

Protege tu .env con un hook de Claude Code

Un hook PreToolUse que impide a Claude leer o modificar tus archivos de secretos, en bash y en PowerShell, más las reglas de permisos que cubren lo que el hook no ve. Todo el código está en el ZIP y está probado.

PARA	Claude Code en macOS, Linux o Windows
PROBADO CON	Claude Code 2.1.238 · Windows 11 · PowerShell 5.1 y Git Bash
ACTUALIZADA	23 de octubre de 2026
DOCUMENTACIÓN	code.claude.com/docs/en/hooks · /hooks-guide · /permissions
ARTÍCULO	ecomlabs.dev/g/hook
AUTOR	Luis Morales · @luismoralesdev

Índice

Antes de empezar	Qué vas a construir
Paso 01	Crea la carpeta de hooks
Paso 02	El script en bash
Paso 03	El script en PowerShell
Paso 04	Registra el hook en settings.json
Paso 05	Las reglas deny: una segunda capa
Paso 06	Prueba el script sin abrir Claude Code
Paso 07	Pruébalo dentro de Claude Code
Para ampliarlo	Llaves, migraciones y producción
Si algo falla	Solución de problemas
Al terminar	Checklist

Qué vas a construir

Un hook es un comando que Claude Code ejecuta solo, en un momento fijo de su ciclo de trabajo. El de esta guía corre en el evento `PreToolUse`: justo antes de que Claude use una herramienta. Cada vez que Claude intenta leer, editar o escribir un archivo, o correr un comando, Claude Code le pasa al script un JSON con la ruta o el comando. Si apunta a un `.env`, el script sale con código 2 y la herramienta no se ejecuta.

La diferencia con escribir «no toques el .env» en `CLAUDE.md` es que una instrucción es una petición que el modelo puede pasar por alto; el hook es código que corre siempre, en cualquier modo de permisos.

Tres capas, cada una con su alcance

CAPA	QUÉ DETIENE	QUÉ NO DETIENE
Hook PreToolUse	Read, Edit, Write y Grep sobre <code>.env</code> y <code>.env.*</code> , y los comandos de Bash o PowerShell que nombran esos archivos.	Un comando que llega al archivo sin nombrarlo (<code>grep -r CLAVE .</code>) o un alias suyo: un enlace simbólico o el nombre corto <code>ENV~1</code> de Windows.
Reglas deny	Las herramientas de archivos y los comandos que Claude Code reconoce: <code>cat</code> , <code>head</code> , <code>Get-Content</code> , redirecciones <code>> .env</code> .	Un script que abre el archivo por su cuenta (<code>node leer.js</code>).
Sandbox	Cualquier proceso que lance Claude, a nivel del sistema operativo.	Lo que tú hagas fuera de Claude Code.

Esta guía instala las dos primeras. El sandbox se activa aparte y está documentado en code.claude.com/docs/en/sandboxing.

Requisitos

- Claude Code instalado y un proyecto con su carpeta `.claude/` (se crea en el paso 1 si no existe).
- macOS o Linux: bash y `jq`. Windows: PowerShell 5.1 o 7, o Git Bash con `jq`.
- El `.env` ya debe estar en `.gitignore`. El hook no reemplaza eso: evita que Claude lo lea, no que se suba al repositorio.

Crea la carpeta de hooks

Los scripts viven dentro del proyecto, junto a su configuración. En la raíz del repositorio:

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

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

Si trabajas en equipo, esta carpeta se versiona: así cada persona que clone el repositorio tiene la misma protección sin configurar nada.

PASO 02

El script en bash

Guárdalo como `.claude/hooks/proteger-env.sh`. Es el mismo archivo que viene en el ZIP.

