When the Agent Decides: Ten Design Failures in Autonomous Offensive Security Agents

Ricardo Martínez

Cloud Security & Applied AI

September 2026

Abstract

Autonomous AI agents for penetration testing promise to scale offensive work without scaling the human team. This piece documents ten field observations, drawn from operating one such system against real targets over several months, on why the reliability of these agents depends less on the capability of the model orchestrating them and more on concrete engineering decisions: what the LLM is allowed to decide, where the recall problem actually lives, when a failure is worth retrying, how a finding gets verified (and how the verifier itself gets verified) before it is believed, and how silently a system degrades when its environment, its liveness clock, or its redaction layers fail without making any noise. Ten figures and one comparison table grounded in real measurements are presented, closing with nine design principles applicable to any agentic system operating against real attack surface.

1 The Mirage of Total Autonomy

The first instinct when building a pentesting agent is to let the language model decide which tool to use at each step. It is also the first source of a coverage problem that stays invisible until someone actually measures it.

In a system with roughly sixty available attack capabilities (injection, SSRF, deserialization, secret leakage, access control, JWT, GraphQL), auditing how many ran with guaranteed certainty on every run produced an uncomfortable number: fewer than twenty. Everything else only ran if the agent, in that moment, decided it was worth calling. One concrete test confirmed it: a domain reconnaissance technique showed up in one run and vanished in the next against the exact same target, with nothing in the environment having changed.

The engineering answer was not to tune the prompt so the agent would "remember" to use more tools (Figure 1). It was to pull high-value, low-false-positive-risk capabilities out of the LLM's discretionary territory entirely and turn them into a deterministic sweep that always runs, in parallel, before the agent even enters the picture. The model is left for what genuinely requires judgment: chaining findings together, prioritizing ambiguous surface area, writing up context.

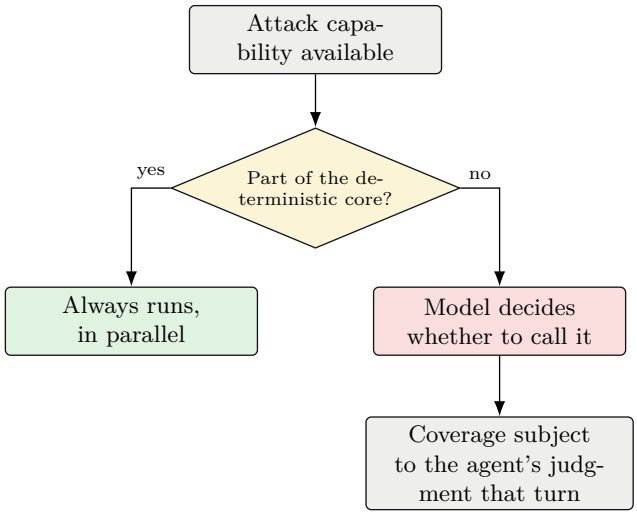


Figure 1: Coverage arbitration. Of ~60 available attack capabilities, fewer than 20 ran with guaranteed certainty on every run; the rest were subject to the agent's judgment call each turn.

Taken to its final form, the complete pipeline of a real engagement looks like Figure 2: a long chain of deterministic steps that always run, with a single discretionary hinge in the middle.

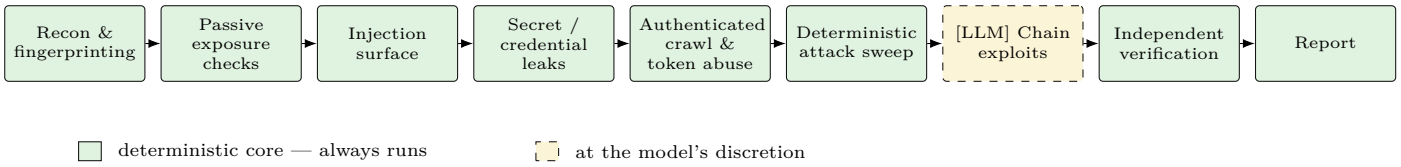


Figure 2: The actual shape of an end-to-end pentest pipeline. Of nine representative stages, eight are a deterministic sweep that always runs; only exploit chaining is left to the model's discretion, precisely because deciding *whether* to combine two findings is the kind of judgment worth delegating.

