

Protect your .env with a Claude Code hook

A PreToolUse hook that stops Claude from reading or changing your secrets files, from its file tools and from the shell, in bash and PowerShell, plus the permission rules that back it up. Every file is in the ZIP and has been tested.

FOR	Claude Code on macOS, Linux or Windows
TESTED WITH	Claude Code 2.1.238 · Windows 11 · PowerShell 5.1 and Git Bash
UPDATED	23 October 2026
DOCUMENTATION	code.claude.com/docs/en/hooks · /hooks-guide · /permissions
ARTICLE	ecomlabs.dev/blog/claude-code-hooks-protect-env
AUTHOR	Luis Morales · @luismoralesdev

Contents

Before you start	What you are building
---------------------	-----------------------

Step 01	Create the hooks folder
---------	-------------------------

Step 02	The bash script
---------	-----------------

Step 03	The PowerShell script
---------	-----------------------

Step 04	Register the hook in settings.json
---------	------------------------------------

Step 05	Deny rules: a second layer
---------	----------------------------

Step 06	Test the script without opening Claude Code
---------	---

Step 07	Try it inside Claude Code
---------	---------------------------

Extending it	Keys, migrations and production
-----------------	---------------------------------

If something fails	Troubleshooting
--------------------------	-----------------

When you are done	Checklist
----------------------	-----------

What you are building

A hook is a command Claude Code runs on its own at a fixed point in its lifecycle. The one in this guide runs on the `PreToolUse` event: right before Claude uses a tool. Every time Claude tries to read, edit or write a file, or run a command, Claude Code hands the script a JSON object with the path or the command. If it points at a `.env`, the script exits with code 2 and the tool does not run.

The difference from writing "do not touch `.env`" in `CLAUDE.md` is that an instruction is a request the model can overlook; the hook is code that always runs, in every permission mode.

Three layers, each with its own reach

LAYER	WHAT IT STOPS	WHAT IT DOES NOT STOP
PreToolUse hook	Read, Edit, Write and Grep on <code>.env</code> and <code>.env.*</code> , and Bash or PowerShell commands that name those files.	A command that reaches the file without naming it (<code>grep -r KEY .</code>) or an alias of it: a symlink or the Windows short name <code>ENV~1</code> .
Deny rules	The file tools and the commands Claude Code recognises: <code>cat</code> , <code>head</code> , <code>Get-Content</code> , redirects like <code>> .env</code> .	A script that opens the file by itself (<code>node read.js</code>).
Sandbox	Any process Claude starts, at the operating-system level.	What you do outside Claude Code.

This guide sets up the first two. The sandbox is enabled separately and documented at code.claude.com/docs/en/sandboxing.

Requirements

- Claude Code installed and a project with a `.claude/` folder (step 1 creates it if it is missing).
- macOS or Linux: bash and `jq`. Windows: PowerShell 5.1 or 7, or Git Bash with `jq`.
- `.env` already listed in `.gitignore`. The hook does not replace that: it keeps Claude from reading the file, not the file from reaching the repository.

STEP 01

Create the hooks folder

The scripts live inside the project, next to its settings. At the root of the repository:

```
# macOS, Linux or Git Bash
mkdir -p .claude/hooks

# PowerShell
New-Item -ItemType Directory -Force .claude\hooks
```

On a team, commit this folder: everyone who clones the repository gets the same protection without configuring anything.

STEP 02

The bash script

Save it as `.claude/hooks/protect-env.sh`. It is the same file that ships in the ZIP.

