BlastShield

Kernel-Level Protection for AI Coding Agents
Against Destructive Cloud CLI Commands

Cedar Labs

April 2026

Abstract

AI coding agents now operate with near-unrestricted access to developer machines, including the ability to execute destructive cloud infrastructure commands. Existing sandboxing solutions — both agent-built-in and OS-level — fail to address this threat model: they protect files and secrets but cannot distinguish terraform destroy from terraform plan. This whitepaper presents BlastShield, a two-layer defense architecture for macOS that combines kernel-level filesystem sandboxing (via Apple Seatbelt) with command-argument filtering (via biometric-authenticated PATH wrappers). We analyze why existing approaches are insufficient, describe the unique design decisions behind BlastShield, catalog remaining vulnerabilities, and identify areas for further research and empirical verification.

Keywords: AI agent safety, cloud infrastructure protection, macOS sandbox-exec, Seatbelt, SBPL, command-argument filtering, defense in depth, Terraform, Kubernetes, AWS, GCP, Azure

1. The Problem

AI coding agents — Claude Code, Codex, OpenCode, Cursor, and their peers — now operate with near-unrestricted access to developer machines. They read files, execute shell commands, and invoke cloud CLIs. The default deployment model trusts the agent entirely. When an agent runs with `--dangerously-skip-permissions`, `--full-auto`, or equivalent unrestricted modes, it can destroy production infrastructure in seconds.

The threat is not theoretical. A single `terraform destroy -auto-approve`, `gcloud compute instances delete`, `aws s3 rb --force`, or `kubectl delete namespace` issued by an autonomous agent — whether through misinterpretation, prompt injection, or hallucination — can cause irreversible damage. The blast radius is asymmetric: hours of human review versus seconds of automated execution.

2. Why Existing Solutions Don't Work

2.1 Agent Built-in Sandboxes

Most AI coding agents include their own sandbox or approval mechanism. Claude Code has `/sandbox` and confirmation prompts. Codex has approval policies. These operate at the **tool level** — they gate the agent's own tools and network access.

Why they fail:

1. **Shell escape.** An agent that executes Bash or Python can run any subprocess. The agent's sandbox only sees its own tool calls, not the commands spawned by those tools. `bash -c "terraform destroy -auto-approve"` is invisible to the agent's permission system.
2. **Opt-out culture.** The industry standard is `--dangerously-skip-permissions`. Developers disable safety mechanisms because they slow down workflows. Any protection that can be bypassed with a single flag will be bypassed.
3. **Model-dependent.** Agent sandboxes rely on the LLM to follow instructions about what it should and shouldn't do. Prompt injection, jailbreaks, and simple hallucinations can cause the model to ignore its own constraints. Security that depends on a language model behaving correctly is not security.

2.2 Existing macOS Sandbox Tools

Three open-source projects address AI agent sandboxing on macOS:

Project	Approach	Focus
sandvault	Separate macOS user account + sandbox-exec	User isolation, file/secrets protection
agent-safehouse	Composable sandbox-exec profiles, Homebrew	Filesystem policy, dotfile protection

Project	Approach	Focus
agent-seatbelt	Two-file minimal sandbox-exec wrapper	File/secrets protection

Table 1: Existing macOS AI agent sandboxing tools.

What they all share: a focus on protecting secrets and dotfiles. They prevent the agent from reading SSH keys, browser data, and shell history. This is valuable, but it addresses the wrong threat model for cloud infrastructure.

Why they're insufficient:

1. **No cloud CLI awareness.** None of them distinguish *terraform plan* from *terraform destroy*. The sandbox either allows the *terraform* binary entirely or blocks it entirely. There's no concept of destructive versus non-destructive subcommands.
2. **No credential-path blocking.** Protecting `~/.ssh/` and `~/.bashrc` is necessary but not sufficient. Cloud CLIs authenticate through many paths — AWS credentials, Azure config, GCloud config, application-default credentials, SSO token caches, MSAL caches, Keychain entries. Existing tools don't systematically enumerate and block these credential paths for each cloud provider.
3. **No state file protection.** Terraform state (*.tfstate*), Helm chart locks, CDK/SAM state — these are the objects that destructive operations mutate. Existing sandboxes don't protect them.
4. **No command-argument filtering.** `sandbox-exec` operates at the file/process level. It cannot see command arguments. No existing tool addresses this gap with a second layer that filters by subcommand.