2 Recall Is a Discovery Problem Before It's a Detection Problem

When a scanner fails to find a real vulnerability, the instinct is to suspect the detector: the payload was wrong, the confirmation engine was too conservative. In practice, the more common cause sat one step earlier: the detector never saw the vulnerable endpoint, because the surface-discovery stage never put it on the table.

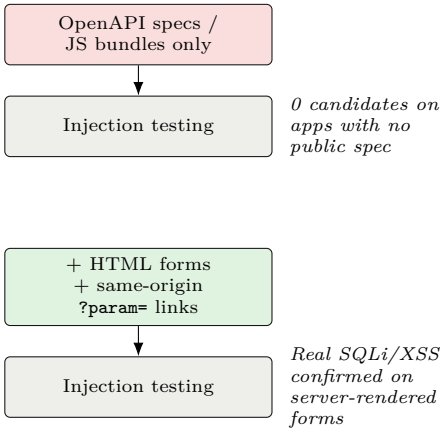


Figure 3: Widening surface discovery, with zero changes to the confirmation engine, moved recall from 0 to real confirmed findings. The test app had no OpenAPI spec at all: its parameters lived only in server-rendered forms.

Figure 3 documents the case: the exact same injection-confirmation engine, unchanged, went from having nothing to test to confirming real vulnerabilities the moment discovery started parsing forms and same-origin `?param=` links, on top of the OpenAPI specs and JS bundles it already covered. The detection engine was never the problem. Before tuning payloads or confirmation thresholds, it is worth confirming that the discovered surface actually includes where the vulnerability lives.

3 Not All Retries Are Equal

An adversarial code audit found the same fragility pattern repeated across more than thirty distinct call sites: a single network request, and any exception at all — a timeout, a connection reset, a transient 5xx — was treated as a definitive result. A network blip during discovery or confirmation turned into a silent miss, indistinguishable from a genuinely clean target.

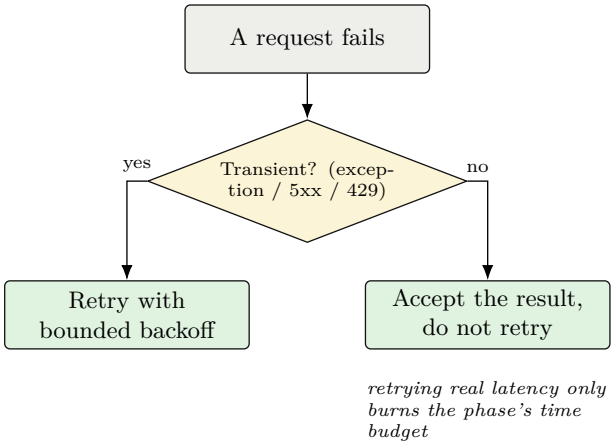


Figure 4: Bounded retry only on transient failure. Retrying a genuine timeout against a slow target does not improve recall, and can exhaust the phase's time budget — indiscriminate retry is as damaging as never retrying at all.

The fix (Figure 4) was not to retry everything, but to distinguish the type of failure: only a connection exception, a 5xx, or a 429 (rate limit) triggers a bounded retry with backoff; a definitive response — including a genuine timeout against a slow target — is accepted and never retried. An additional nuance surfaced while debugging a genuinely unstable test target under load: the exact same code passed and failed across different runs against the same target, with nothing changed in between. The lesson there was to separate two distinct questions before "fixing" anything: did the code regress, or is the test target simply unstable? Confusing the two leads to chasing bugs that do not exist.