`.claude/hooks/protect-env.sh`

```
#!/usr/bin/env bash
# protect-env.sh: Claude Code PreToolUse hook.
# Blocks Read, Edit, Write and Grep on .env and .env.* (templates excepted),
# and Bash or PowerShell commands that name those files.
# Claude Code sends the event as JSON on stdin.
# exit 0 = the tool runs · exit 2 = blocked, and Claude reads stderr.
# Requires jq: https://jqlang.org/download/

# File names Claude may not read or change. Wildcards allowed.
# To extend it, add names here: "*.pem" "*.key" "id_rsa"
protected=".env" ".env.*")

# Templates with no secrets, which Claude may read and edit.
allowed=".env.example" ".env.sample" ".env.template")

block() {
    echo "Blocked by protect-env: $1. Ask the user to make this change by hand." >&2
    exit 2
}

# True when the name is a template or is not protected.
passes() {
    for p in "${allowed[@]}"; do [[ $1 == $p ]] && return 0; done
    # $p unquoted so bash treats it as a pattern (.env.* matches .env.local).
    for p in "${protected[@]}"; do [[ $1 == $p ]] && return 1; done
    return 0
}

# Without jq there is no safe way to read the input: block rather than allow.
command -v jq >/dev/null 2>&1 || block "jq is missing, so the hook cannot read its input"

input=$(cat)
# Read, Edit and Write send file_path; Grep sends path. Bash and PowerShell send
# command; Grep can also send a glob filter.
path=$(jq -r '.tool_input.file_path // .tool_input.path // empty' <<<"$input") || block "the input is not
valid JSON"
text=$(jq -r '.tool_input.command // .tool_input.glob // empty' <<<"$input")
filter=$(jq -r '.tool_input.glob // empty' <<<"$input")

# .ENV and .env are the same file on Windows and macOS.
```

```

shopt -s nocasematch

# Every name in the command that starts with .env gets the same test as a path:
# .env.example passes, .env.local does not; process.env and .envic never count.
if [ -n "$text" ]; then
    while read -r name; do
        name=${name#[!.]}
        passes "$name" || block "the command mentions $name"
    done < <(grep -oiE '^[^[:alnum:]]*\.[^[:alnum:]]*' <<"$text")
fi

# A Grep glob is a ripgrep pattern, where * also matches a name that starts
# with a dot: * covers .env and *.local covers .env.local. Only extension
# filters pass (*.ts, src/**/*.tsx, *.{ts,tsx}), with extensions from this
# list. If Grep needs another one, add it here.
safe=" ts tsx js jsx mjs cjs json md mdx css scss html liquid py rb go rs php vue svelte yml yaml toml sql sh
ps1 txt "
extension_only='^(\\*\\/|[A-Za-z0-9_+\\/])*\\.(\\{[A-Za-z0-9]+(,[A-Za-z0-9]+)*\\}|[A-Za-z0-9]+)$'
if [ -n "$filter" ]; then
    [[ $filter =~ $extension_only ]] || block "the filter $filter is not an extension filter; use one like *.ts"
    extensions=${filter##*\\.}
    extensions=${extensions%\\}
    IFS=, read -ra list <<<"${extensions%\\}"
    for e in "${list[@]}"; do
        [[ $safe == *" $e " ]] || block "the filter $filter uses .$e, which is not on the safe extension list"
    done
fi

[ -z "$path" ] && exit 0

# Windows sends C:\repo\.env: normalise to / and keep only the file name.
# Also, .env:x is an alternate stream of .env, and .env. is .env.
path=${path//\\//}
name=${path##*/}

# C:.env is .env in the current folder of drive C:, not a stream of "C".
[[ $name =~ ^[A-Za-z]: ]] && name=${name:2}
name=${name%:*}
while [[ $name == *[\. ] ]]; do name=${name%?}; done

passes "$name" || block "$name may contain secrets"
exit 0

```

The decisions that matter:

- **It reads** `tool_input.file_path`. Claude Code always sends an absolute path for Read, Edit and Write, even if Claude wrote a relative one or used `~`. Grep sends `path`, which may be relative; the script only uses the name, so it does not matter.
- **It also checks** `tool_input.command`. Bash and PowerShell calls carry a command, not a path (and Grep can carry a `glob` filter). Every name in the text that starts with `.env` gets the same test as a path: `process.env` does not count and `.env.example.local` does. A Grep glob only passes if it is an extension filter, such as `*.ts`

or `*.{ts,tsx}`, using an extension from the `safe` list; any other is blocked. In ripgrep, `*` also matches a name that starts with a dot: `*.local` would cover `.env.local`.

- **It compares the file name, not the whole path.** Searching the path for ".env" would block `.env.example` as well.
- **It normalises the name.** It strips anything after `:` and any trailing dots: on Windows, `.env:x` is an alternate stream of `.env` and `.env.` is `.env`. It also drops the drive from a path like `C:.env`.
- **Templates go through first.** Claude can read `.env.example` to learn which variables exist without seeing a single value.
- **It ignores case.** On Windows and macOS, `.ENV` is the same file as `.env`.
- **If it cannot read its input, it blocks.** A guard that fails open protects nothing.

