Straiker

Claude Code Source Leak: With Great Agency Comes Great Responsibility

Appendix: Referenced Source Code

All references point to the Claude Code TypeScript source (`claude/src/`).

Ref 1: `query.ts:307-310`

Main query loop — the while(true) that drives all four compaction stages

```
  while (true) {
    // Destructure state at the top of each iteration. toolUseContext alone
    // is reassigned within an iteration (queryTracking, messages updates);
    // the rest are read-only between continue sites.
```

Ref 2: `query.ts:379-394`

Stage 1: Tool result budgeting — applies per-message byte limits

```
    messagesForQuery = await applyToolResultBudget(
      messagesForQuery,
      toolUseContext.contentReplacementState,
      persistReplacements
        ? records =>
            void recordContentReplacement(
              records,
              toolUseContext.agentId,
            ).catch(logError)
        : undefined,
      new Set(
        toolUseContext.options.tools
          .filter(t => !Number.isFinite(t.maxResultSizeChars))
          .map(t => t.name),
      ),
    )
```

Ref 3: `query.ts:413-419`

Stage 2: Microcompact — clears old tool results

```
    queryCheckpoint('query_microcompact_start')
    const microcompactResult = await deps.microcompact(
      messagesForQuery,
      toolUseContext,
      querySource,
```

```
    )
    messagesForQuery = microcompactResult.messages
```

Ref 4: `query.ts:440-447`

Stage 3: Context collapse — archives message segments

```
    if (feature('CONTEXT_COLLAPSE') && contextCollapse) {
      const collapseResult = await contextCollapse.applyCollapsesIfNeeded(
        messagesForQuery,
        toolUseContext,
        querySource,
      )
      messagesForQuery = collapseResult.messages
    }
```

Ref 5: `query.ts:453-468`

Stage 4: Autocompact — full conversation summarization

```
    queryCheckpoint('query_autocompact_start')
    const { compactionResult, consecutiveFailures } = await deps.autocompact(
      messagesForQuery,
      toolUseContext,
      {
        systemPrompt,
        userContext,
        systemContext,
        toolUseContext,
        forkContextMessages: messagesForQuery,
      },
      querySource,
      tracking,
      snipTokensFreed,
    )
    queryCheckpoint('query_autocompact_end')
```

Ref 6: `services/compact/microCompact.ts:41-51`

COMPACTABLE_TOOLS — MCP tools are absent from this set

```
const COMPACTABLE_TOOLS = new Set<string>([
  FILE_READ_TOOL_NAME,
  ...SHELL_TOOL_NAMES,
  GREP_TOOL_NAME,
  GLOB_TOOL_NAME,
  WEB_SEARCH_TOOL_NAME,
  WEB_FETCH_TOOL_NAME,
  FILE_EDIT_TOOL_NAME,
  FILE_WRITE_TOOL_NAME,
```

```
])
```

Ref 7: utils/toolResultStorage.ts:816-818

Read tool results exempted from budgeting (maxResultSizeChars: Infinity)

```
    // Tools with maxResultSizeChars: Infinity (Read) — never persist.
    // Mark as seen (frozen) so the decision sticks across turns. They don't
    // count toward freshSize; if that lets the group slip under budget and
```

Ref 8: utils/toolResultStorage.ts:375-377,391-392

seenIds — once processed, a result's fate is frozen for the session

```
 *   - seenIds: results that have passed through the budget check (replaced
 *     or not). Once seen, a result's fate is frozen for the conversation.
 *   - replacements: subset of seenIds that were persisted to disk and
 *     replaced with previews, mapped to the exact preview string shown to
 *     the model. Re-application is a Map lookup — no file I/O, guaranteed
 *     byte-identical, cannot fail.
 *
 * Lifecycle: one instance per conversation thread, carried on
ToolUseContext.
 * Main thread: REPL provisions once, never resets — stale entries after
 * /clear, rewind, resume, or compact are never looked up (tool_use_ids are
 * UUIDs) so they're harmless. Subagents: createSubagentContext clones the
 * parent's state by default (cache-sharing forks like agentSummary need
 * identical decisions), or resumeAgentBackground threads one reconstructed
 * from sidechain records.
 */
export type ContentReplacementState = {
  seenIds: Set<string>
  replacements: Map<string, string>
```