4 Adversarial Verification, Not Optional

An agent that reports its own findings with no one else questioning them has a structural conflict of interest: it was trained to complete the task, not to doubt itself. The literature on LLMs as judges documents the same pattern from another angle — a model tends to rate an output more favorably when it recognizes it as its own [3], exactly the bias that adversarial verification, with no access to the original reasoning trace, is designed to remove.

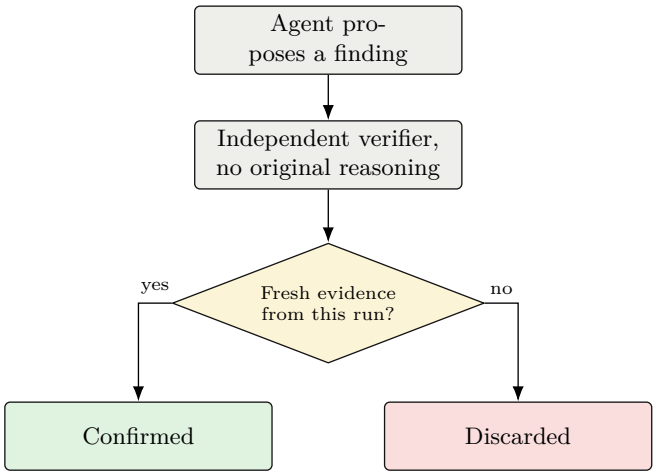


Figure 5: Verification pipeline. The verification step runs on a separate model instance, without the reasoning thread that produced the original finding.

In one run against a production target, the attack agent reported 7 critical or high severity findings in a single pass. The pipeline in Figure 5, run immediately after, discarded 6 of the 7 for lack of fresh evidence. The cause, found by reading the agent's exact prompt: an instruction told it "don't re-test what's already confirmed from previous runs." The agent cited historical findings as if they belonged to this run, without executing a single new tool against them.

The instruction was not wrong by carelessness, it was wrong because it optimized for the wrong thing: saving time by skipping re-confirmation of the obvious, without accounting for the fact that a language model does not distinguish between "this is already known" and "I can claim this as my own." The fix was not to ban historical context, but to change its function: from fact to hint.

The system, in this case, worked exactly as it should. The verification layer caught 86% of the noise. But a system that needs that layer to avoid lying by accident deserves the same structural distrust applied to any unverified source, even when it arrives dressed as a technical report.

5 Verifying the Verifier

The verification layer from the previous section solves one problem and creates a subtler new one: who audits whether the verifier itself keeps measuring something real? A concrete case exposed this. A test harness checked that every mandatory pipeline phase had run, comparing against a reference list maintained by hand inside the harness itself. As the real pipeline gained and lost phases across successive iterations, that reference list was not updated at the same pace. The result: the check kept reporting success, always, regardless of what actually ran — a tautological test that had stopped testing anything.

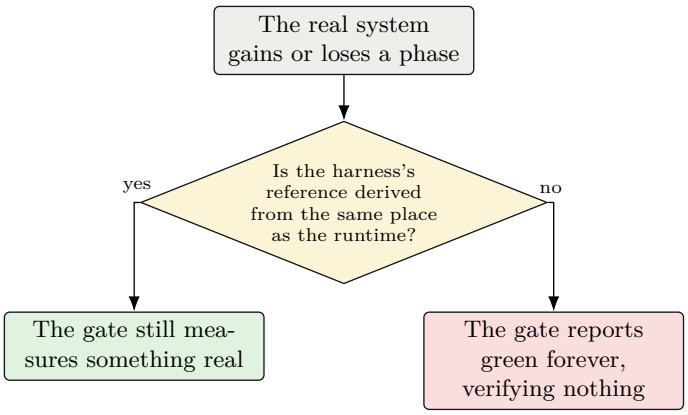


Figure 6: A verifier with its own hand-maintained copy of the truth, not derived from the same source as the real system, drifts silently. The gate never fails again — not because the system is perfect, but because the gate stopped comparing against anything real.