On macOS and Linux, make it executable:

```
chmod +x .claude/hooks/protect-env.sh
```

STEP 03

The PowerShell script

For Windows without Git Bash, or if you would rather not install `jq`. Save it as `.claude/hooks/protect-env.ps1`.

`.claude/hooks/protect-env.ps1`

```
# protect-env.ps1: Claude Code PreToolUse hook for Windows (PowerShell 5.1 or 7).
# Blocks Read, Edit, Write and Grep on .env and .env.* (templates excepted),
# and Bash or PowerShell commands that name those files.
# Claude Code sends the event as JSON on stdin.
# exit 0 = the tool runs · exit 2 = blocked, and Claude reads stderr.

# File names Claude may not read or change. Wildcards allowed (-like).
# To extend it, add names here: '*.pem', '*.key', 'id_rsa'
$protected = @('.env', '.env.*')

# Templates with no secrets, which Claude may read and edit.
$allowed = @('.env.example', '.env.sample', '.env.template')

[Console]::OutputEncoding = [System.Text.Encoding]::UTF8

function Block($reason) {
    [Console]::Error.WriteLine("Blocked by protect-env: $reason. Ask the user to make this change by hand.")
    exit 2
}

# Any unexpected error blocks the call instead of letting it through.
$ErrorActionPreference = 'Stop'
trap { Block 'unexpected error in the hook' }

# True when the name is a template or is not protected.
# -contains and -like ignore case: .ENV and .env are the same file on Windows.
function Passes($name) {
```

```

    if ($allowed -contains $name) { return $true }
    foreach ($p in $protected) { if ($name -like $p) { return $false } }
    return $true
}

# If the input cannot be read, block rather than let the call through.
try {
    $hookInput = [Console]::In.ReadToEnd() | ConvertFrom-Json
} catch {
    Block 'the hook could not read its input'
}

# Read, Edit and Write send file_path; Grep sends path. Bash and PowerShell send
# command; Grep can also send a glob filter.
$path = $hookInput.tool_input.file_path
if (-not $path) { $path = $hookInput.tool_input.path }
$text = $hookInput.tool_input.command
if (-not $text) { $text = $hookInput.tool_input.glob }

# Every name in the command that starts with .env gets the same test as a path:
# .env.example passes, .env.local does not; process.env and .envrc never count.
if ($text) {
    foreach ($m in [regex]::Matches($text, '(?<![\w.-])\.env[\w.-]*', 'IgnoreCase')) {
        if (-not (Passes $m.Value)) { Block "the command mentions $($m.Value)" }
    }
}

# A Grep glob is a ripgrep pattern, where * also matches a name that starts
# with a dot: * covers .env and *.local covers .env.local. Only extension
# filters pass (*.ts, src/**/*.tsx, *.{ts,tsx}), with extensions from this
# list. If Grep needs another one, add it here.
$safe = @('ts', 'tsx', 'js', 'jsx', 'mjs', 'cjs', 'json', 'md', 'mdx', 'css', 'scss', 'html', 'liquid', 'py',
'rb', 'go', 'rs', 'php', 'vue', 'svelte', 'yaml', 'yml', 'toml', 'sql', 'sh', 'ps1', 'txt')
$filter = $hookInput.tool_input.glob
if ($filter) {
    if ($filter -notmatch '^(\\*\\/|\\[\\w-\\+\\]\\*\\.(\\[a-z0-9\\+\\(\\[a-z0-9\\+\\]\\*\\)|\\[a-z0-9\\+\\])$') {
        Block "the filter $filter is not an extension filter; use one like *.ts"
    }
    foreach ($e in ($filter -split '\\*\\.')[1..$filter.Length].Trim('{', '}') -split ',') {
        if ($safe -notcontains $e) { Block "the filter $filter uses .$e, which is not on the safe extension list" }
    }
}

if (-not $path) { exit 0 }

# The name is whatever follows the last \ or /. Not GetFileName: on PowerShell 5.1
# it splits at ':', so .env:x (an alternate stream of .env) would slip through.
# Windows also ignores trailing dots and spaces: .env. is .env.
$name = ($path -split '[\\\/]')[1]
# C:.env is .env in the current folder of drive C:, not a stream of 'C'.

