PRODUCTION SECURITY BLUEPRINT • KAROL JASZTAL · HEAD OF OPERATIONS & AI SYSTEMS LEADER

Why Asking AI to "Be Careful" Fails: How I Vetted 11,000+ Files and Built Real OS Firewalls for Autonomous Agents

One rogue AI command or prompt injection can wipe a filesystem or leak your API keys in seconds. I audited 11,000+ community skills, scrapped the 49% that were dangerous traps, and hooked hard OS firewalls directly into three agent harnesses—blocking rogue execution with zero breaches across 241 production tools.

CANDIDATE FILES	CONTROL PLANE	RUNTIME FIREWALL	HARNESS PARITY	UPSTREAM DRIFT
11,000+	241 Primitives	0 Breaches	3 Harnesses	100% Catch
5,363 Rejected (49%)	180 Skills • 33 MCP	Zero Leaks	100% Shared Config	Radar v2 Engine
Malicious Traps Blocked	50k+ Vault Files	Hard OS Firewalls	Claude • DSH • Antigravity	3-Way Security Diffing

01 // THREAT LANDSCAPE & ATTACK VECTORS

The Enterprise Autonomous Agent Dilemma: Why Asking AI Nicely Fails

Giving autonomous LLMs raw terminal access and API keys without guardrails is a recipe for disaster. The industry standard of 'model self-policing'—putting "Please don't delete files" into a system prompt—is a complete joke. Hidden prompt injections in untrusted web pages, sneaky tracking URLs in open-source tools, and model hallucinations easily bypass prompt instructions. Real security requires deterministic, out-of-band execution firewalls that kill dangerous commands before they ever reach the operating system.

THREAT VECTOR 01

Remote Code Execution & Disk Wiping

Hallucinated or poisoned models run destructive shell commands like `rm -rf /`, `del /s /q`, `git push --force`, or unconstrained database drops. A single bad command wipes uncommitted engineering work or corrupts remote repositories.

THREAT VECTOR 02

Credential & Secret Exfiltration

Agents reading project directories vacuum up `.env` files, `id_rsa` SSH keys, and cloud tokens, leaking private credentials into external third-party LLM inference logs or outbound HTTP endpoints.

THREAT VECTOR 03

Indirect Prompt Injection

Malicious instructions hidden in fetched web pages, scraped HTML comments, or dependency READMEs hijack the agent's reasoning loop, tricking the model into executing secondary unauthorized actions under elevated developer permissions.

THREAT VECTOR 04

Supply-Chain Poisoning & MCP Tool Drift

Third-party Model Context Protocol (MCP) servers and external GitHub repos push silent upstream updates, injecting unauthorized background daemons, crypto-miners, or telemetry beacons into previously trusted tools.

Probabilistic System Prompts vs. Karol's Deterministic Zero-Trust Architecture

SECURITY DIMENSION	CONVENTIONAL "PROMPT-GUARDED" AGENTS	KAROL'S ZERO-TRUST ARCHITECTURE
Enforcement Layer	System prompt instructions ("*Please do not delete files*")	**Out-of-band PreToolUse hooks compiled in native PowerShell / OS**
Failure Mode	**Probabilistic; fails under jailbreaks, long context, or role-play**	**Deterministic; hard-blocked at exit code 2 before shell executes**
Secret Protection	Agent is asked politely to ignore secrets; files remain readable	**Filesystem regex deny interceptor hard-blocks all `.env*`, `~/.ssh/` access**
Subagent Tool Scoping	Monolithic context where every subagent gets full bash/write access	**Strict Role-Based Access Control (RBAC); research agents have zero shell tools**

SECURITY DIMENSION	CONVENTIONAL "PROMPT-GUARDED" AGENTS	KAROL'S ZERO-TRUST ARCHITECTURE
Supply-Chain Upgrades	Blind auto-pulls and untracked git submodules from external repos	**Radar v2: Automated AST delta briefs, immutable staging, and human sign-off**

Four Concentric Rings of Defense: End-to-End Operational Pipeline

RING 01: INGESTION GATE

Static AST & Encoding Gate

Pre-commit token checks, mojibake audits, hardcoded URL scrapers, and frontmatter schema linters. Unapproved tools never enter the library.

RING 02: CANONICAL VAULT

Harness-Neutral Packaging

Zero vendor lock-in. Single source of truth (`SKILL.md` + `_arsenal.yml` sidecar) rendered deterministically for Claude Code, DSH, and Antigravity.