The fix (Figure 6) was to make the reference list derive from the same place the runtime pulls its phases from, instead of living as a hand-maintained parallel copy. Any verification layer — this section's or the previous one's — is itself code that can rot silently if its own notion of "truth" isn't tied to the same source as the system it watches.

6 An Evidence Hierarchy

Not every "confirmed finding" a security AI agent produces carries the same evidentiary weight. Ranking confirmation techniques by how hard they are to false-positive changes entirely how an AI-generated security report should be read.

When a system has an out-of-band network callback

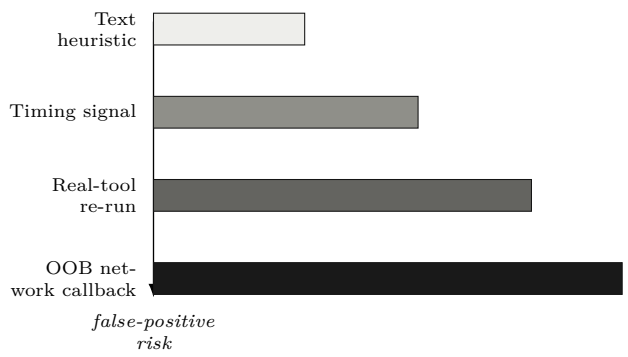


Figure 7: Confidence hierarchy for automated confirmation techniques. The out-of-band network callback, where the target itself makes an outbound request to a unique domain generated for that specific probe, is self-contained proof: it requires no interpretation from the model at all.

technique available (Figure 7), the correct design decision is to run it every time the scan mode allows, not to leave it as one option among dozens the agent may or may not choose to call.

7 The Model Isn't the Bottleneck

The natural assumption is that a bigger, more specialized, or "uncensored" model produces better offensive security results. Put to a controlled test, that assumption did not hold.

The lesson is not that specialized models are bad. It is that reliably orchestrating tools within a structured contract is a distinct skill from "knowing offensive security." Four local models were benchmarked under the exact same tool-calling contract used by the orchestrator; results are summarized in Table 1.

Neither offensive-specialized model made it as far as measuring recall: one invented its own output syntax instead of using the structured function calls the system expected; the other rejected the request at the architecture level, since its model family does not implement tool-calling at all. Both proved useful as free-text security Q&A or synthesis models, but neither could participate in the agentic loop.

8 Environment Fragility as a Silent Failure

Of all the possible failure modes, the most dangerous is not the model hallucinating. It is a real, correctly-built tool silently failing to run for a reason entirely unrelated to the system's logic, with nobody noticing because the system keeps producing a report that looks perfectly normal.

A deterministic injection engine failed to run for an entire testing session against a real target. The cause was not in its code: the orchestrating process had been started with an environment variable different from the reference environment, and that single difference made the operating system resolve the binary's name to the wrong executable sharing that name. The visible symptom was one generic error line buried among dozens of log entries, easy to mistake for "the target wasn't vulnerable" instead of "the tool never ran."

This kind of failure drops real coverage below reported coverage, consistently, for as long as the environmental condition persists. The correct response is not to document "remember to export the right variable"; that offloads the responsibility onto human memory. The correct response is for the system itself to guarantee its own environmental conditions when executing each tool, regardless of how it was launched.

The same silent-failure pattern shows up at the level of an entire engagement, not just a single tool. An active scan sat marked "running" for over an hour with nobody noticing: the target was resetting connections under a web application firewall, the phase's loop froze waiting for a response that never came, and the scan's own heartbeat mechanism froze right along with it — a process that stops breathing cannot report that it stopped breathing.

The lesson in Figure 9 is distinct from the tool with the broken PATH, even though both are silent failures: here, no mechanism *internal* to the task can detect its own freeze, by definition. It takes an external observer — a periodic sweep, unrelated to the loop being monitored — that measures liveness by elapsed time, not by the task's own claim that it's still alive.

