Claude Code Safety Mastery

800+ Hours of Battle-Tested Configurations

8 Chapters · 12 Hook Scripts · 50-Point Safety Audit

by yurukusa · Edition 1

Chapter 1: The 5 Essential Hooks Every Claude Code User Needs

This chapter is the free preview of Claude Code Safety Mastery.

You just installed Claude Code. It's powerful — it can read your files, run shell commands, edit code, and push to git. That's the point.

It's also the problem.

Left to run unattended, Claude Code can: - Delete files with `rm -rf` when a cleanup instruction points one directory too high - Force-push to main when it decides that is faster than resolving a conflict - Overwrite `.env` with placeholder values while scaffolding a feature - Publish a package because publishing looked like the logical next step - Run `git reset --hard` to get past a failing test

These five hooks exist to make those specific failures impossible. Install them first, then read Chapter 8 — because the incidents that kept happening after these were in place were a different problem entirely: the guards themselves running on the wrong event, or not running at all.

The fix takes 5 minutes. Five hooks in your `settings.json`, and the most common disasters become impossible.

What Are Hooks?

Hooks are shell scripts that run automatically when Claude Code is about to do something. Think of them as security guards:

- Claude says "I want to run `rm -rf /tmp/build`"
- Your hook checks the command
- If it matches a dangerous pattern → **blocked**
- If it's safe → proceeds normally

Claude Code never sees the hook. It just gets told "that command was blocked." No argument, no retry, no workaround.

Hook #1: Block `rm -rf`

What it prevents: Recursive file deletion. The single most destructive command in any shell.

What this prevents: you ask Claude to "clean up" temporary files. It runs `rm -rf ~/projects/` instead of `rm -rf ~/projects/tmp/` — one directory too high. In autonomous mode there is no confirmation prompt between the decision and the filesystem, and untracked files (local configs, build artifacts, editor settings) do not come back from git.

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "CMD=$(cat | jq -r '.tool_input.command // empty' 2>/dev/
null); if echo \"$CMD\" | grep -qE 'rm\\s+-r(f|F)|rm\\s+-(f|F)r'; then echo
'BLOCKED: rm -rf detected' >&2; exit 2; fi"
          }
        ]
      }
    ]
  }
}
```

How it works: 1. Every time Claude tries to run a Bash command, this hook fires 2. It extracts the command text from Claude's input 3. It checks for `rm -rf` patterns (including `rm -fR`, `rm -Rf`, etc.) 4. If found → prints "BLOCKED" to stderr and exits with code 2 5. Exit code 2 tells Claude Code to cancel the operation

Why exit 2? Claude Code uses three exit codes: - `0` = success (hook ran, no issues) - `1` = error (hook itself failed, but don't block) - `2` = block (stop the operation)

Hook #2: Block Force Push

What it prevents: `git push --force` which overwrites remote history. If your team has pulled the old history, their local branches break.

```
{
  "matcher": "Bash",
  "hooks": [
    {
      "type": "command",
      "command": "CMD=$(cat | jq -r '.tool_input.command // empty' 2>/dev/null);
if echo \"$CMD\" | grep -qE 'git\\s+push.*--force|git\\s+push.*-f\\b'; then echo
'BLOCKED: Force push. Use --force-with-lease.' >&2; exit 2; fi"
    }
  ]
}
```

Tip: `--force-with-lease` is the safe alternative. It only force-pushes if nobody else has pushed since your last fetch.

Hook #3: Protect Main Branch

What it prevents: Direct pushes to `main` or `master`. All changes should go through pull requests.

```
{
  "matcher": "Bash",
  "hooks": [
    {
      "type": "command",
      "command": "CMD=$(cat | jq -r '.tool_input.command // empty' 2>/dev/null);
if echo \"$CMD\" | grep -qE 'git\\s+push\\s+(origin\\s+)?(main|master)\\b'; then
echo 'BLOCKED: Push to main. Use a feature branch.' >&2; exit 2; fi"
    }
  ]
}
```

Hook #4: Guard `.env` Files

What it prevents: Writing to environment files that contain API keys, database credentials, and other secrets.

What this prevents: while scaffolding a feature, Claude "helpfully" creates a clean `.env` with placeholder values, overwriting the real credentials. Nothing errors at write time — the failure arrives later, at the first call that needs the key, in whatever component happens to run first.