RING 03: RUNTIME FIREWALL

PreToolUse Intercept Hooks

Native OS hooks intercept tool calls out-of-band: `security-gate.ps1` kills destructive shell commands; `secrets-guard.ps1` guards keys.

RING 04: PROVENANCE RADAR

Radar v2 Upstream Drift

Monitors 25+ upstream GitHub repos, flags unauthorized changes via AST delta briefs, and prevents unvetted code from polluting production systems.

02 // INGESTION & STATIC LINTER GATE

Auditing 11,000+ Files: Static Linters & The Rejection Taxonomy

Before any skill, agent, or MCP server is allowed into the production library, it must survive a multi-stage static analysis pipeline. We scan for unpinned dependencies, suspicious dynamic evals, hardcoded URLs, and broken mojibake encoding. Out of 11,000+ candidate files audited, 5,363 were permanently rejected. Here is the automated taxonomy that caught them.

THREAT CLASSIFICATION	REJECTION SHARE	AUTOMATED TRIGGER CONDITION	DEFECT RATIONALE & EXPLOIT VECTOR
Hardcoded Runtime URLs	`41% (2,198 files)`	`WebFetch, curl, wget, Invoke-WebRequest to hardcoded domains`	Deferred attack vector: External domain owner can redirect endpoints post-install to exfiltrate private corporate data or inject rogue instructions.
Arbitrary Dynamic Execution	`28% (1,501 files)`	`eval(), Invoke-Expression, $(...), backtick command interpolation`	Remote code execution: Constructs and executes arbitrary shell commands from unvalidated model strings at runtime.
Secret & Credential Probing	`16% (858 files)`	`Explicit regex matches for .env, ~/.ssh/, ~/.aws/, ~/.claude.json`	Credential harvesting: Ingests infrastructure secrets and transmits them via prompt context or outbound webhooks.
Token Bloat & Toy Prompts	`11% (590 files)`	`Unconstrained prose >4000 tokens, conversational banter, zero tests`	Operational degradation: Dilutes attention context, triggers model hallucinations, and burns unnecessary inference tokens with zero automated tests.
Mojibake & Encoding Defects	`4% (216 files)`	`Malformed UTF-8 bytes, double-encoded cp1252 byte sequences`	Syntax corruption: Mangled quotation marks and dashes corrupt shell arguments and parse trees silently during execution.

STATIC LINTER GATE

Deterministic Harness-Neutrality & Mojibake Scanner

Production PowerShell linter executed before any candidate skill or agent is staged into the vault. Hard-fails on encoding corruption or harness-specific leaks.

01 Deterministic AST & Mojibake Ingestion Gate

Execution Directive: Scans candidate packages for harness-specific token leaks and double-encoded cp1252 mojibake artifacts. Immediately halts ingest pipeline with exit code 1 if defects are found.

</> scripts/audit-arsenal.ps1 (Harness-Neutrality & Mojibake Gate)

```
# Extract from scripts/audit-arsenal.ps1 (Universal Harness-
Neutrality Gate & Mojibake Check)
$Script:CcTokens =
@('Read','Edit','NotebookEdit','Write','Glob','Grep','Bash','Agent','Tas
$Script:AgTokens =
@('read_file','edit_file','write_file','run_command','execute_script','s

# 1. Harness Neutrality Linter: Artifacts must never leak vendor-
specific tokens
function Test-HarnessNeutrality {
    param([string]$FilePath, [string]$Content)
    foreach ($token in ($Script:CcTokens + $Script:AgTokens)) {
        if ($Content -match "\b$token\b" -and $FilePath -notmatch "
(_arsenal\.yml|references/)") {
            Write-Error "VIOLATION: Vendor token '$token' leaked
into canonical artifact: $FilePath"
            return $false
        }
    }
    return $true
}

# 2. Mojibake Gate: Catches double-encoded cp1252 byte corruption
before commit
function Test-MojibakePurity {
    param([string]$FilePath, [string]$Content)
    # Detects common cp1252 byte misinterpretations (â€™, â€œ, â€,
Ã©)
    $mojibakePattern = "[\xC2-\xDF][\x80-\xBF]|Ã¢â‚¬|ÃƒÂ"
    if ($Content -match $mojibakePattern) {
        Write-Error "ENCODING DEFECT: Mojibake byte artifacts
detected in $FilePath"
        return $false
    }
    return $true
}

# Gate mode enforces exit 1 on any frontmatter error or encoding
defect
if (-not (Test-HarnessNeutrality $TargetFile $FileContent) -or -not
(Test-MojibakePurity $TargetFile $FileContent)) {
```

```
    exit 1
}
```

Toy Industry Prompts vs. Karol's Multi-Mode Autonomous Subsystems

The visual disparity between naive industry prompts and production engineering: a fragile 20-line single-file prompt versus an unabridged 46-file deterministic state machine with automated linters, language playbooks, and 11 specialist checklists.