9 Defense in Depth for Sensitive Data

A system investigating real attack surface inevitably touches data that should never reach a report in the clear: credentials, tokens, user listings. The design temptation is to redact that data at the point where

Model	Params	Tool-calling contract	Recall / usability	Runtime / verdict
Base	9B	Honored without exception	6/6 known vulnerabilities	21.5 min — reference
"Uncensored" fine-tune	9B	Honored without exception	6/6 known vulnerabilities	60.4 min (2.8×) + 1 new FP from mistaking a broken tool install for a real finding
Offensive-specialized A	7B	Ollama reports it as tool-capable, but it fabricates its own XML instead of filling `tool_calls`	Unusable as an agent	Complex prompt: 33+ min, never closed the task
Offensive-specialized B	8B	Rejected outright (completion-only architecture, no tool support)	Unusable as an agent	4.9s in pure Q&A mode — correctly cited the CVE queried, but never executes a tool

Table 1: Empirical comparison of four local models under the orchestrator's exact tool-calling contract. The two "offensive-specialized" models fail before recall can even be measured: one invents its own output format, the other doesn't declare tool support at all. Both are useful as security Q&A or synthesis models; neither works as an agent.

it's generated and call it solved. That temptation is exactly the mistake: a single redaction point is, by definition, a single point of failure.

The real case that forced this design (Figure 10): a new phase of the system generated evidence in a format the first redaction layer didn't know to recognize, and a listing of real usernames slipped through unmasked all the way to the persistence point. It never reached a reader because a second layer, at the persistence point — which redacts *everything* written, regardless of which phase produced it — intercepted it there. The property that matters is not that each layer is perfect; it's that no single layer is a requirement for the whole system to be safe.

10 A One-Time Validation Is Not a Boundary

Restricting what a scan is allowed to target (never cloud metadata, never private ranges without explicit authorization) sounds, at first glance, like a check performed once, when the request is accepted. That is precisely the surface an attacker — or a hostile target — can exploit.

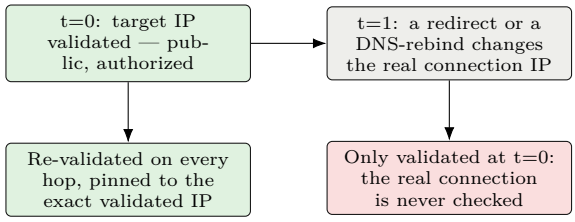
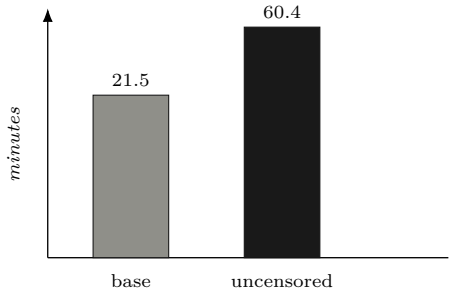
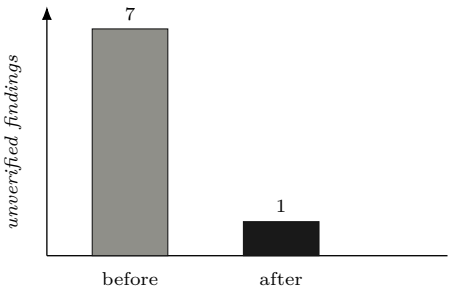


Figure 11: A front-door validation checks the IP at the moment the request is accepted (t=0); a redirect or a DNS-rebind can change what the system actually connects to afterward (t=1). Closing the window requires re-validating on every hop and pinning the connection to the exact IP that was validated.

The principle in Figure 11 is an instance of a well-documented vulnerability class, TOCTOU (*time-of-check to time-of-use*) [1]; the concrete DNS-rebind vector that exploits it was formally characterized nearly two decades ago [2] and remains effective against new systems today. The principle generalizes beyond network scanning: any security control expressed as "checked on the way in" leaves open the window between the check and the actual use. If the system later follows redirects, resolves names again, or reuses a decision made minutes earlier, that window is exploitable. The fix is not a stricter check at the start; it is repeating the same validation at every point where the system actually acts, and tying the action to the exact result of that re-validation.