```

```
$name = $name -replace '^[A-Za-z]:', ''
$name = ($name -split ':')[0].TrimEnd(' ')

if (-not (Passes $name)) { Block "$name may contain secrets" }
exit 0
```

- `ConvertFrom-Json` stands in for `jq`.
- `$ErrorActionPreference = 'Stop'` and a `trap` make any unexpected error block the call instead of letting it through.
- The name comes from splitting the path on `\` and `/`, not from `GetFileName`: on PowerShell 5.1 it splits at `:` and `.env:x` would slip through.
- `-like`, `-match` and `-contains` ignore case, like the Windows file system.
- If you translate the messages, keep them in plain ASCII: Windows PowerShell 5.1 reads a UTF-8 file without a BOM as ANSI.

STEP 04

Register the hook in settings.json

A project's hooks go in `.claude/settings.json`. For macOS, Linux and Windows with Git Bash:

`.claude/settings.json`

```
{
  "permissions": {
    "deny": [
      "Read(.env)",
      "Read(.env.*)",
      "Read(!.env.example)",
      "Edit(.env)",
      "Edit(.env.*)",
      "Edit(!.env.example)"
    ]
  },
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Read|Edit|Write|Grep|Bash|PowerShell",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/protect-env.sh"
          }
        ]
      }
    ]
  }
}
```

- `"matcher": "Read|Edit|Write|Grep|Bash|PowerShell"` is a list of exact tool names separated by `|`. It is case-sensitive: `edit` matches nothing. `PowerShell` is Claude Code's shell tool on Windows; on macOS and Linux that name simply never matches.
- `$CLAUDE_PROJECT_DIR` is the project root. It is quoted because the path may contain spaces.

For Windows with PowerShell, use `windows/settings.json` instead:

windows/settings.json

```
{
  "permissions": {
    "deny": [
      "Read(.env)",
      "Read(.env.*)",
      "Read(!.env.example)",
      "Edit(.env)",
      "Edit(.env.*)",
      "Edit(!.env.example)"
    ]
  },
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Read|Edit|Write|Grep|Bash|PowerShell",
        "hooks": [
          {
            "type": "command",
            "command": "powershell.exe",
            "args": [
              "-NoProfile",
              "-ExecutionPolicy",
              "Bypass",
              "-File",
              "${CLAUDE_PROJECT_DIR}/.claude/hooks/protect-env.ps1"
            ]
          }
        ]
      }
    ]
  }
}
```

With `args`, Claude Code starts `powershell.exe` directly, with no shell in between: each element arrives as one argument, so the path needs no quoting, and `-ExecutionPolicy Bypass` keeps Windows' execution policy from refusing the script.

CAUTION

If you already have a `.claude/settings.json`, do not overwrite it. Copy the `permissions.deny` and `hooks.PreToolUse` keys into yours. JSON allows no comments and no trailing commas.

Deny rules: a second layer

The six `permissions.deny` rules already ship in both settings files. `Read(.env)` blocks reading any `.env` in the project, at any depth; `Read(.env.*)` does the same for `.env.local`, `.env.production` and the rest; `Read(!.env.example)` gives the template back. The three `Edit` rules repeat the idea for edits.

Claude Code also applies them to the shell commands it recognises, such as `cat`, `head`, `tail`, `sed`, `Get-Content` and redirects. It is a second check, done by Claude Code rather than by your script: if the hook breaks or someone deletes it, the rules are still there.

CAUTION

The first version of this hook only matched Read, Edit and Write. In testing, in `acceptEdits` mode, Claude read the `.env` with `Get-Content` and appended a line with `Add-Content` through its PowerShell tool, without asking. That is why the hook also checks commands and why the deny rules stay. And if, in an interactive session, Claude asks you to approve a command that reads or writes `.env`, the answer is no.