NAIVE INDUSTRY STANDARD (TOY PROMPT)

Single 20-Line Markdown Prompt

`prompts/code-reviewer.md (Unstructured, Fragile)`

How Most Teams Do It: Drop a single loose 20-line markdown file into a prompt directory. Give the LLM full terminal access and ask it nicely to *"Review code and fix all bugs."*

</> prompts/code-reviewer.md (20 Lines)

```
# Generic Toy Skill (Single File: prompts/code-reviewer.md)
Role: You are an expert code reviewer.
Instructions:
- Read all files in the current repository.
- Identify all security vulnerabilities, performance issues,
  and bugs.
- Fix all issues directly in place using your tools.
- Ensure the code passes all tests.
- Commit and push your changes when done.
- If anything fails, try again until it succeeds.

# Why This Fails in Production:
# 1. Zero state tracking; models loop infinitely on syntax errors.
# 2. No regression tests; fixes silently break other modules.
# 3. Model self-policing; easily tricked by malicious comments.
# 4. Unbounded bash writes wipe uncommitted developer files.
```

- Zero automated regression tests or syntax validation
- No error-handling loop; silent hallucinations on syntax errors
- No persistent audit trail; changes are untracked and ephemeral
- Model self-policing; easily tricked by adversarial prompt injection

KAROL'S PRODUCTION STANDARD (ENGINEERED SUBSYSTEM)

Multi-Mode Subsystem Engine (46 Files • 13-Stage State Machine)

review-fix-technical-code/ (Unabridged 46-File Production Hierarchy)

How Karol Builds It: An unabridged 46-file orchestrated engine pairing a 13-stage deterministic state machine with isolated subagents, test suites, and audit logs. Every file serves an explicit operational purpose across root analysis, language patterns, reproducible test replays, and 11 domain specialist review checklists.

</> review-fix-technical-code/ (46-File Production Tree)

```
review-fix-technical-code/ (Unabridged 46-File Production Subsystem Tree)
├── SKILL.md                              # 634-line core engine: 6
operational modes & state machine orchestrator
├── _arsenal.yml                          # Package metadata, semver,
provenance & multi-harness target mappings
├── changelog.md                          # Immutable audit ledger of
subsystem modifications across releases
├── hitl-loop.template.ps1                # Human-in-the-loop checkpoint
runner (PowerShell / Windows environment)
├── hitl-loop.template.sh                 # Human-in-the-loop checkpoint
runner (POSIX Bash / Unix fallback)
├── analysis/                             # Deterministic Diagnostics &
Fault Isolation Engines (2 files):
│   ├── root-cause.md                     # 5-Whys systematic failure
hypothesis builder and diagnostic matrix
│   └── stack-trace.md                    # Exception frame demangling &
source line correlation analyzer
├── patterns/                             # Technology-Specific Audit &
Fix Playbooks (9 files):
│   ├── database.md                       # Connection pooling,
transaction boundaries & query deadlock recovery
│   ├── docker.md                         # Multi-stage cache
invalidation, signal traps (SIGTERM) & hygiene
│   ├── git.md                            # Merge conflicts, rebase branch
drift & detached HEAD state recovery
│   ├── go.md                             # Goroutine leaks, channel
deadlocks & memory profiling playbooks
```

- 13-stage deterministic state machine enforcing two-round independent fix verifications

- AST-aware linters and static analysis gates verifying code integrity before git staging
- Out-of-band PreToolUse hooks preventing accidental branch deletion or force-push
- Persistent `evidence.md` and immutable test logs verifying every code fix

03 // RUNTIME EXECUTION FIREWALL

PreToolUse Lifecycle Interception: Killing Dangerous Commands Out-of-Band

The most critical layer in the zero-trust architecture operates at runtime. Whenever an agent decides to run a shell command, read a file, or spawn a subagent, the harness triggers a PreToolUse hook. The hook runs out-of-band in the native operating system shell. If the hook returns exit code 2 or a deny decision, execution is immediately aborted before the shell ever touches the filesystem.