(a) Same target, same recall



(b) Before / after the prompt fix

Figure 8: (a) Two variants of a local ~9B-parameter model against the same lab target: identical recall, but the "uncensored" variant took 2.8× longer and produced one additional false positive. (b) In the same system, fixing a single prompt instruction reduced from 7 to 1 the critical/high findings reaching verification without fresh evidence.

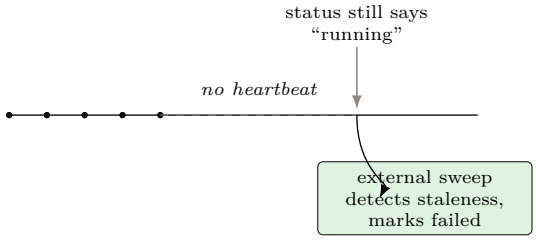


Figure 9: A process that freezes cannot report that it froze. The task's own heartbeat died along with the task; only an external liveness sweep, unrelated to the loop that failed, could detect and correct the stuck state.

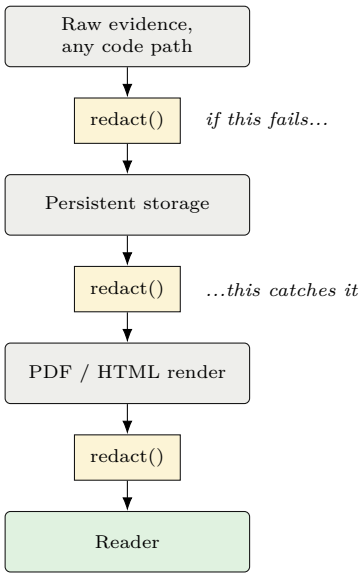


Figure 10: Three independent redaction layers, not a sequential trust chain. A new code path that skips the first layer still passes through the other two before reaching a reader.

11 Design Principles

1. **Critical capabilities are never optional for the model.** If something must always run, make it a deterministic pipeline step, not just another tool among many.
2. **Recall is won in discovery, not just in detection.** A perfect confirmation engine finds nothing if it's never shown the vulnerable endpoint.
3. **Retry the transient failure, accept the definitive one.** Retrying every error indiscriminately is as damaging to recall as never retrying at all.
4. **No finding is trusted in the same pass that produced it.** An independent verification step, without the original reasoning in view, is the difference between a report and a hallucination wearing a report's clothes.
5. **The verifier needs verifying too.** A test whose own notion of "truth" is hand-maintained, not tied to the real source, turns tautological in silence.
6. **Rank your evidence, and say so in the report.** A network-callback detection and a text-heuristic detection should not reach the reader under the same "confirmed" label.
7. **Historical context is a hint, never a fact.** If a prompt gives the model permission to skip re-verifying something, that something will eventually get reported without evidence.
8. **The runtime environment is part of the reliability surface.** A system that depends on its

environment being configured "the way it should be" has a fragile dependency disguised as a technical assumption; this includes being able to detect its own freeze, something no purely internal mechanism can do.

9. **No security control is a one-time validation.** Repeat the check at every point where the system actually acts, and use independent layers for sensitive data — never a single point of redaction or validation.

Methodological note: the observations above come from operating a single system in production across several work sessions, not from a controlled study across multiple systems. The figures cited are real measurements from those runs; target, client, and tool names are deliberately omitted for confidentiality.

References

[1] MITRE. *CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition.* cwe.mitre.org/data/definitions/367.html.

[2] Jackson, C., Barth, A., Bortz, A., Shao, W., Boneh, D. *Protecting Browsers from DNS Rebinding Attacks.* ACM Conference on Computer and Communications Security (CCS), 2007.

[3] Wataoka, K., Takahashi, T., Ri, R. *Self-Preference Bias in LLM-as-a-Judge.* arXiv:2410.21819, 2024.