Test the script without opening Claude Code

A hook is a program that reads JSON on stdin. You can test it by piping in the same JSON Claude Code would send:

```
# bash
echo '{"tool_name":"Edit","tool_input":{"file_path":"/repo/.env"}}' | bash .claude/hooks/protect-env.sh
echo $? # 2

# PowerShell
'{"tool_name":"Edit","tool_input":{"file_path":"C:\\repo\\.env"}}' |
powershell -NoProfile -ExecutionPolicy Bypass -File .claude\\hooks\\protect-env.ps1
$LASTEXITCODE # 2
```

The ZIP includes a script that runs twenty-five cases at once: ten paths, six Grep calls, eight shell commands and one input that is not JSON.

```
bash test/test-hook.sh
powershell -NoProfile -ExecutionPolicy Bypass -File test\\test-hook.ps1
```

Each case prints a line that starts with `ok` or `FAIL`, and the last one says "All good." when every case passed. If any line says `FAIL`, your script does not behave like the one in the ZIP: compare them line by line before you register it.

Try it inside Claude Code

1. Open Claude Code in the project and type `/hooks`. The hook should be listed under `PreToolUse`. If it is not, check that the JSON is valid and that the file is `.claude/settings.json`.

2. Ask for something that touches the file: "Add the line `DEBUG=true` to the end of `.env`".
3. Claude receives the block and explains it. `Ctrl+O` shows the full transcript, where the result of the read looks like this:

```
PreToolUse:Read hook error: [...]: Blocked by protect-env:
.env may contain secrets. Ask the user to make this change by hand.
```

To test the shell route, also ask it to use Bash or PowerShell if a tool fails. The attempt should end in a `PreToolUse:PowerShell hook error` (or `Bash`) saying "the command mentions a `.env` file".

To see each hook's exit code and stderr, start Claude Code with `claude --debug-file /tmp/claude.log` and read that file.

EXTENDING IT

Keys, migrations and production

Private keys

Add the names to the script's list and, for the same cover in the shell, to the deny rules:

```
protected=( ".env" ".env.*" "*.pem" "*.key" "id_rsa" "id_ed25519" )
```

```
"Read(*.pem)", "Edit(*.pem)", "Read(*.key)", "Edit(*.key)"
```

Applied migrations

A migration that already ran in production is fixed with a new migration, not by editing it. This block allows new files in `migrations/` but blocks changes to existing ones. In bash, add it right before the last check, `passes "$name"`

```
|| block ...:
```

```
tool=$(jq -r '.tool_name // empty' <<"$input")
if [[ $path == */migrations/* && $tool != "Read" && -f $path ]]; then
    block "that migration already exists; write a new one instead of editing it"
fi
```

In PowerShell, right before the line `if (-not (Passes $name)):`

```
if ($path -match '[\\/]migrations[\\/]' -and $hookInput.tool_name -ne 'Read' -and (Test-Path -LiteralPath $path)) {
    Block 'that migration already exists; write a new one instead of editing it'
}
```

Production files

`.env.production` is already covered by `.env.*`. If your project keeps production settings under another name, add it to `protected`, for example `"*.production.json"`.

Troubleshooting

SYMPTOM	LIKELY CAUSE	FIX
<code>/hooks</code> does not list the hook	Invalid JSON or a file in another folder	Remove trailing commas and comments; the path is <code>.claude/settings.json</code>
Everything is blocked with "jq is missing"	<code>jq</code> is not installed	Install it, or use the PowerShell version
The hook never runs on macOS or Linux	The script is not executable	<code>chmod +x .claude/hooks/protect-env.sh</code>
"blocking because Git Bash wasn't found"	Windows without Git Bash and a hook written for bash	Use <code>windows/settings.json</code>
<code>.env.example</code> is blocked too	A script that searches for ".env" as text	Compare the file name, as this one does
Claude changes <code>.env</code> from the shell	The matcher does not include <code>Bash PowerShell</code>	Use the full matcher from step 4 and the deny rules from step 5
A legitimate command is blocked	It mentions <code>.env</code> without opening it, like <code>echo .env >> .gitignore</code>	Run it yourself, or adjust the script's expression

Checklist

- ☐ `.claude/hooks/protect-env.sh` or `.ps1` is in the project.
- ☐ `.claude/settings.json` registers the hook with `"matcher": "Read|Edit|Write|Grep|Bash|PowerShell"`.
- ☐ The `deny` rules include the `!.env.example` carve-out.
- ☐ `test-hook` ends with "All good."
- ☐ `/hooks` lists the hook under `PreToolUse`.
- ☐ You asked Claude to edit `.env` and it was blocked.
- ☐ `.env` is still in `.gitignore`.
- ☐ `.claude/settings.json` is committed so the team gets the same protection.