```
{
  "matcher": "Write",
  "hooks": [
    {
      "type": "command",
      "command": "FILE=$(cat | jq -r '.tool_input.file_path // empty' 2>/dev/
null); if echo \"$FILE\" | grep -qE '\\.env$|\\.env\\.'; then echo \"BLOCKED:
$FILE is a sensitive file\" >&2; exit 2; fi"
    }
  ]
}
```

Note: This hooks into the `Write` tool, not `Bash`. Claude Code has separate tools for writing files and running commands. You need to guard both.

Hook #5: Block Package Publish

What it prevents: `npm publish`, `yarn publish`, and similar commands that push packages to public registries.

```
{
  "matcher": "Bash",
  "hooks": [
    {
      "type": "command",
      "command": "CMD=$(cat | jq -r '.tool_input.command // empty' 2>/dev/null);
if echo \"$CMD\" | grep -qE 'npm\\s+publish|yarn\\s+publish'; then echo 'BLOCKED:
Package publish requires manual execution.' >&2; exit 2; fi"
    }
  ]
}
```

Complete settings.json

Here's all five hooks combined into a single `settings.json` file. Copy this to `~/.claude/settings.json`:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [{"type": "command", "command": "CMD=$(cat | jq -r
'.tool_input.command // empty' 2>/dev/null); if echo \"$CMD\" | grep -qE 'rm\\s+-
r(f|F)|rm\\s+-(f|F)r'; then echo 'BLOCKED: rm -rf' >&2; exit 2; fi"}]
      },
      {
        "matcher": "Bash",
        "hooks": [{"type": "command", "command": "CMD=$(cat | jq -r
'.tool_input.command // empty' 2>/dev/null); if echo \"$CMD\" | grep -qE 'git\
\s+push.*--force|git\\s+push.*-f\\b'; then echo 'BLOCKED: Force push' >&2; exit 2;
fi"}]
      },
      {
        "matcher": "Bash",
        "hooks": [{"type": "command", "command": "CMD=$(cat | jq -r
'.tool_input.command // empty' 2>/dev/null); if echo \"$CMD\" | grep -qE 'git\
\s+push\\s+(origin\\s+)?(main|master)\\b'; then echo 'BLOCKED: Push to main' >&2;
exit 2; fi"}]
      },
      {
        "matcher": "Write",
        "hooks": [{"type": "command", "command": "FILE=$(cat | jq -r
'.tool_input.file_path // empty' 2>/dev/null); if echo \"$FILE\" | grep -qE '\
\.env$|\\.env\\.'; then echo \"BLOCKED: $FILE\" >&2; exit 2; fi"}]
      },
      {
        "matcher": "Bash",
        "hooks": [{"type": "command", "command": "CMD=$(cat | jq -r
'.tool_input.command // empty' 2>/dev/null); if echo \"$CMD\" | grep -qE 'npm\
\s+publish|yarn\\s+publish'; then echo 'BLOCKED: publish' >&2; exit 2; fi"}]
      }
    ]
  }
}
```

What's Next

These 5 hooks handle the most common disasters. But there are more sophisticated protections:

- **Git protection** (Chapter 2): `git reset --hard`, `git clean`, branch deletion
- **Credential guards** (Chapter 3): API keys in commands, service account files, token detection

- **Token optimization** (Chapter 4): Preventing wasteful file reads, duplicate operations, context bloat
- **Autonomous safety** (Chapter 5): Watchdogs, idle detection, runaway prevention

Each chapter includes ready-to-use hook scripts and the real incidents that motivated them.

Claude Code Safety Mastery — 800+ hours of lessons so you don't have to learn them the hard way.

Chapter 2: Git Protection — Preventing Irreversible Damage

Git is Claude Code's most powerful tool and its most dangerous one. Every `git push`, `git reset`, and `git clean` is a one-way door. This chapter covers the hooks that turn those one-way doors into two-way conversations.

This is the free sample

You have reached the end of the free sample of Claude Code Safety Mastery. The full edition is 56 pages.

The remaining chapters cover Git protection, credential guards, token-spike alerts, autonomous-operation safeguards, multi-agent safety, hooks that silently fail to fire, documented real incidents, and a 50-point safety audit checklist.

The hooks referenced throughout are MIT-licensed and free at github.com/yurukusa/cc-safe-setup.