

Self-Modifying Runtime

A verifier-gated containment standard for adaptive AI agents, built and evaluated against the July 2026 Hugging Face intrusion

Apart Research — AI Incident Response Sprint. Track 1 (Containment), with material also relevant to Track 2 (verification checklist) and Track 5 (open track / defender's dilemma). Repository: github.com/dwan-ith/self-modifying-runtime

What we set out to answer

The prompt for Track 1 is a fair one to be uncomfortable with: what standard would have stopped this? The July 2026 Hugging Face intrusion did not happen because nobody had ever written down sandboxing advice. It happened because the one thing standing between an evaluation agent and the open internet was a package registry proxy with an undiscovered bug in it, and once that proxy gave way there was very little standing in the way of file reads turning into code execution turning into cluster admin. Our worry going into the sprint was that most containment proposals would be a checklist a lab could nod along to and then quietly fail to enforce anywhere. So instead of writing a document, we built a small runtime and made the standard the thing the runtime actually does. Every clause we propose corresponds to a piece of code you can run, and the reason we can put a number next to a claim like "identity checks stop credential spoofing" is that we measured it, on the same machine, with the same seeds, rather than just asserting it.

The core idea

Most of the interesting security failures in the incident were not failures of a single control. They were failures of layering: a compromised credential was trusted because nothing below the policy layer double checked who was actually issuing a command, a secret value moved into a less sensitive container and only got noticed afterwards, and a defensive change could in principle have quietly reopened a hole that had already been closed. So rather than modeling "a program that rewrites itself," which is old news, we built a system where the thing that gets rewritten is deliberately narrow: a security policy sitting on top of a fixed, non-negotiable substrate. The substrate handles identity, the information-flow rules, and the checks a proposed policy has to pass before it is signed and installed. The policy handles which commands are allowed, denied, transformed, or quarantined, and that part is free to evolve every round in response to whatever the attacker just tried.

The reason this split matters is that a self-improving defense is only trustworthy if there is some part of it that the defense cannot talk itself into weakening. If the policy could also rewrite the verifier, or rewrite what counts as a valid credential, then "the defense got better" and "the defense got backdoored" would look identical from the outside. Keeping identity, taint semantics, and verification fixed, while letting the policy adapt underneath them, gives you a concrete answer to the containment question: the standard is not "write better rules," it is "decide in advance what is not allowed to be a rule."

What was actually built

The system is an abstract machine holding data at three confidentiality levels, public, protected, and secret, where any derived value takes the highest classification of whatever it was computed from. An

attacker probes this machine through commands like read, write, copy, compare, and a small set of arithmetic transforms, and a defender watches what gets disclosed and proposes patches. Four attacker modes are implemented: a random baseline, a budgeted game tree search, an epsilon-greedy reinforcement learner, and an oracle attacker that deliberately tries to infer secret values from indirect signals rather than reading them outright, the same way the real intrusion pieced together access from small, individually unremarkable steps rather than one dramatic breach.

The defender runs a counterexample-guided loop. When a disclosure happens, it becomes a counterexample the defender has to explain away, and the defender proposes narrower or broader rules, tests them against everything it has seen so far plus a set of legitimate operations it must not break, and only then hands the candidate to the verifier. The verifier is the one component we kept deliberately boring: it checks well-formedness, enforces a complexity ceiling so the policy cannot sprawl into something unreadable, checks that legitimate use still works, refuses to let a rule grant the policy author itself new authority, and runs a differential probe pass to make sure the candidate does not silently reopen anything the current policy already denies. Only a policy that survives all of that gets an HMAC signature and gets installed.

Identity sits below all of this. A command can claim to come from a legitimate user, but the runtime checks a credential against a registry rather than trusting the label on the command, and if the two disagree the command is treated as coming from an attacker no matter what it says. This is a small design choice with a large consequence: without it, a policy that is otherwise evolving correctly could still be defeated by simply forging who sent a request, and no amount of clever rule synthesis upstream would matter.

How this maps to the actual incident

We did not want to claim more isomorphism than the abstraction can carry, so the mapping is stated plainly rather than implied. The default scenario in the repository is built around the roles that actually mattered in the Hugging Face timeline: a permitted egress path that turned out to have a flaw in it, a place for covert coordination to happen, credentials that got reused past their intended scope, and a chain that ran from a file read to code execution to broader infrastructure access in a matter of hours. Each of those gets a named cell in the abstract machine with a confidentiality level attached, and the kill-chain replay script walks through seven phases drawn from the public timeline, checking at each phase whether the unhardened starting policy leaks and whether a policy that has already been through verifier-gated CEGIS rounds contains it.

What we are not claiming is that this is a simulation of Kubernetes, real network topology, or an actual package registry vulnerability. It is a policy-level abstraction of the kill chain, useful for asking whether a given control would have mattered at a given phase, not a replacement for testing the same ideas against a real cluster. The follow-up section below is explicit about what porting this to a real sandbox would require.

What the numbers actually show

The strongest single result is also the simplest one to reproduce. We ran the same binary with a fixed set of seeds, once with identity verification on and once with it switched off, against an attacker that spoofs its claimed identity half the time. Everything else about the run, the scenario, the epochs, the attacker

mode, stayed the same.

