterize its inputs in the common case. The project also implements a tool restriction mechanism via `--allowed-tools` and `--disallowed-tools` CLI flags, signaling an intent to constrain the assistant's execution capabilities — though as noted in finding [M-1], the enforcement of these restrictions is incomplete.

This audit identified one low-severity finding relating to the tool restriction subsystem. While the core Rust architecture provides a strong compile-time safety foundation and the codebase is free of high or low-severity vulnerabilities, the medium finding reveals a gap between the project's stated security policy and its runtime enforcement: tool restriction flags are accepted and parsed at the CLI layer but never propagated to the execution path where tool invocations are dispatched. The absence of critical memory safety or injection vulnerabilities reflects well on the development team's use of Rust idioms, and the overall attack surface is narrow relative to the project's functionality.

Findings

1 issues identified

1 Low

L-1 LOW Open

Tool restriction flags (--allowed-tools, --disallowed-tools) parsed but never applied

</> src-rust/crates/cli/src/main.rs

Description

The CLI defines –allowed-tools and –disallowed-tools flags that accept comma-separated tool names to restrict which tools the AI agent can invoke. However, these parsed values are never assigned to the Config, ToolContext, QueryConfig, or any other runtime structure. The flags are silently ignored, meaning a user who passes –disallowed-tools Bash believes they have restricted Bash execution when no restriction is enforced.

Impact

Users in CI/CD or automated environments may believe tool execution is restricted when it is not. Security policies relying on CLI-level tool restriction are ineffective. –disallowed-tools Bash would fail to prevent arbitrary command execution.

Recommendation

Wire cli.allowed_tools and cli.disallowed_tools into the Config struct after building config, and enforce filtering in the tool execution layer.

Fix

Parses the comma-separated tool names from the CLI flags and assigns them to the Config struct so downstream tool execution logic can enforce the restrictions. The Config struct's allowed_tools/disallowed_tools fields must also exist (may require addition in cc_core).

src-rust/crates/cli/src/main.rs

```
-     config.additional_dirs = cli.add_dir.clone();
-     if cli.no_auto_compact {
-         config.auto_compact = false;
-     }
+     config.additional_dirs = cli.add_dir.clone();
+     if let Some(ref allowed) = cli.allowed_tools {
+         config.allowed_tools = Some(
+             allowed.split(',').map(|s| s.trim().to_string()).collect()
+         );
+     }
+     if let Some(ref disallowed) = cli.disallowed_tools {
+         config.disallowed_tools = Some(
+             disallowed.split(',').map(|s| s.trim().to_string()).collect()
```

```
+         );
+     }
+     if cli.no_auto_compact {
+         config.auto_compact = false;
+     }
```

Conclusion

The overall code quality of ClaURST reflects a team with solid Rust proficiency and good architectural instincts. The modular crate structure, use of Rust's type system for inter-component contracts, and clean separation between the command layer and execution layer all indicate a mature development approach. The codebase's reliance on Rust's ownership and borrowing semantics eliminates entire classes of memory safety vulnerabilities that would be present in comparable C or C++ implementations, and the structured approach to external API communication avoids the common pitfalls of string-based request construction.

The single low-severity finding identified in this audit — the unenforced tool restriction flags — represents a meaningful gap in the platform's access control posture. If ClaURST is deployed in contexts where constraining the assistant's tool usage is a security requirement, the current implementation provides a false sense of restriction: operators who configure `--allowed-tools` or `--disallowed-tools` would reasonably expect those flags to limit execution, but the assistant would proceed to invoke any tool regardless. This is particularly relevant in multi-tenant or sandboxed deployment scenarios where limiting tool access is a key defense-in-depth measure. The finding does not indicate a broader pattern of security neglect, but rather an incomplete implementation path where the CLI parsing layer advanced ahead of the enforcement layer.

ClaURST's underlying architecture is well-positioned for secure deployment once the tool restriction enforcement gap is closed. The recommended remediation is to propagate the parsed allow and deny lists from the CLI configuration through to the tool dispatch layer, ensuring that any tool invocation is validated against the configured policy before execution proceeds. With this targeted fix applied, the platform's security posture would be consistent with its architectural quality, and no structural redesign would be required. Continued attention should be paid to the boundaries where the assistant interfaces with the host operating system and external services, as these integration points remain the areas where Rust's compile-time guarantees must be supplemented by application-level security logic.