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

```
#!/usr/bin/env bash
# proteger-env.sh: hook PreToolUse de Claude Code.
# Bloquea Read, Edit, Write y Grep sobre .env y .env.* (salvo las plantillas),
# y los comandos de Bash o PowerShell que nombran esos archivos.
# Claude Code le pasa el evento en JSON por stdin.
# exit 0 = la herramienta sigue · exit 2 = se bloquea y Claude lee el stderr.
# Requiere jq: https://jqlang.org/download/

# Nombres de archivo que Claude no puede leer ni modificar. Acepta comodines.
# Para ampliarlo, agrega nombres aquí: "*.pem" "*.key" "id_rsa"
protegidos=".env" ".env.*")

# Plantillas sin secretos que sí puede leer y editar.
permitidos=".env.example" ".env.sample" ".env.template")

bloquear() {
    echo "Bloqueado por proteger-env: $1. Pide al usuario que haga este cambio a mano." >&2
    exit 2
}

# Si cuando el nombre es una plantilla o no está protegido.
libre() {
    for p in "${permitidos[@]}"; do [[ $1 == $p ]] && return 0; done
    # $p sin comillas: así bash lo trata como patrón (.env.* incluye .env.local).
    for p in "${protegidos[@]}"; do [[ $1 == $p ]] && return 1; done
    return 0
}

# Sin jq no hay forma segura de leer la entrada: mejor bloquear que dejar pasar.
command -v jq >/dev/null 2>&1 || bloquear "falta jq para leer la entrada del hook"

entrada=$(cat)
# Read, Edit y Write mandan file_path; Grep, path. Bash y PowerShell mandan
# command; Grep, además, un filtro glob.
ruta=$(jq -r '.tool_input.file_path // .tool_input.path // empty' <<<"$entrada") || bloquear "la entrada no es
JSON válido"
texto=$(jq -r '.tool_input.command // .tool_input.glob // empty' <<<"$entrada")
filtro=$(jq -r '.tool_input.glob // empty' <<<"$entrada")

# .ENV y .env son el mismo archivo en Windows y macOS.
```

```
shopt -s nocasematch

# Cada nombre que empieza con .env dentro del comando pasa la misma prueba que
# una ruta: .env.example sí, .env.local no; process.env y .envrc ni cuentan.
if [ -n "$texto" ]; then
    while read -r nombre; do
        nombre=${nombre#[!.]}
        libre "$nombre" || bloquear "el comando menciona $nombre"
    done < <(grep -oiE '^[^[:alnum:]]*\.[^[:alnum:]]*' <<"$texto")
fi

# El filtro de Grep es un patrón de ripgrep, donde * también coincide con un
# nombre que empieza con punto: * cubre .env y *.local cubre .env.local. Solo
# pasan filtros por extensión (*.ts, src/**/*.tsx, *.{ts,tsx}) con extensiones
# de esta lista. Si Grep necesita otra, agrégala aquí.
seguras=" 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 psi txt "
solo_extension='^(\\*\\*/|[A-Za-z0-9_-]+/)*\\.(\\{[A-Za-z0-9]+(,[A-Za-z0-9]+)*\\}|[A-Za-z0-9]+)$'
if [ -n "$filtro" ]; then
    [[ $filtro =~ $solo_extension ]] || bloquear "el filtro $filtro no es solo por extensión; usa uno como *.ts"
    extensiones=${filtro##*\\.}
    extensiones=${extensiones#\\{ }
    IFS=, read -ra lista <<<"${extensiones%\\}"
    for e in "${lista[@]}"; do
        [[ $seguras == "$e" ]] || bloquear "el filtro $filtro usa .$e, que no está en la lista de extensiones
seguras"
    done
fi

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

# Windows manda C:\\repo\\.env: se normaliza a / y se toma solo el nombre.
# Además, .env:x es un flujo alterno de .env y .env. es .env.
ruta=${ruta//\\//}
nombre=${ruta##*/}
# C:.env es .env en la carpeta actual de la unidad C:, no un flujo de "C".
[[ $nombre =~ ^[A-Za-z]: ]] && nombre=${nombre:2}
nombre=${nombre%:*}
while [[ $nombre == *\\. \\ ] ]; do nombre=${nombre%?}; done

libre "$nombre" || bloquear "$nombre puede contener secretos"
exit 0
```

Las decisiones que importan:

- **Lee** `tool_input.file_path`. Claude Code siempre manda una ruta absoluta para Read, Edit y Write, aunque Claude haya escrito una relativa o con `~`. Grep manda `path`, que puede ser relativa; el script solo usa el nombre, así que da igual.
- **Revisa también** `tool_input.command`. Para Bash y PowerShell no hay ruta, hay un comando (y Grep puede traer un filtro `glob`). Cada nombre que empieza con `.env` dentro del texto pasa la misma prueba que una ruta: `process.env` no cuenta y `.env.example.local` sí. El filtro de Grep solo pasa si es por extensión, como `*.ts` o

`*.{ts,tsx}`, y con una extensión de la lista `seguras`; cualquier otro se bloquea. En ripgrep, `*` también coincide con un nombre que empieza con punto: `*.local` cubriría `.env.local`.

