Claude Code Guardrails

Five safety checks for Claude Code, explained

by Gabrielė · Codeart (codeart.lt)

The problem

An agent that can run arbitrary shell commands will, sooner or later, try to run one you did not mean to approve. Guardrails catch that before it executes — not after.

What’s inside

- A .claude/settings.json wiring up three hook registrations
- A guard.sh script implementing five distinct safety checks
- This guide
- Drop-in setup — one dependency (jq), no build step
- An honest list of what it does NOT protect against

How it’s wired

Claude Code hooks are shell commands the CLI runs before or after a tool call, fed a JSON payload on stdin. guard.sh reads that payload once and runs five checks against it. A PreToolUse hook can block the call outright (exit code 2); a PostToolUse hook can only observe, so it’s used here for the audit log.

`.claude/settings.json`

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          { "type": "command", "command": "$CLAUDE_PROJECT_DIR/.claude/guard.sh" }
        ]
      },
      {
        "matcher": "Write|Edit|MultiEdit",
        "hooks": [
          { "type": "command", "command": "$CLAUDE_PROJECT_DIR/.claude/guard.sh" }
        ]
      }
    ],
    "PostToolUse": [
      {
        "matcher": "Bash|Write|Edit|MultiEdit",
        "hooks": [
          { "type": "command", "command": "$CLAUDE_PROJECT_DIR/.claude/guard.sh" }
        ]
      }
    ]
  }
}
```

Three hook registrations, five checks — guard.sh looks at hook_event_name and tool_name to decide which checks apply to this call.

1 · Destructive filesystem commands

Blocks rm -rf aimed at /, ~, or the current directory, raw disk writes via dd, and fork-bomb patterns. These are the commands where there is no undo.

2 · Force-push and history rewrites on protected branches

Blocks git push --force (or -f) specifically when the target is main or master, plus filter-branch and force-pushing --all. Force-pushing a feature branch you own is still allowed — the check only fires on shared history.

3 · Skipping safety checks

Blocks --no-verify and --no-gpg-sign (both exist to skip a check someone deliberately set up), chmod 777, and piping a remote script straight into a shell (curl ... | bash, wget ... | sh) — read a script before it runs, always.

4 · Writes to sensitive files

Blocks Write/Edit/MultiEdit targeting .env, SSH keys, .aws/credentials, credentials.json, or .git/config. If a credential needs to change, do it by hand — an agent shouldn’t be the one touching it.

5 · Audit log

Every tool call gets one timestamped line in .claude/guard.log, labelled RAN or BLOCKED. The catch: a PreToolUse hook that exits 2 stops the tool, so PostToolUse never fires — a naive setup logs only what succeeded and silently loses every blocked command, the very set you want a record of. That is why block() writes its own log line before exiting.

The full guard.sh script

`.claude/guard.sh`

```
#!/usr/bin/env bash
# Claude Code Guardrails — guard.sh
# Reads a PreToolUse/PostToolUse hook payload from stdin and decides
# whether to allow, block, or just log the tool call.
#
# Exit codes (per Claude Code's hook contract):
#   0 = allow (stdout/stderr shown to the user only)
#   2 = block (stderr is fed back to Claude as the reason)
#
# This is a guardrail against mistakes, not a security boundary.
# See the "What this does not protect against" section of the guide.

# Deliberately no -e: a failed log write must never take the hook down,
# because a hook that crashes is a hook that stops protecting you.
set -uo pipefail

payload="$(cat)"

# One jq call instead of four, NUL-separated. Not @tsv + IFS=$'\t':
# bash treats tab as IFS *whitespace*, so an empty field (e.g. no
# .command on a Write call) collapses and every later field shifts left.
# NUL can't appear inside a value, and `read -d ''` keeps empty fields
# and multi-line commands intact.
{
  read -r -d '' hook_event
  read -r -d '' tool_name
  read -r -d '' command
  read -r -d '' file_path
} < <(printf '%s' "$payload" | jq -j '
  (.hook_event_name // ""), "\u0000",
  (.tool_name // ""), "\u0000",
  (.tool_input.command // ""), "\u0000",
  (.tool_input.file_path // ""), "\u0000"
')

log_dir="$(dirname "$0")"
log_file="$log_dir/guard.log"

# $1 = outcome label (RAN / BLOCKED)
write_audit_log() {
  printf '%s\t%s\t%s\t%s\n' \
    "$(date -Iseconds)" "$1" "$tool_name" "${command:-$file_path}" \
    >>"$log_file" 2>/dev/null || true
}

# Log the block BEFORE exiting. PreToolUse exit 2 stops the tool, so
# PostToolUse never fires — without this line the blocked commands, the
# ones you actually want a record of, would never reach the log.
block() {
  write_audit_log "BLOCKED"
  echo "Guardrails blocked this: $1" >&2
  exit 2
}
```