2.3 Container-Based Isolation

Docker, Podman, and VM-based approaches provide strong isolation but at significant cost:

1. **Environment mismatch.** The agent operates in a container that doesn't match the developer's actual environment — different tools, different paths, different credentials. This defeats the purpose of an agent that's supposed to work *with* your infrastructure.
2. **Credential passthrough.** To be useful, the container must receive cloud credentials. Mounting `~/.aws/` or passing `AWS_ACCESS_KEY_ID` gives the agent the same destructive capability, just in a different process namespace.
3. **Workflow friction.** Running an agent inside a container adds setup complexity, volume management, and networking overhead that most developers won't tolerate for daily use.

3. The BlastShield Approach

BlastShield introduces a two-layer defense architecture specifically designed for the cloud CLI destruction threat model.

3.1 Layer 1: Kernel-Level Sandboxing (Hard Boundary)

BlastShield uses macOS `sandbox-exec` (Apple Seatbelt) with carefully crafted SBPL profiles that:

- **Block credential paths by provider.** Each cloud provider profile enumerates the specific credential, token cache, and authentication file paths for that provider. The agent process physically cannot read these files — the kernel enforces this regardless of what the agent tries.
- **Protect state files.** Terraform state, Helm chart locks, and backend configurations are write-protected. Even if the agent somehow authenticates, it cannot modify state.
- **Compose by intersection.** Profiles combine by intersecting their deny rules. Loading more profiles can only make the sandbox *more restrictive*, never less. This is a critical safety property — there's no accidental loosening.
- **Auto-detect from project.** BlastShield scans the project directory for indicator files (`*.tf`, `Chart.yaml`, `cdk.json`, etc.) and automatically loads the appropriate profiles. Zero configuration for the common case.

3.2 Layer 2: Command-Argument Filtering (Speed Bump)

Since `sandbox-exec` cannot distinguish *terraform destroy* from *terraform plan*, BlastShield adds a second layer:

- **PATH wrappers.** `blastshield-guard install` creates wrapper scripts that intercept cloud CLI invocations before they reach the real binary.
- **Read-only by default.** Each wrapper classifies subcommands as read-only or mutating. Read-only commands (`plan`, `list`, `describe`, `get`) pass through immediately. Everything else requires authentication.
- **Biometric/password gate.** Mutating commands require `sudo` authentication, which on MacBooks triggers Touch ID or a password prompt. The agent cannot satisfy this — it requires a human.
- **Fresh auth each time.** `sudo -k` invalidates the timestamp before each check, ensuring that a previous successful authentication doesn't carry over.

3.3 Why This Combination Matters

Neither layer alone is sufficient. Table 2 illustrates the complementarity.

Scenario	Layer 1 (sandbox)	Layer 2 (guard)
Agent reads ~/.aws/credentials directly	■ Blocked	■ Not visible
Agent runs terraform destroy	■ Same binary as plan	■ Intercepted
Agent uses full path to terraform destroy	■ Same binary	■ Bypassed
Agent has credentials in env vars	■ Not file-based	■ Intercepted

Table 2: Layer complementarity across attack scenarios.

4. Unique Design Decisions

4.1 Cloud CLI Focus, Not File Focus

BlastShield inverts the existing tooling priority. Instead of "protect all files, hope the agent doesn't run destructive commands," it asks: "what specific operations would cause catastrophic damage, and how do we prevent those?" The credential paths blocked are those that specifically enable cloud CLI authentication. The commands guarded are those that specifically destroy infrastructure.

4.2 Profile Intersection, Not Union

When multiple profiles are loaded, their deny rules intersect. This means `blastshield -p terraform -p aws` is at least as restrictive as either profile alone. There is no possibility of two profiles accidentally creating an allow rule that neither intended.

4.3 Human-in-the-Loop for Mutations

The guard layer enforces a principle: **the agent plans, you execute.** An agent can `terraform plan` and `aws describe-` all day. The moment it tries to mutate infrastructure, a human must be present. This aligns the security model with how infrastructure changes should work regardless of AI involvement.

4.4 Authentication as Authorization

Rather than building a custom authorization system, BlastShield uses `sudo` as its authentication gate. This leverages macOS's existing biometric infrastructure (Touch ID) and PAM configuration. No new accounts, no new tokens, no new attack surface. The user's existing macOS authentication is the authorization.