- **Compara el nombre del archivo, no la ruta completa.** Buscar el texto «.env» en la ruta también bloquearía `.env.example`.
- **Normaliza el nombre.** Quita lo que sigue a `:` y los puntos finales: en Windows, `.env:x` es un flujo alterno de `.env` y `.env.` es `.env`. También quita la unidad de una ruta como `C:.env`.
- **Las plantillas pasan primero.** Claude puede leer `.env.example` para saber qué variables existen sin ver ningún valor.
- **Sin distinguir mayúsculas.** En Windows y en macOS, `.ENV` es el mismo archivo que `.env`.
- **Si no puede leer la entrada, bloquea.** Un guardia que falla abierto no protege nada.

En macOS y Linux, dale permiso de ejecución:

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

PASO 03

El script en PowerShell

Para Windows sin Git Bash, o si prefieres no instalar `jq`. Guárdalo como `.claude/hooks/proteger-env.ps1`.

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

```
# proteger-env.ps1: hook PreToolUse de Claude Code para Windows (PowerShell 5.1 o 7).
# Bloquea Read, Edit, Write y Grep sobre .env y .env.* (salvo las plantillas),
# y los comandos de Bash o PowerShell que nombran esos archivos.
# Claude Code le pasa el evento en JSON por stdin.
# exit 0 = la herramienta sigue · exit 2 = se bloquea y Claude lee el stderr.
# Los mensajes van sin acentos: PowerShell 5.1 lee como ANSI un .ps1 UTF-8 sin BOM.

# Nombres de archivo que Claude no puede leer ni modificar. Acepta comodines (-like).
# Para ampliarlo, agrega nombres aquí: '*.pem', '*.key', 'id_rsa'
$protegidos = @('.env', '.env.*')

# Plantillas sin secretos que sí puede leer y editar.
$permitidos = @('.env.example', '.env.sample', '.env.template')

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

function Bloquear($motivo) {
    [Console]::Error.WriteLine("Bloqueado por proteger-env: $motivo. Pide al usuario que haga este cambio a mano.")
    exit 2
}

# Cualquier error inesperado bloquea en lugar de dejar pasar.
$ErrorActionPreference = 'Stop'
trap { Bloquear 'error inesperado en el hook' }

# Verdadero cuando el nombre es una plantilla o no está protegido.
```

```

# -contains y -like ignoran mayúsculas: .ENV y .env son el mismo archivo en Windows.
function Libre($nombre) {
    if ($permitidos -contains $nombre) { return $true }
    foreach ($p in $protegidos) { if ($nombre -like $p) { return $false } }
    return $true
}

# Si la entrada no se puede leer, mejor bloquear que dejar pasar.
try {
    $entrada = [Console]::In.ReadToEnd() | ConvertFrom-Json
} catch {
    Bloquear 'no se pudo leer la entrada del hook'
}

# Read, Edit y Write mandan file_path; Grep, path. Bash y PowerShell mandan
# command; Grep, además, un filtro glob.
$ruta = $entrada.tool_input.file_path
if (-not $ruta) { $ruta = $entrada.tool_input.path }
$texto = $entrada.tool_input.command
if (-not $texto) { $texto = $entrada.tool_input.glob }

# Cada nombre que empieza con .env dentro del comando pasa la misma prueba que
# una ruta: .env.example sí, .env.local no; process.env y .envrc ni cuentan.
if ($texto) {
    foreach ($m in [regex]::Matches($texto, '(?![\w.-])\.env[\w.-]*', 'IgnoreCase')) {
        if (-not (Libre $m.Value)) { Bloquear "el comando menciona $($m.Value)" }
    }
}

# El filtro de Grep es un patrón de ripgrep, donde * también coincide con un
# nombre que empieza con punto: * cubre .env y *.local cubre .env.local. Solo
# pasan filtros por extensión (*.ts, src/**/*.tsx, *.{ts,tsx}) con extensiones
# de esta lista. Si Grep necesita otra, agrégala aquí.
$seguras = @('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')
$filtro = $entrada.tool_input.glob
if ($filtro) {
    if ($filtro -notmatch '^(\\*\\/|\\w+\\/)*\\.(\\[[a-z0-9]+(\\[[a-z0-9]+\\])*\\]|\\[[a-z0-9]+\\])$') {
        Bloquear "el filtro $filtro no es solo por extension; usa uno como *.ts"
    }
    foreach ($e in ($filtro -split '\\.')[1..$].Trim('{', '}') -split ',') {
        if ($seguras -notcontains $e) { Bloquear "el filtro $filtro usa .$e, que no esta en la lista de
extensiones seguras" }
    }
}

if (-not $ruta) { exit 0 }

# El nombre es lo que sigue a la última \ o /. GetFileName no sirve: en
# PowerShell 5.1 corta en ':' y .env:x (un flujo alterno de .env) pasaría.
# Windows también ignora los puntos y espacios finales: .env. es .env.