```
# --- 1: destructive filesystem commands ---------------------------------
# Flag-order independent: -rf, -fr, -r -f, --recursive --force all count.
check_destructive_fs() {
  local c=" $command "

  if printf '%s' "$c" | grep -qE '(^|[;&|(]|[[:space:]])rm([[:space:]]|$)'; then
    local rec=0 force=0
    printf '%s' "$c" | grep -qE '([[:space:]]-[a-zA-Z]*[rR][a-zA-Z]*([[:space:]]|$)|[[:space:]]--
recursive([[:space:]]|$))' && rec=1
    printf '%s' "$c" | grep -qE '([[:space:]]-[a-zA-Z]*f[a-zA-Z]*([[:space:]]|$)|[[:space:]]--force(
[[:space:]]|$))' && force=1
    if [ "$rec" -eq 1 ] && [ "$force" -eq 1 ]; then
      if printf '%s' "$c" | grep -qE '([[:space:]]/([[:space:]]|$)|[[:space:]]/\*|[[:space:]]~|\
$HOME|[[:space:]]\./?([[:space:]]|$)|[[:space:]]\.\./?([[:space:]]|$)|--no-preserve-root)'; then
        block "recursive force-delete aimed at root, home, or the working directory."
      fi
    fi
  fi

  if printf '%s' "$c" | grep -qE 'dd[[:space:]].*of=/dev/'; then
    block "raw disk write via dd — this can destroy a whole disk."
  fi

  case "$command" in
    *":(){ :|:& };:"*) block "fork bomb pattern detected." ;;
  esac
}
```

```
# --- 2: force-push / history rewrite on protected branches ---------------
check_protected_branch() {
  case "$command" in
    *"filter-branch"*)
      block "history-rewriting operation (filter-branch)." ;;
  esac

  # Only inspect what comes AFTER "push", so `cd main && git push origin dev`
  # is not mistaken for a push to main.
  local after_push
  after_push="$(printf '%s' "$command" | sed -n 's/.*[Pp]ush//p')"
  [ -z "$after_push" ] && return 0

  # A leading + in a refspec is a force push even without --force.
  if printf '%s' "$after_push" | grep -qE '(^|[[:space:]])\+[^[:space:]]*(main|master)([[:space:]]|:
|$)'; then
    block "force-push to main/master via a + refspec."
  fi

  local forced=0
  printf '%s' "$after_push" | grep -qE '([[:space:]]-f([[:space:]]|$)|--force([[:space:]]|$)|--
force-with-lease)' && forced=1
  [ "$forced" -eq 0 ] && return 0

  if printf '%s' "$after_push" | grep -qE '(^|[[:space:]]|:)(main|master)([[:space:]]|:|$)'; then
    block "force-push to main/master."
  fi
  if printf '%s' "$after_push" | grep -qE '[[:space:]]--all([[:space:]]|$)'; then
    block "force-push --all rewrites every branch, main/master included."
  fi
}
```

```
# --- 3: skipping safety checks ----------------------------------------
check_skip_safety() {
  case "$command" in
    *"--no-verify"*|*"--no-gpg-sign"*)
      block "a flag that skips commit hooks or signature verification." ;;
  esac

  if printf '%s' "$command" | grep -qE 'chmod[[:space:]]+(-[a-zA-Z]+[[:space:]]+)*777'; then
    block "chmod 777 — overly permissive file permissions."
  fi

  # Any remote fetch piped into any shell: curl|sh, curl|bash,
  # wget|sh, wget | sudo bash, ...
  if printf '%s' "$command" | grep -qE '(curl|wget)[^|]*\|[[:space:]]*(sudo[[:space:]]+)*(
ba|z|k|da|c)?sh([[:space:]]|$)'; then
    block "piping a remote script straight into a shell — read it first."
  fi
}
```

```
# --- 4: secrets -------------------------------------------------------
is_sensitive_path() {
  case "$1" in
    *".env"*|*"id_rsa"*|*"id_ed25519"*|*".aws/credentials"*|*".ssh/"*|*"credentials.json"*) return
0 ;;
    *".git/config") return 0 ;;
  esac
  return 1
}

# Write/Edit/MultiEdit tools
check_sensitive_write() {
  if is_sensitive_path "$file_path"; then
    block "write to a sensitive/credentials path ($file_path). Edit it by hand instead."
  fi
}

# The same files, but written from the shell — `echo ... >> .env`, `tee`.
# The file_path check above only sees the Write/Edit tools.
check_secret_write_via_bash() {
  if printf '%s' "$command" | grep -qE '(>>?|tee[[:space:]]+(-a[[:space:]]+)*)[[:space:]]*[^[:space:
]]*(\.env|id_rsa|id_ed25519|\.aws/credentials|credentials\.json|\.ssh/)'; then
    block "writing to a secrets file from the shell. Edit it by hand instead."
  fi
}
```

```
# --- dispatch ------------------------------------------------------------
case "$hook_event" in
  PreToolUse)
    if [ "$tool_name" = "Bash" ]; then
      check_destructive_fs
      check_protected_branch
      check_skip_safety
      check_secret_write_via_bash
    fi
    case "$tool_name" in
      Write|Edit|MultiEdit) check_sensitive_write ;;
    esac
    ;;
  PostToolUse)
    write_audit_log "RAN"
    ;;
esac

exit 0
```

What this does not protect against

This is a guardrail against mistakes, not a security boundary. It is pattern matching on a command string, and pattern matching can always be walked around:

- Deliberate evasion — base64 into a shell, writing a script and running it, aliases.
- Prompt injection — caught only if the resulting command happens to match a pattern.
- Files pulled in with @ — no tool call, so PreToolUse never runs. Use a Read deny rule.
- Everything not on the list — no sudo rule, no network rule. Five checks, that is it.
- Non-bash environments — needs WSL or Git Bash on Windows, and jq installed.

Setup

1. Create a .claude folder in your project root if it doesn’t exist.
2. Save the settings.json above as .claude/settings.json.
3. Save the guard.sh script above as .claude/guard.sh.
4. Make it executable: chmod +x .claude/guard.sh
5. Make sure jq is installed (brew install jq / apt install jq).
6. Restart Claude Code so it picks up the new settings.

Free resource by Gabrielė · Codeart — more at codeart.lt/en/resources. Questions, or found an edge case these checks miss? Reach out via the socials linked on the site.