RUNTIME HOOK 01 // CROSS-HARNESS INTERCEPTION

Multi-Dialect Decision Adapter (Universal Harness Protocol)

Emits structured decision payloads compatible with Google Antigravity, Claude Code, and DeepSeek Harness formats simultaneously.

01

Multi-Dialect Output Adapter

Execution Directive: Formats intercept decisions into all three major vendor schemas simultaneously. Enables a single PowerShell script to govern Claude Code, Antigravity, and DeepSeek Harness without code duplication.

</> library/hooks/secrets-guard/secrets-guard.ps1 (Write-Decision Function)

```
# Multi-Dialect Hook Output Adapter Function (Production Code:
library/hooks/secrets-guard/secrets-guard.ps1)
function Write-Decision {
    param([string]$Decision, [string]$Reason)
    $hookEventName = if ($payload.hook_event_name) {
$payload.hook_event_name } else { "PreToolUse" }
    @{
        # Dialect 1: Google Antigravity Format
        decision = $Decision
        reason = $Reason
        # Dialect 2: Claude Code Standard Format
        permissionDecision = $Decision
        permissionDecisionReason = $Reason
        # Dialect 3: DeepSeek Harness & Claude Code Nested Spec
        hookSpecificOutput = @{
            hookEventName = $hookEventName
            permissionDecision = $Decision
            permissionDecisionReason = $Reason
        }
    } | ConvertTo-Json -Compress
}
```

RUNTIME HOOK 02 // DESTRUCTIVE EXECUTION FIREWALL

The Execution Gate (Dual-OS Regex Denylist)

Inspects shell command payloads before execution. Hard-blocks destructive filesystem operations, unconstrained database drops, and forced git branch overwrites.

02

Dual-OS Destructive Command Interceptor

Execution Directive: Matches proposed shell execution strings against a rigid cross-platform regex denylist. Emits a hard deny and exits with code 2 if any forbidden destructive pattern is detected.

</> library/hooks/security-gate/security-gate.ps1 (Dual-OS Regex Denylist)

```
# Production Code: library/hooks/security-gate/security-gate.ps1
(Dual-OS Denylist)
$patterns = @(
    # Unix & Cross-Platform Destructive Patterns
    'rm -rf', 'rm -fr', 'sudo rm', 'chmod -R 777', 'mkfs', 'dd if=',
':\(\)\{:\|:&\};:',

    # Windows PowerShell Destructive Patterns
    'Remove-Item\s+(-Recurse├─Force)', 'ri\s+.*(-r├─fo)',
'rd\s+/s\s+/q', 'del\s+/s\s+/q',
    'Clear-Disk', 'Format-Volume',

    # Git Destructive History & Branch Overwrites
    'git push .*(\+|--force├─f\b|--force-with-lease)',
    'git reset --hard', 'git clean -fd', 'git clean -f', 'git branch
-D',
    'git checkout \.', 'git restore \.',

    # Database Unconstrained Deletion Patterns
    'DROP TABLE', 'DROP DATABASE', 'TRUNCATE TABLE', 'DELETE FROM .*
WHERE 1=1'
)

foreach ($pat in $patterns) {
    if ($cmd -match "(?i)$pat") {
        $msg = "BLOCKED by security-gate: destructive pattern
detected ($pat)."
        [Console]::Error.WriteLine($msg)
        [Console]::Error.WriteLine("   If this is intentional, run
it manually in your terminal.")

        Write-Decision -Decision "deny" -Reason "$msg Command: $cmd"

        if (-not $isAntigravity) { exit 2 }
        exit 0
    }
}
```

RUNTIME HOOK 03 // CREDENTIAL BOUNDARY

The Secret Firewall (Tiered Credential Protection)

Prevents autonomous agents from reading or modifying environment files, SSH keys, cloud provider tokens, or harness configuration databases.

03 Filesystem Credential Interceptor

Execution Directive: Intercepts file read and write operations. Grants access to harmless documentation templates (.env.example) while hard-blocking all live credential paths.