Ref 9: services/compact/prompt.ts:43

Compaction prompt: 'pay special attention to specific user feedback'

```
   - Pay special attention to specific user feedback that you received,
especially if the user told you to do something differently.
2. Double-check for technical accuracy and completeness, addressing each
required element thoroughly.`
```

Ref 10: services/compact/prompt.ts:69-73

Compaction prompt: preserve all user messages, note user feedback

```
2. Key Technical Concepts: List all important technical concepts,
technologies, and frameworks discussed.
3. Files and Code Sections: Enumerate specific files and code sections
examined, modified, or created. Pay special attention to the most recent
messages and include full code snippets where applicable and include a
summary of why this file read or edit is important.
4. Errors and fixes: List all errors that you ran into, and how you fixed
them. Pay special attention to specific user feedback that you received,
especially if the user told you to do something differently.
5. Problem Solving: Document problems solved and any ongoing troubleshooting
efforts.
6. All user messages: List ALL user messages that are not tool results. These
are critical for understanding the users' feedback and changing intent.
```

Ref 11: services/compact/prompt.ts:358-362

Post-compaction: 'continue without asking the user any further questions'

```
    let continuation = `${baseSummary}
Continue the conversation from where it left off without asking the user any
further questions. Resume directly — do not acknowledge the summary, do not
recap what was happening, do not preface with "I'll continue" or similar.
Pick up the last task as if the break never happened.`

    if (
      (feature('PROACTIVE') || feature('KAIROS')) &&
```

Ref 12: tools/BashTool/BashTool.tsx:242

dangerouslyDisableSandbox parameter definition

```
  dangerouslyDisableSandbox:
semanticBoolean(z.boolean().optional()).describe('Set this to true to
dangerously override sandbox mode and run commands without sandboxing.'),
  _simulatedSedEdit: z.object({
    filePath: z.string(),
```

Ref 13: services/compact/microCompact.ts:461-463

keepRecent threshold — controls how many tool results survive

```
  const keepRecent = Math.max(1, config.keepRecent)
  const keepSet = new Set(compactableIds.slice(-keepRecent))
  const clearSet = new Set(compactableIds.filter(id => !keepSet.has(id)))
```

Ref 14: tools/BashTool/bashSecurity.ts:2308-2313

Early validators — can short-circuit entire chain

```
  const earlyValidators = [
    validateEmpty,
    validateIncompleteCommands,
    validateSafeCommandSubstitution,
    validateGitCommit,
  ]
```

Ref 15: `tools/BashTool/bashSecurity.ts:2348-2378`

Main validators — 19 checks in sequence

```
  const validators = [
    validateJqCommand,
    validateObfuscatedFlags,
    validateShellMetacharacters,
    validateDangerousVariables,
    // Run comment-quote-desync BEFORE validateNewlines: it detects cases
where
    // the quote tracker would miss newlines due to # comment desync.
    validateCommentQuoteDesync,
    // Run quoted-newline BEFORE validateNewlines: it detects the INVERSE
case
    // (newlines INSIDE quotes, which validateNewlines ignores by design).
Quoted
    // newlines let attackers split commands across lines so that line-based
    // processing (stripCommentLines) drops sensitive content.
    validateQuotedNewline,
    // CR check runs BEFORE validateNewlines — CR is a MISPARSING concern
    // (shell-quote/bash tokenization differential), LF is not.
    validateCarriageReturn,
    validateNewlines,
    validateIFSInjection,
    validateProcEnvironAccess,
    validateDangerousPatterns,
    validateRedirections,
    validateBackslashEscapedWhitespace,
    validateBackslashEscapedOperators,
    validateUnicodeWhitespace,
    validateMidWordHash,
    validateBraceExpansion,
    validateZshDangerousCommands,
    // Run malformed token check last - other validators should catch
specific patterns first
    // (e.g., $() substitution, backticks, etc.) since they have more precise
error messages
    validateMalformedTokenInjection,
  ]
```

Straiker

Ref 16: tools/BashTool/bashSecurity.ts:289-295

EARLY-ALLOW comment: bypasses ALL subsequent validators

```
 * Checks if a command is a "safe" heredoc-in-substitution pattern that can
 * bypass the generic $() validator.
 *
 * This is an EARLY-ALLOW path: returning `true` causes bashCommandIsSafe to
 * return `passthrough`, bypassing ALL subsequent validators. Given this
 * authority, the check must be PROVABLY safe, not "probably safe".
 *
```

