Xiaotime Labs

POSITION PAPER · OFFENSIVE AI GOVERNANCE

The Governed Attacker

Why an autonomous AI red team is the hardest agent to govern, and what it takes to cage one you can audit.

For CISOs and security teams weighing what it takes to run autonomous AI agents, especially offensive ones, in an environment that has to pass a real audit.

The agent you should worry about most

Most of the worry about AI agents is abstract. An agent might go off task. It might touch something it should not. It might do something nobody asked for and nobody recorded.

There is one agent where that worry stops being abstract. An autonomous red team. An agent whose entire job is to find systems, reach into them, and prove they can be broken. Every fear people have about agentic AI is, in this one case, the literal job description. So it is the right place to test a claim. If you can govern an attacker, an agent built to do the thing everyone is afraid of, governing the rest of the fleet is the easy case.

We built one. We run it against our own platform. This paper is about what it takes to make that safe, and what it taught us when we turned it inward.

What makes it a cage

Start with a definition, because the rest depends on it. A *caged agent* is an autonomous agent that cannot act outside a boundary it has no way to reach around. Not an agent that is asked nicely to stay in bounds. An agent for which out of bounds is not a reachable state. For an offensive agent that boundary has to hold against the one actor specifically trying to break boundaries, which is the agent itself.

Every agent in the fleet runs on a *harness*: a fixed persona that sets what it is for, a fixed set of tools that sets what it can reach, a fixed memory that sets what it can recall, and a binding to one control point it cannot make a model call without crossing. A red team needs all of that and four boundaries more, because the cost of a scope failure is no longer a wasted call. It is a real action against a real system.

Four boundaries, no escape

01 · AUTHORIZATION

Comes first

No target is touched until there is a current, authorized engagement that names what is in scope, for how long, under what rules. Every action checks itself against that engagement before it runs. No engagement, no reconnaissance. The scope check is not a setting the agent consults. It is a gate the action passes through or does not happen.

02 · NO AUTHORSHIP

Orchestrates, never authors

The agent selects and runs vetted, existing tools. It has no path that takes something the model wrote, a payload, an exploit, a script, and executes it. The agent assembles a command from validated parts, and there is nothing that will run model-generated code as code. A model that cannot write its own weapon cannot improvise a new one under pressure.

03 · AVAILABILITY

Won't take a service down

Availability comes before the finding. Actions that degrade or deny a service are refused in the code, regardless of how the engagement is configured. Where a vulnerability can be confirmed, it is confirmed and stopped at proof, not exploited for effect. The point of the exercise is to know, not to break.

04 · RECORD

Contained and recorded

The agent runs inside an isolated private network with scoped reach, on a metered budget. Every action it takes is attributed to it, tagged to the MITRE ATT&CK technique it emulates, and written to an append-only, signed audit record. Credentials the agent must handle during a test are masked from that record, so the log stays shareable without leaking anything sensitive.

Authorization decides whether it acts. The no-author boundary decides what it can become. Availability-first decides what it will never do. The record decides what is knowable afterward. Together they are the difference between an autonomous attacker and a liability.

Why the agent cannot write its own weapon

Picture the version without it. An offensive agent that can write and run its own code is an agent that can invent a capability you never reviewed, in the middle of a run, against a live system. Its reach is no longer the set of tools you vetted. It is whatever the model can compose. You cannot scope what you cannot enumerate, and you cannot enumerate what the model has not written yet. That is an agent you hope behaves, not one you can prove is safe.

Now hold the boundary. The agent can reason about which vetted tool fits, in which order, against which in-scope target. It cannot manufacture a new tool. The set of things the red team can do is the set of things you reviewed, and that set does not grow at runtime.

This is the line between a red team you can put in front of an auditor and an autonomous attacker you are quietly hoping behaves. It is also what makes the audit record honest. Every action maps to a known technique because every action came from a known tool. The record is not a description the agent wrote about what it did. It is the byproduct of an action that could not happen any other way.