```

```
$nombre = ($ruta -split '[\\/]')[-1]
# C:.env es .env en la carpeta actual de la unidad C:, no un flujo de 'C'.
$nombre = $nombre -replace '^[A-Za-z]:', ''
$nombre = ($nombre -split ':')[0].TrimEnd('. ')

if (-not (Libre $nombre)) { Bloquear "$nombre puede contener secretos" }
exit 0
```

- `ConvertFrom-Json` reemplaza a `jq`.
- `$ErrorActionPreference = 'Stop'` y un `trap` hacen que cualquier error inesperado bloquee en lugar de dejar pasar.
- El nombre se saca partiendo la ruta en `\` y `/`, no con `GetFileName`: en PowerShell 5.1 corta en `:` y `.env:x` pasaría.
- `-like` y `-contains` no distinguen mayúsculas, igual que el sistema de archivos de Windows.
- Los mensajes van sin acentos porque Windows PowerShell 5.1 lee como ANSI los archivos UTF-8 sin BOM.

PASO 04

Registra el hook en settings.json

Los hooks de un proyecto van en `.claude/settings.json`. Para macOS, Linux y Windows con 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/protector-env.sh"
          }
        ]
      }
    ]
  }
}
```

- `"matcher": "Read|Edit|Write|Grep|Bash|PowerShell"` es una lista de nombres exactos de herramientas, separados por `|`. Distingue mayúsculas: `edit` no coincide con nada. `PowerShell` es la herramienta de terminal de Claude Code en Windows; en macOS y Linux ese nombre simplemente no coincide.
- `$CLAUDE_PROJECT_DIR` es la raíz del proyecto. Va entre comillas porque la ruta puede tener espacios.

Para Windows con PowerShell, usa en su lugar `windows/settings.json`:

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/protector-env.ps1"
            ]
          }
        ]
      }
    ]
  }
}
```

Con `args`, Claude Code ejecuta `powershell.exe` directamente, sin pasar por una shell: cada elemento llega como un argumento, así que la ruta no necesita comillas, y `-ExecutionPolicy Bypass` evita que la política de ejecución de Windows bloquee el script.

CUIDADO

Si ya tienes un `.claude/settings.json`, no lo sobrescribas. Copia las claves `permissions.deny` y `hooks.PreToolUse` dentro del tuyo. El JSON no admite comentarios ni comas al final.

Las reglas deny: una segunda capa

Las seis reglas de `permissions.deny` ya vienen en los dos `settings.json`. `Read(.env)` bloquea leer cualquier `.env` del proyecto, a cualquier profundidad; `Read(.env.*)` hace lo mismo con `.env.local`, `.env.production` y compañía; `Read(!.env.example)` devuelve el permiso a la plantilla. Las tres de `Edit` repiten la idea para las ediciones.

Claude Code también las aplica a los comandos de terminal que reconoce, como `cat`, `head`, `tail`, `sed`, `Get-Content` y las redirecciones. Es una segunda revisión, hecha por Claude Code y no por tu script: si el hook falla o alguien lo borra, las reglas siguen ahí.

CUIDADO

La primera versión de este hook solo miraba Read, Edit y Write. En la prueba, en modo `acceptEdits`, Claude leyó el `.env` con `Get-Content` y le agregó la línea con `Add-Content` desde su herramienta de PowerShell, sin pedir permiso. Por eso el hook revisa también los comandos y por eso las reglas deny se quedan. Y si en una sesión interactiva Claude te pide aprobar un comando que lee o escribe `.env`, la respuesta es no.

Prueba el script sin abrir Claude Code

Un hook es un programa que lee JSON por stdin. Puedes probarlo pasándole el mismo JSON que mandaría Claude Code:

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

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

El ZIP trae un script que corre veinticinco casos de una vez: diez de rutas, seis de Grep, ocho de comandos de terminal y una entrada que no es JSON.