</> library/hooks/secrets-guard/secrets-guard.ps1 (Tier 1 Enforcements)

```
# Production Code: library/hooks/secrets-guard/secrets-guard.ps1
$envExclusions = @(".env.example", ".env.sample", ".env.template")

# Safe template files are permitted for developer onboarding
if (-not ($envExclusions -contains $basenameFull)) {
    # Tier 1 Hard Blocks: Live credentials and cloud secrets
    if ($basenameFull -match '^\.env(\..+)?$' -or
        $filePath -match
'(\.ssh[/\]|\.aws[/\]|\.claude\.json$|id_rsa|id_ed25519)') {

        $msg = "BLOCKED by secrets-guard: Sensitive credential path
detected ($filePath)."
        [Console]::Error.WriteLine($msg)
        [Console]::Error.WriteLine("   Never expose infrastructure
secrets to LLM context.")

        Write-Decision -Decision "deny" -Reason "$msg Path:
$filePath"

        if (-not $isAntigravity) { exit 2 }
        exit 0
    }
}
```

RUNTIME HOOK 04 // LEAST-PRIVILEGE FILTER

Smart Permission Filter (Auto-Approving Safe Reads)

Classifies tool operations into read-only commands (instantly approved for maximum agent velocity) and mutation commands (gated for human confirmation).

04 Read/Write Tool Classifier

Execution Directive: Auto-approves safe filesystem inspection tools across all harnesses while preventing unapproved background file mutations.

</> library/hooks/smart-permission-filter/smart-permission-filter.ps1

```
# Production Code: library/hooks/smart-permission-filter/smart-
permission-filter.ps1
$readonlyTools = @('Read', 'Glob', 'Grep', 'LS', 'read_file',
'list_dir', 'search_files', 'view_file', 'search_web')

# Safe read-only operations receive instantaneous out-of-band
approval
if ($readonlyTools -contains $toolName) {
    Write-Decision -Decision "allow" -Reason "Read-only exploration
tool pre-approved"
    exit 0
}

# Any tool attempting disk mutations falls back to human oversight
gate
Write-Decision -Decision "ask" -Reason "Write/Execution operation
requires human confirmation"
exit 0
```

04 // SUBAGENT BOUNDARY MATRIX

Principle of Least Privilege: Confining Autonomous Subagents

In a multi-agent system, giving every agent the full toolkit is an architectural anti-pattern. A codebase researcher has zero business writing to disk or running shell commands. An implementation subagent should be confined to its assigned branch and barred from pushing code or deploying. We enforce a strict Role-Based Access Control (RBAC) matrix that defines exactly what tools each agent is permitted to touch.

SUBAGENT ROLE	PERMITTED TOOL DOMAIN	HARD-STRIPPED / FORBIDDEN TOOLS	WORKSPACE BOUNDARY & CONFINEMENT STANDARD
researcher	`Read-Only`	`write_file, edit_file, run_command, Bash`	Triage sandbox in `research/`; evaluates external candidate repos without write privileges to production `library/`.
librarian	`Vault-Ops`	`WebSearch, WebFetch, external network endpoints`	Strictly local vault operations (`library/`, `index.md`, `log.md`); physically blocked from outbound HTTP calls.
code-builder	`Task-Scoped Write`	`git push, git commit, git merge, deploy scripts`	Confined to declared task files in isolated git branch; forbidden from staging, committing, or pushing to remote.
code-reviewer	`Read-Only QA`	`write_file, edit_file, sed, in-place file mutations`	Cold second-opinion reviewer; inspects git diffs and outputs structured findings without authority to touch source code.
security-gate	`Runtime Hook`	`Self-mutation, bypassing exit code 2 contract`	Out-of-band OS execution interceptor; evaluates shell string syntax before process spawning occurs.

05 // SUPPLY-CHAIN PROVENANCE & UPSTREAM RADAR

Radar v2: Automated Upstream Drift Detection & 3-Way Merge Reviews

Open-source AI tools move fast. When an upstream author updates a repository you rely on, pulling changes blindly is how backdoors and prompt injections enter production. Karol built Radar v2—an automated drift-monitoring engine that tracks 25+ upstream GitHub repositories, generates structured AST delta briefs, and requires an explicit human security verdict before code can ever be staged.

STAGE 01: DRIFT SCAN

Watch-Universe Delta Detection

Nightly cron runs `radar.ps1 scan` across all monitored upstream repos, computing cryptographic SHA-256 hashes against stored baselines.

STAGE 02: DELTA BRIEF

Librarian AST Inspection

Generates human-readable delta briefs categorizing upstream additions into new capabilities, behavioral modifications, or potential attack surfaces.