Condition	Spoofed commands accepted	Runs that converged	Legitimate-use score
Identity checks on	0	100%	1.00
Identity checks off	234	0% (mostly unrecoverable)	1.00

One flag flip is the entire difference between zero spoofed commands getting through and two hundred and thirty four getting through, and between every run converging to a stable, contained policy and almost every run ending in a state we report honestly as unrecoverable rather than dressing up as still improving. Legitimate use does not suffer either way, which matters because a containment standard that only works by also blocking the people it is supposed to serve is not a standard anyone will adopt. We think this single comparison is a reasonable stand-in for the broader argument: identity has to sit underneath the layer that gets to adapt, or the adaptation is defending a fiction.

Beyond that headline comparison, the attack-phase matrix in the submission walks through each control individually, auth off, schema validation off, posture flipped, unsigned installs, oracle grading versus exact string matching, and cites the specific test or ablation that backs each row rather than asserting it. The kill-chain replay shows the unhardened policy leaking on five of the seven modeled phases and the signed, verifier-approved policy leaking on none of them, run on the same default scenario every other experiment uses rather than a separate, more favorable setup built just for the demo.

Where we think this is weaker than it sounds

We spent a chunk of the sprint having the project audited critically rather than just polishing the pitch, and a few things came back that are worth stating plainly rather than leaving for a judge to find first.

- The verifier's anti-rollback guarantee is currently a statement about the probes we chose to run, not a proof over the whole policy language. A policy could in principle pass every probe we test and still contain a narrow gap the probe set does not exercise. We call this "no weakening found," not "no weakening exists," and that distinction should stay visible in how the project is judged.
- The label-downgrade protection is currently a detector, not a preventer. When sensitive data gets copied into a less sensitive container, the write happens first and the system notices and flags it immediately afterward, rather than blocking the write before it lands. For a research testbed that is a fine place to start, but it means the current README language about raising a counterexample "before the value settles" is a little ahead of what the machine actually does, and we would rather flag that ourselves than have it found later.
- The hash-chained audit log detects tampering after the fact, which is useful, but it is not anchored to anything outside the process. Someone who can rewrite the whole log can also recompute a consistent chain for it. Calling it "cryptographically immutable" overstates what a self-contained hash chain gives you; "tamper evident" is the accurate phrase until there is an external checkpoint.
- The abstract machine is exactly that, abstract. There are no containers, no real network stack, and no timing channels. The isomorphism to the actual incident is argued cell by cell in the mapping table, not proven, and we think that is the honest way to present it rather than implying the harness is a drop-in replacement for testing a real sandbox.

Reproducing the results

Everything below runs offline, which matters for the sprint's own judging criterion: a reviewer should be able to check the claims without needing network access to our infrastructure or trusting a demo video.

```
pip install -r requirements.txt
python -m pytest tests/ -q    # expect 131 passed
python sprint_evidence.py    # regenerates the ablation table, about two minutes
python sprint_hf_replay.py    # seven-phase kill-chain replay, about ten seconds
python main.py sweep --modes random,oracle --seeds 1,7,42 --epochs 25 --rounds 6 --out
sweep.json
python main.py serve --attacker search --epochs 20 --seed 42 --rounds 5    # live
dashboard
```

A regulator or judge can also verify compliance without re-running anything, using the policy fingerprint chain, the HMAC signature on each installed policy, and the exported audit log's own verification routine. That was a deliberate design goal rather than an afterthought: a standard that can only be checked by trusting the team that built it is not much of a standard.

What a month of follow-up would add

The honest next step is not more attacker modes or a nicer dashboard. It is closing the gap between what the verifier currently checks by sampling and what a real formal guarantee would require, starting with the label-downgrade issue above, which should become a precondition on the write operation rather than a post-hoc alarm. In parallel, we would want to port the command set, read, write, copy, compare, transform, send, onto real primitives: eBPF for the execution side, egress proxy events for the network side, and Kubernetes audit logs for the credential and lateral-movement side, and then check whether the same ablation results hold against something closer to the real thing rather than the abstract machine. We would also want to run the refusal-safe forensic protocol described in the checklist annex against an actual graded benchmark, since the incident itself involved frontier models declining forensic work and the responding team falling back to a self-hosted, quantized open-weight model to do the analysis instead. That gap, what a lab actually does when its primary tooling refuses to help investigate an incident involving that same tooling, is not something our current harness models at all, and we think it deserves its own line of work rather than a footnote.

Where the incident details come from

Every specific claim about the incident used to build the default scenario is drawn from material either party published: Hugging Face's technical timeline from 27 July 2026, OpenAI's own account and technical report, and the joint disclosure the two organizations issued on 21 July. Where the two accounts agree, we treat the detail as established. Where only one side has said something and the underlying logs have not been released, we flag it as asserted rather than confirmed. Figures we did not verify ourselves, action counts, cluster counts, timestamps, are attributed to their source rather than restated as our own finding.

The short version: this is a working testbed, not a finished proof, and we think that is the right amount of confidence to have in a project built over a single sprint. What we would ask a judge to weigh is not whether the system is complete, but whether the underlying idea, keep identity and verification outside

the part of a defense that is allowed to adapt, is the right shape for a containment standard, and whether the harness makes that idea checkable rather than just arguable.