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

Cada caso imprime una línea que empieza con `ok` o `FALLA`, y la última dice «Todo en orden.» si pasaron todos. Si alguna dice `FALLA`, el script no se comporta como el del ZIP: compáralo línea por línea antes de registrarlo.

Pruébalo dentro de Claude Code

1. Abre Claude Code en el proyecto y escribe `/hooks`. El hook debe aparecer bajo `PreToolUse`. Si no aparece, revisa que el JSON sea válido y que el archivo esté en `.claude/settings.json`.
2. Pídele algo que toque el archivo: «Agrega la línea `DEBUG=true` al final de `.env`».
3. Claude recibe el bloqueo y te lo explica. Con `Ctrl+0` ves la transcripción completa. En nuestra prueba, el resultado de la lectura fue este:

```
PreToolUse:Read hook error: [...]: Bloqueado por proteger-env:
.env puede contener secretos. Píde al usuario que haga este cambio a mano.
```

Para probar la vía de la terminal, pídele además que use Bash o PowerShell si una herramienta falla. El intento debe terminar en un `PreToolUse:PowerShell hook error` (o `Bash`) con «el comando menciona un archivo `.env`».

Para ver el código de salida y el stderr de cada hook, arranca con `claude --debug-file /tmp/claude.log` y lee ese archivo.

— PARA AMPLIARLO

Llaves, migraciones y producción

Llaves privadas

Agrega los nombres a la lista del script y, si quieres la misma cobertura en terminal, a las reglas deny:

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

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

Migraciones ya aplicadas

Una migración que ya corrió en producción se corrige con una migración nueva, no editándola. Este bloque deja crear archivos nuevos en `migrations/` pero bloquea cambiar los que existen. En bash, agrégalo justo antes de la última comprobación, `libre "$nombre" || bloquear ...`:

```
herramienta=$(jq -r '.tool_name // empty' <<"$entrada")
if [[ $ruta == */migrations/* && $herramienta != "Read" && -f $ruta ]]; then
    bloquear "esa migración ya existe; crea una nueva en lugar de editarla"
fi
```

En PowerShell, justo antes de la línea `if (-not (Libre $nombre))`:

```
if ($ruta -match '[\\/]migrations[\\/]' -and $entrada.tool_name -ne 'Read' -and (Test-Path -LiteralPath
$ruta)) {
    Bloquear 'esa migracion ya existe; crea una nueva en lugar de editarla'
}
```

Archivos de producción

`.env.production` ya queda cubierto por `.env.*`. Si tu proyecto guarda configuración de producción con otro nombre, agrégalo a `protegidos`, por ejemplo `"*.production.json"`.

— SI ALGO FALLA

Solución de problemas

SÍNTOMA	CAUSA PROBABLE	ARREGLO
<code>/hooks</code> no muestra el hook	JSON inválido o archivo en otra carpeta	Quita comas finales y comentarios; la ruta es <code>.claude/settings.json</code>
Todo se bloquea con «falta jq»	<code>jq</code> no está instalado	Instálalo, o usa la versión de PowerShell
El hook no corre en macOS o Linux	El script no es ejecutable	<code>chmod +x .claude/hooks/proteger-env.sh</code>
«blocking because Git Bash wasn't found»	Windows sin Git Bash y un hook escrito en bash	Usa <code>windows/settings.json</code>
<code>.env.example</code> también se bloquea	Un script que busca «.env» como texto	Compara el nombre del archivo, como el de esta guía
Claude modifica <code>.env</code> desde la terminal	El matcher no incluye <code>Bash PowerShell</code>	Usa el matcher completo del paso 4 y las reglas deny del paso 5
Un comando legítimo queda bloqueado	Menciona <code>.env</code> aunque no lo abra, como <code>echo .env >> .gitignore</code>	Hazlo tú en tu terminal, o ajusta la expresión del script

— AL TERMINAR

Checklist

- ☐ `.claude/hooks/proteger-env.sh` o `.ps1` está en el proyecto.
- ☐ `.claude/settings.json` registra el hook con `"matcher": "Read|Edit|Write|Grep|Bash|PowerShell"`.
- ☐ Las reglas `deny` incluyen la excepción `!.env.example`.
- ☐ `probar-hook` termina con «Todo en orden.».
- ☐ `/hooks` muestra el hook bajo `PreToolUse`.
- ☐ Pediste a Claude que editara `.env` y lo bloqueó.
- ☐ `.env` sigue en `.gitignore`.
- ☐ `.claude/settings.json` está versionado para que el equipo tenga la misma protección.