4.5 Defense in Depth by Default

The recommended setup includes both layers. But even Layer 1 alone provides meaningful protection — it blocks the credential reads that would enable destructive operations in the first place. The guard layer is the additional safety net for credentials that enter the process through other means (environment variables, credential helpers, Keychain).

5. Remaining Vulnerabilities

BlastShield does not claim to be a complete solution. The following vulnerabilities remain.

5.1 Credential Exfiltration via Network

Severity: High

The sandbox allows network access because agents need it (API calls, package downloads, git operations). If a credential enters the process through an allowed path — an environment variable without `-c`, a credential helper response, or a Keychain entry — the agent can exfiltrate it over the network.

Mitigation: Use `-c` / `--clean-env` to strip API keys from environment variables. For credential helpers, there is no current mitigation — this is a fundamental limitation of file-path-based sandboxing.

Research needed: Network-level filtering that distinguishes legitimate API calls from exfiltration. Possible approaches include egress proxies with allowlisted domains, DNS-level filtering, or `sandbox-exec`'s network rules (which would break most agent workflows).

5.2 Keychain Access

Severity: Medium

Keychain access is allowed so that credential helpers work (e.g., `aws` CLI using Keychain as its credential store). An agent can perform authenticated actions via credential helpers — `git push`, `aws s3 ls` — but cannot read raw tokens from files. The agent can *act* on your behalf but cannot *steal* your credentials.

Mitigation: None currently without breaking legitimate workflows. A future approach could restrict Keychain access to specific service entries rather than blanket allow.

Research needed: Fine-grained Keychain access control via `security` CLI or `sandbox-exec` Keychain rules.

5.3 PATH Wrapper Bypass

Severity: Medium

Layer 2 (guard) is explicitly a speed bump, not a hard boundary. A determined agent that specifies the full path to a CLI (`/usr/local/bin/terraform destroy`) bypasses PATH wrappers.

Mitigation: Layer 1 (sandbox) is the hard boundary. Even with the guard bypassed, the agent cannot read credentials. The guard catches accidental and casual misuse; Layer 1 catches everything else.

Research needed: Binary-level argument filtering that doesn't depend on PATH ordering. Possible approaches include `DYLD_INSERT_LIBRARIES` to intercept `exec*` calls, or macOS Endpoint Security Framework monitoring — though both have significant complexity and stability trade-offs.

5.4 Environment Variable Secrets

Severity: Medium

Without `-c`, secrets in environment variables (`AWS_ACCESS_KEY_ID`, `GOOGLE_APPLICATION_CREDENTIALS` content, etc.) are readable by the agent. The sandbox operates on file paths, not process memory.

Mitigation: Always use `-c` / `--clean-env`. Add `blastshield -c` to your agent launch commands.

Research needed: Automatic detection and stripping of cloud-related environment variables without requiring the `-c` flag.

5.5 sandbox-exec Deprecation

Severity: Long-term

Apple deprecated the `sandbox-exec` command-line interface in macOS 10.15 Catalina. It still works on macOS Sequoia (15.x), but Apple provides no guarantee of future support.

Mitigation: None currently. There is no Apple-provided replacement for ad-hoc CLI sandboxing.

Research needed: Monitor Apple's frameworks for a replacement API (possibly based on Endpoint Security Framework or improved entitlement-based sandboxing). Develop a bubblewrap/Firejail backend for Linux. Evaluate whether Endpoint Security Framework can replicate the file-path deny rules.

5.6 Nested Sandbox Limitation

Severity: Low

macOS does not support recursive `sandbox-exec`. If the agent process is already running inside a sandbox (e.g., a sandboxed app), BlastShield's Layer 1 cannot be applied.

Mitigation: Use Layer 2 (guard) alone in nested environments. Restructure setups to use a single `sandbox-exec` layer.

Research needed: Determine whether Endpoint Security Framework or `DYLD_INSERT_LIBRARIES` approach can provide equivalent file-path protection without conflicting with an existing sandbox.

5.7 Content-Level Filtering Impossibility

Severity: Low

`sandbox-exec` operates on file paths, not file contents. It cannot prevent an agent from reading a file that's allowed by path but contains secrets in its content. It also cannot prevent an agent from writing destructive content to an allowed file path.

Mitigation: This is a fundamental limitation of OS-level path-based sandboxing. There is no mitigation within the BlastShield model.

Research needed: Content-aware file access monitoring (possibly via Endpoint Security Framework or FSEvents), though this would add significant performance overhead.