STAGE 03: HUMAN GATE

Strict Adjudication Verdict

The operator reviews the brief and assigns a permanent recorded verdict: `clean`, `skip`, `absorb`, or `ticket`.

STAGE 04: ATOMIC SWAP

Staging & Rollback Verification

Absorptions stage updates in isolated temporary directories. Full linter suites run before atomically updating the production library.

SUPPLY-CHAIN ENGINE CLI

Radar v2 Command & Verification Protocol

Production CLI commands for scanning upstream drift, inspecting delta briefs, recording security verdicts, and executing crash-safe atomic absorptions.

01

Radar v2 Upstream Drift Inspection Contract

Execution Directive: CLI workflow executed to detect, review, and absorb upstream repository changes with complete cryptographic verification and rollback safety.

</> scripts/radar.ps1 (Radar v2 Operational Contract)

```
# 1. Scan monitored upstream repository watch-universe for drift
.\scripts\radar.ps1 scan -Repo
"https://github.com/DietrichGebert/ponytail"

# 2. View generated delta brief and AST changes for a modified skill
.\scripts\radar.ps1 diff -Item "ponytail-review" -Verbose

# 3. Record human verdict following security review (clean | skip |
absorb | ticket)
.\scripts\radar.ps1 verdict -Item "ponytail-review" -Action "absorb"
-Rationale "Reviewed AST diff; no hardcoded URLs"

# 4. Verify staged content against static linters and test suites
.\scripts\radar.ps1 verify -Item "ponytail-review"

# 5. Finalize promotion, update changelog.md, and bump semver in
_arsenal.yml
.\scripts\radar.ps1 promote -Item "ponytail-review"
```

06 // EMPIRICAL VETTING REGISTRY & ROI

Forensic Case Studies & Executive Risk Reduction Scorecard

Security isn't hypothetical; it's proven in the trenches. Over months of auditing candidate packages, our static linters and PreToolUse hooks caught real vulnerabilities in highly popular open-source packages before they could touch our developer environments.

CAUGHT IN INGESTION // TELEMETRY EXFILTRATION

Anthropic Common Room Plugin

`Hardcoded Outbound Tracking Beacon`

Defect Caught: External plugin contained unvetted background webhooks phoning home on every model invocation, transmitting user query metadata to unauthorized telemetry servers.

Zero-Trust Action: Hard-blocked at Static Linter Gate (Ring 01). Telemetry code was permanently purged before the tool was approved.

CAUGHT IN INGESTION // REMOTE CODE EXECUTION

Community Web Scraper MCP Server

`Unsanitized eval() on Scraped HTML`

Defect Caught: Third-party scraper parsed webpage scripts using `eval()`, creating an immediate indirect prompt-injection exploit allowing malicious websites to run arbitrary local shell commands.

Zero-Trust Action: Permanently rejected into `rejected/` with forensic rationale recorded.

Executive ROI & Enterprise Risk Reduction Scorecard

OPERATIONAL RISK CATEGORY	UNCONSTRAINED AGENT INSTALLATION	KAROL'S ZERO-TRUST CONTROL PLANE
Engineering Data & Secrets	High Risk (Unvetted `.env` & SSH key access)	**Zero Exposure (Regex denylists hard-block access)**
System Integrity	Critical Risk (Unrestricted destructive shell commands)	**100% Protected (Dual-OS denylists kill destructive syntax)**
Supply-Chain Upgrades	Unmonitored (Silent background updates introduce vulnerabilities)	**Cryptographically Audited (Radar v2 AST delta briefs & human gate)**
Audit Trail & Compliance	None (Ephemeral conversational sessions)	**Immutable (50,787 vault files versioned in Git with changelogs)**

The Security Retrospective: 3 Rules We Learned the Hard Way

RULE 01

Never Let a Model Police Itself

System prompts are advisory; operating system hooks are mandatory. Never trust an LLM to follow "security guidelines" when a single line of out-of-band PowerShell regex can prove compliance deterministically.

RULE 02

Strip Tools to the Absolute Minimum

If an agent is meant to research code or read documentation, don't give it bash tools. If an agent is meant to edit files, don't give it permission to push to production. Least privilege prevents 99% of accidents.

RULE 03

Never Trust Upstream Without a Diff

Open-source AI packages change overnight. Blindly running `git pull` on external agent repositories is an open invitation for supply-chain attacks. Always inspect AST delta briefs before promoting code.

© 2026 Karol Jasztal • Hosted on GitHub Pages • All Rights Reserved