Ref 17: tools/BashTool/bashSecurity.ts:671-678

Documented attack: git commit → bashrc overwrite → RCE

```
    // `allow` here short-circuits bashCommandIsSafe and SKIPS
    // validateRedirections. So we must bail to passthrough on unquoted `<>`
    // to let the main validators handle it.
    //
    // Attack: `git commit --allow-empty -m 'payload' > ~/.bashrc`
    //   validateGitCommit returns allow → bashCommandIsSafe short-circuits
→
    //   validateRedirections NEVER runs → ~/.bashrc overwritten with git
    //   stdout containing `payload` → RCE on next shell login.
```

Ref 18: tools/BashTool/bashSecurity.ts:2431-2432

Three parsers: splitCommand, shell-quote, tree-sitter

```
  const parsed = await ParsedCommand.parse(command)
  const tsAnalysis = parsed?.getTreeSitterAnalysis() ?? null
```

Ref 19: tools/BashTool/bashSecurity.ts:2340-2342

Known differential: CR as word separator in JS but not bash

```
  // shell-quote's `[^\s]` treats CR as a word separator (JS `\s` ⊃ \r), but
  // bash IFS does NOT include CR. splitCommand collapses CR→space, which IS
  // misparsing. See validateCarriageReturn for the full attack trace.
```

Ref 20: tools/BashTool/bashSecurity.ts:2380-2407

Non-misparsing results deferred and potentially discarded

```
  // SECURITY: We must NOT short-circuit when a non-misparsing validator
  // returns 'ask' if there are still misparsing validators later in the
list.
  // Non-misparsing ask results are discarded at
bashPermissions.ts:~1301-1303
```

```
  // (the gate only blocks when isBashSecurityCheckForMisparsing is set). If
  // validateRedirections (index 10, non-misparsing) fires first on `>`, it
  // returns ask-without-flag — but validateBackslashEscapedOperators (index
12,
  // misparsing) would have caught `\;` WITH the flag. Short-circuiting lets
a
  // payload like `cat safe.txt \; echo /etc/passwd > ./out` slip through.
  //
  // Fix: defer non-misparsing ask results. Continue running validators; if
any
  // misparsing validator fires, return THAT (with the flag). Only if we
reach
  // the end without a misparsing ask, return the deferred non-misparsing
ask.
  let deferredNonMisparsingResult: PermissionResult | null = null
  for (const validator of validators) {
    const result = validator(context)
    if (result.behavior === 'ask') {
      if (nonMisparsingValidators.has(validator)) {
        if (deferredNonMisparsingResult === null) {
          deferredNonMisparsingResult = result
        }
        continue
      }
      return { ...result, isBashSecurityCheckForMisparsing: true as const }
    }
  }
  if (deferredNonMisparsingResult !== null) {
    return deferredNonMisparsingResult
  }
```

Ref 21: `tools/BashTool/bashSecurity.ts:1149-1162`

ANSI-C quoting blocked by validateObfuscatedFlags

```
  // 1. Block ANSI-C quoting ($'...') - can encode any character via escape
sequences
  // Simple pattern that matches $'...' anywhere. This correctly handles:
  // - grep '$' file => no match ($ is regex anchor inside quotes, no $'...'
structure)
  // - 'test'$'-exec' => match (quote concatenation with ANSI-C)
  // - Zero-width space and other invisible chars => match
  // The pattern requires $' followed by content (can be empty) followed by
closing '
  if (/\$'[^']*'/.test(originalCommand)) {
    logEvent('tengu_bash_security_check_triggered', {
      checkId: BASH_SECURITY_CHECK_IDS.OBFUSCATED_FLAGS,
      subId: 5,
    })
```

```
    return {
      behavior: 'ask',
      message: 'Command contains ANSI-C quoting which can hide characters',
```

Ref 22: `tools/BashTool/bashSecurity.ts:193-198`

Warning: extractQuotedContent does not handle ANSI-C quoting

```
 *
 * IMPORTANT: This function only handles single characters, not strings. If
you need to extend
 * this to handle multi-character strings, be EXTREMELY CAREFUL about shell
ANSI-C quoting
 * (e.g., $'\n', $'\x41', $'\u0041') which can encode arbitrary characters
and strings in ways
 * that are very difficult to parse correctly. Incorrect handling could
introduce security
 * vulnerabilities by allowing attackers to bypass security checks.
```