We pointed it at ourselves

Here is the part that taught us something. We turned the red team on our own production agents. Call it recursive integrity: the fleet that ships our software, reviewed by the fleet that attacks it, with no agent standing outside the test. An attacker that only ever runs against other people's systems is a product. An attacker you are willing to run against your own is a position.

THE FINDING

Agent integrity is structural, not a list of rules.

The intuitive fix for prompt injection and data exfiltration is to add a clause: tell the agent to treat external content as data, not instructions. We found that a clause does not hold. An agent's resistance to having its task hijacked comes from the structure of how it is built to reason, not from a longer list of prohibitions stapled to the front of its prompt. Strengthening the list did not close the gap. Restructuring how the agent operates did.

That is a design principle, and not an intuitive one: the safe-sounding fix, adding a clause to the prompt, is the one that fails. Resistance is structural. We learned it by attacking ourselves, on the record, and rebuilding to the finding. The same red team that found the weakness re-ran against the rebuild. Find, fix, re-test, all inside the same governed loop.

What you gain from a governed attacker

You can actually run one. An autonomous red team that cannot escape its scope, cannot author its own weapon, and cannot take a service down is one you can schedule, not one you have to fear. Most teams cannot run an offensive agent safely. The cage is why we can.

Every test is reproducible and audit-ready. Because each action is tagged and signed as it happens, a red-team run is not a report someone writes afterward. It is a record an auditor can query: what was tested, by what technique, against what scope, with what result.

Red-teaming becomes continuous, not annual. When the attacker is a governed agent in the pipeline rather than a once-a-year engagement, find-fix-re-test runs at the speed of the rest of the work. Findings route into the same governed remediation path as everything else.

None of these (safe execution, reproducibility, continuous testing) are features bolted on. They are what you get when the attacker is held to the same architecture as the rest of the fleet, or they are what you do without.

The objections, head-on

- *An autonomous offensive AI is reckless on its face.* It would be, ungoverned. The cage is the answer: it cannot act outside an authorized scope, cannot write its own weapon, and cannot degrade a service. The recklessness is in the ungoverned version, which is exactly the version this architecture refuses to ship.
- *Red-teaming your own agents is grading your own homework.* It would be, if the grade were a claim. It is not. The red team is itself a governed agent, and its actions, findings, and the fixes that followed are written to the same signed record as everything else. The homework is on the record, and so is the re-test.
- *Surely a strong enough instruction would stop prompt injection.* That is the intuition we tested directly, against ourselves. It does not hold. Resistance is structural. We would rather tell you that than sell you a clause.

About Xiaotime Labs

Xiaotime Labs builds the GRCDevSecOps platform: a governed fleet of AI agents that ships software and the evidence of its own governance in one motion. ICRG, Integrated Cyber Risk Governance, is the governance face of that fleet. The AI-SDLC pipeline is its execution face. The red team described here is one more agent in that fleet, held to the same control point as the rest. Xiaotime Labs runs the attacker on itself because the architecture leaves no other option.

Related reading

- **The Governed Fleet.** Why a fleet of AI agents is only as trustworthy as the control point they share. The substrate this paper turns toward the offensive case.
- **GRCDevSecOps.** Why governance belongs in the pipeline, not the report.
- **Building Upstream.** Why AI governance starts in the SDLC, not after deployment.
- **Closing the Loop.** The continuous monitoring runtime the industry described for fifteen years, finally built.

- **Everything as Code.** The velocity case for moving governance into the code layer.
- **Estate as Code.** Capturing the whole estate, including the AI, as code you can govern.
- **DoctorWhiskers Explains Everything as Code.** How AI agents author, operate, and govern the as-code stack.

Next steps

If you are running or buying autonomous AI agents, and especially if any of them can act offensively, carry one question into the evaluation. Not what can the agent do. What can it *not* do, and who would know if it did. If the platform in front of you cannot answer that about its own attacker, the architecture is the place to look.

Read this as an argument. If it holds up against your environment, we should talk.

CONTACT · info@xiaotimelabs.ai
LEARN MORE · xiaotimelabs.ai/icrg.html

A technical companion to this paper, covering the engagement-scope cage, the no-author boundary, and the agent-hardening structure, is available to prospective customers under NDA.