Nano Claude Code

audited.xyz
by zack.eth

Date
Apr 03, 2026 04:44 UTC

Commit
3c81ae0
40 files, 12,563 lines

Summary

Nano Claude Code is a multi-provider AI coding assistant CLI implemented as a flat-module Python layout with a central tool registry, threaded sub-agent execution via `ThreadPoolExecutor`, and streaming adapters for multiple LLM providers. The application supports tool execution including file I/O, shell commands, and web fetching, alongside persistent memory, sub-agent spawning with git worktree isolation, and a reusable skill system. The architecture maintains clear module boundaries and unidirectional dependencies, with the tool registry serving as the primary dispatch and permission-checking boundary for all external operations.

The codebase implements several intentional security controls that reflect mature design thinking. A three-tier permission system (`auto`/`accept-all`/`manual`) gates destructive tool calls, output truncation prevents context window overflow attacks, and depth-limited sub-agent recursion guards against infinite loops. The memory system enforces scope isolation between user and project levels through frontmatter-based markdown files, and cooperative cancellation provides clean shutdown semantics for background tasks. These patterns demonstrate a security-conscious development approach that goes beyond minimal functionality.

This audit identified two findings: one medium-severity SSRF vulnerability in the WebFetch tool's unrestricted URL handling, and one low-severity shell metacharacter bypass in the safe command allowlist. The SSRF issue is the more pressing concern, as it could allow an attacker — particularly through prompt injection via untrusted input such as third-party CLAUDE.md files or web content — to reach internal network services or cloud metadata endpoints. The shell metacharacter bypass represents a defense-in-depth failure where the permission model can be circumvented by crafting commands that evade the allowlist matching logic. The overall security posture is reasonable for a local developer tool, though both findings should be addressed before use in environments where the LLM processes untrusted input.

Findings

2 issues identified

1 Medium | 1 Low

M-1 | MEDIUM | Open

SSRF via unrestricted URL fetch in WebFetch tool

`</> tools.py:_webfetch (line 175)`

Description

The `_webfetch` function passes user-supplied URLs directly to `httpx.get()` without validating the scheme, hostname, or resolved IP address. An attacker (or a malicious AI prompt) can supply URLs targeting internal services (e.g., `http://169.254.169.254/latest/meta-data/`, `http://localhost:6379/`, `file:///etc/passwd`, `http://10.0.0.1/admin`) to reach resources that should not be accessible from the server.

Impact

An attacker can scan internal networks, access cloud metadata endpoints (AWS/GCP/Azure instance credentials), interact with internal services (Redis, databases, admin panels), and potentially exfiltrate sensitive data. On cloud infrastructure, SSRF against the metadata endpoint is a well-known path to full account compromise.

Recommendation

Validate the URL scheme (allow only http/https), resolve the hostname to an IP address before making the request, and reject requests to private/reserved IP ranges (RFC 1918, link-local, loopback). Use the resolved IP directly for the connection to prevent DNS rebinding.

Fix

```
- def _webfetch(url: str, prompt: str = None)→ str:
-     try:
-         import httpx
-         r = httpx.get(url, headers={"User-Agent": "NanoClaude/1.0"},
-                       timeout=30, follow_redirects=True)
+ def _webfetch(url: str, prompt: str = None)→ str:
+     try:
+         import httpx
+         import socket
+         import ipaddress
+         from urllib.parse import urlparse
+
+         parsed = urlparse(url)
+         if parsed.scheme not in ("http", "https"):
+             return f"Error: unsupported URL scheme '{parsed.scheme}' (only http/https allowed)"
+         if not parsed.hostname:
```

```
+            return "Error: no hostname in URL"
+
+        # Resolve hostname and block private/reserved IPs to prevent SSRF
+        try:
+            addrinfo = socket.getaddrinfo(parsed.hostname, parsed.port or (443 if parsed.scheme== "https" else 80))
+            for family, _type, _proto, _canonname, sockaddr in addrinfo:
+                ip = ipaddress.ip_address(sockaddr[0])
+                if ip.is_private or ip.is_loopback or ip.is_reserved or ip.is_link_local:
+                    return f"Error: URL resolves to non-public IP address ({ip})"
+        except socket.gaierror:
+            return f"Error: could not resolve hostname '{parsed.hostname}'"
+
+        r = httpx.get(url, headers={"User-Agent": "NanoClaude/1.0"},
+                      timeout=30, follow_redirects=True)
```

L-1 LOW Open

Shell metacharacter bypass in safe command allowlist

</> tools.py:166

Description

The _is_safe_bash() function uses prefix matching only, allowing commands like ‘ls; rm -rf /’ or ‘python -c “malicious code”’ to be auto-approved without user confirmation.

Impact

If the LLM is manipulated via prompt injection, destructive or data-exfiltrating shell commands can bypass the permission system by prefixing with a safe command.

Recommendation

Add shell metacharacter detection to reject commands that chain operations after a safe prefix.

Fix

Before checking safe prefixes, reject any command containing shell metacharacters that could chain additional operations. This prevents 'ls; rm -rf /' from being auto-approved.

tools.py

```
- def _is_safe_bash(cmd: str)→ bool:
-     c = cmd.strip()
-     return any(c.startswith(p) for p in _SAFE_PREFIXES)
+ def _is_safe_bash(cmd: str)→ bool:
+     c = cmd.strip()
+     # Reject commands with shell metacharacters that could chain dangerous operations
+     shell_meta = (';', '&&', '||', '|', '`', '$(', '\n', '<(', '>(', '\r')
+     if any(meta in c for meta in shell_meta):
+         return False
```

```
+     return any(c.startswith(p) for p in _SAFE_PREFIXES)
```

Conclusion

Nano Claude Code demonstrates solid software engineering practices for a local developer tool. The modular architecture with a central tool registry, clean module boundaries, and comprehensive test coverage across six test files indicates mature development. The permission system, output truncation, and depth limiting show intentional security design, and the codebase avoids common Python web vulnerabilities like SQL injection and unsafe deserialization entirely. The streaming provider abstraction and worktree-based agent isolation reflect thoughtful architectural decisions that balance functionality with operational safety.

The two identified vulnerabilities — the unrestricted URL fetching in the WebFetch tool and the shell metacharacter bypass in command allowlisting — both represent gaps in input validation at trust boundaries rather than fundamental architectural weaknesses. In a prompt injection scenario, where a malicious instruction is embedded in content the LLM processes, these gaps could be chained to achieve server-side request forgery or unauthorized command execution. Addressing these issues requires targeted fixes to URL validation and command parsing logic, neither of which demands architectural changes or introduces regression risk.

The codebase is suitable for local single-user development use in its current state. For deployment in shared environments, CI pipelines, or any context where the LLM processes untrusted input — including web content, third-party repository files, or user-supplied prompts referencing external resources — both findings should be remediated. Implementing a URL allowlist or blocking private address ranges in the WebFetch tool, and switching to parsed-argument matching rather than string-based prefix matching in the command allowlist, would close the identified gaps with minimal code changes.