5.8 Agent-Specific Prompt Injection

Severity: Unknown

A malicious input (file content, web page, API response) could cause the agent to attempt destructive operations. BlastShield mitigates the *execution* of such operations but cannot prevent the *attempt*.

Mitigation: Layers 1 and 2 together make execution unlikely but not impossible (see credential exfiltration and PATH bypass above).

Research needed: Integration with agent-level content filtering or guardrail systems that detect and reject prompt injection attempts before they reach the agent's reasoning.

Vulnerability	Severity
1 Credential Exfiltration via Network	High

Vulnerability	Severity
2 Keychain Access	Medium
3 PATH Wrapper Bypass	Medium
4 Environment Variable Secrets	Medium
5 sandbox-exec Deprecation	Long-term
6 Nested Sandbox Limitation	Low
7 Content-Level Filtering Impossibility	Low
8 Agent-Specific Prompt Injection	Unknown

Table 3: Summary of remaining vulnerabilities.

6. Areas for Further Research and Verification

6.1 Empirical Effectiveness Testing

- **Fuzz testing.** Generate random subcommand combinations for each CLI and verify that read-only commands pass and mutating commands are blocked.
- **Real-world audit.** Run actual agent workflows (Terraform deploy, Kubernetes management, AWS operations) inside BlastShield and verify that no destructive operation succeeds without authentication.
- **Regression testing.** Cloud CLIs add new subcommands in every release. The guard patterns need continuous validation against new CLI versions.

6.2 Cross-Platform Portability

- **Linux (bubblewrap/Firejail).** Namespace-based sandboxing can replicate most `sandbox-exec` capabilities. Profile translation from SBPL to bubblewrap arguments is feasible.
- **Windows (AppContainer).** Windows has its own sandboxing primitives (AppContainer, MIC) that could support a similar model.

6.3 Formal Verification of Profile Composition

- The profile intersection property ("loading more profiles can only make the sandbox more restrictive") should be formally verified. If two profiles contain conflicting allow rules, the intersection must still deny access. This requires a formal model of SBPL semantics and a proof that the composition operator is monotonic with respect to restriction.

6.4 Integration with CI/CD Pipelines

- AI agents increasingly run in CI/CD pipelines (GitHub Actions, GitLab CI) where the same destructive capabilities exist. Adapting BlastShield's concepts to container-based CI environments — where `sandbox-exec` is unavailable — would require a different enforcement mechanism (seccomp, AppArmor, or pipeline-level permissions).

6.5 Credential Helper Protocol Analysis

- The interaction between cloud CLIs and credential helpers (Keychain, `aws-sso-util`, `gcloud auth`, `az login`) needs deeper analysis. Understanding exactly which Keychain entries and IPC channels each helper uses would enable fine-grained allowlisting instead of the current blanket Keychain allow.

6.6 Agent Telemetry and Anomaly Detection

- Monitoring agent behavior inside the sandbox could provide early warning of attempted breaches: log sandbox violations (already supported via `--violations`), track unusual command patterns (e.g., an agent attempting multiple destructive commands in sequence), and alert on credential access attempts that were blocked. This telemetry could feed into a real-time risk scoring system that escalates to the user before a breach attempt succeeds.

7. Conclusion

BlastShield addresses a specific, high-severity gap in AI agent safety: preventing autonomous destruction of cloud infrastructure. It does this through a two-layer architecture — kernel-level file sandboxing plus command-argument filtering — that existing tools do not provide.

The approach is pragmatic: it uses macOS's built-in sandboxing primitives rather than building new infrastructure, it composes with existing tools rather than replacing them, and it acknowledges its own limitations rather than claiming false security.

The remaining vulnerabilities are real. Credential exfiltration via network, Keychain access, and `sandbox-exec` deprecation are open problems that require further research. But the current state — AI agents with unrestricted access to production cloud credentials — is strictly worse. BlastShield makes the threat model narrower and the blast radius smaller. That's worth something.

References

[1] Apple Inc. (2011). *Apple Sandbox Guide v1.0*. https://reverse.put.as/wp-content/uploads/2011/09/Apple-Sandbox-Guide-v1.0.pdf

[2] sandvault: https://github.com/webcoyote/sandvault

[3] agent-safehouse: https://github.com/eugene1g/agent-safehouse

[4] agent-seatbelt: https://github.com/CJHwong/agent-seatbelt

[5] bubblewrap: https://github.com/containers/bubblewrap

[6] BlastShield: https://github.com/cdrxyz/blastshield