RS REED SENTINEL

TECHNICAL WHITEPAPER · PENETRATION TESTING ENGINEERING

Designing a Safe Autonomous *Penetration-Testing Engine*

What it actually takes to let software run a by-the-book penetration test on its own — without ever handing a language model a shell.

METHODOLOGY	ARCHITECTURE	ISSUED
PTES	DETERMINISTIC CORE / LLM NARRATOR	2026

00 Abstract

"Autonomous penetration testing" usually means one of two things: a brittle script that runs a fixed list of commands, or a large language model handed a terminal and told to "hack the box." The first cannot reason about results; the second cannot be trusted with the keys. This paper describes a third architecture — one that has run end-to-end, by-the-book engagements against a deliberately vulnerable target and produced auditor-defensible reports — built on a single structural commitment: **the model never gets a shell.** A deterministic engine owns every decision and every command; the model is invited in only at the end, to explain results it had no power to cause. The result is a tool that is genuinely autonomous and genuinely safe at the same time, because those two properties are enforced by the architecture rather than by prompt-wording.

01 The Problem With "AI That Hacks"

The seductive design is obvious: give a capable model a Kali shell, a goal, and a loop. It works in demos. It fails as a product, for reasons that are not fixable by a better prompt:

No scope guarantee

A model with a shell can reach any host it can route to. "Only test 10.0.0.5" is a request, not a constraint. One hallucinated target or one mistyped CIDR is an out-of-scope intrusion — potentially a crime.

No determinism

The same engagement run twice produces different commands, different coverage, different findings. You cannot stand behind a report whose methodology changes every run.

No audit trail that means anything

When the decision-maker is a black box, "why did the tool do that?" has no answer. Clients, insurers and auditors require one.

Destructive blast radius

A shell is a shell. Nothing structurally prevents `rm -rf`, a fork bomb, or a real exploit fired at a production box the model misidentified as the lab.

The instinct is to bolt safety on afterward — sandboxes, command filters, "please don't" in the system prompt. But filtering a general-purpose shell is a losing game: the action space is infinite and the model is creative. The fix is not to constrain a powerful actor. **It is to never grant the power in the first place.**

THE GOVERNING PRINCIPLE

If the model can issue commands, no amount of prompting makes it safe. **Safety has to be a property of what the model is structurally unable to do.**

02 The Core Inversion: Playbook as Authority

The engine is split into two parts with a one-way relationship between them.

The deterministic core

A plain, version-controlled program. It owns the control flow: which step runs, in what order, against which target, under which authorisation. It is the only thing in the system that can cause a packet to leave the machine. There is no language model anywhere on this path.

The narrator OPTIONAL

A language model called only after execution finishes, handed the real output, and asked to write the executive summary in prose. It cannot select, add, reorder or trigger a single action. Remove it entirely and the engine still produces a complete report from a templated summary.

What the core is allowed to do is not open-ended. It is defined by a **playbook**: a human-authored, ordered list of steps, each naming exactly one tool from a fixed allowlist — no raw commands, no flags, no shell strings.

```
// A playbook step is data, not code. The engine can only run
// steps that exist here, in this order. Nothing invents new ones.
steps: [
  { id: 'discovery', tool: 'host_discovery',    // is the host alive?
    intent: 'Establish the target is up before scanning it.' },
  { id: 'ports',     tool: 'port_scan',         // what is listening?
    intent: 'Build the attack surface.' },
  { id: 'versions',  tool: 'service_version',   // exact software + version
    intent: 'Match services against known CVEs.' },
  { id: 'vulns',     tool: 'vuln_scan',         // detect, do not exploit
    intent: 'Flag known-vulnerable services.' },
  { id: 'validate',  tool: 'active_validation',
    intent: 'Prove a door opens — promote Detected to Confirmed.' },
  { id: 'exploit',   tool: 'flagship_exploit',
    intent: 'One vetted exploit + read-only post-exploitation.' },
]
```

This is the whole trick, stated plainly: **the playbook is the authority for what the tool may do.** "Autonomous" means the engine walks this list by itself, gating each step, recording each result, with no human in the loop. It does not mean the machine improvises. Improvisation is exactly the property you do not want in something that fires exploits across a network.

DESIGN PRINCIPLE

Determinism is not a limitation you tolerate for safety. **It is the feature that makes the output a defensible methodology instead of a one-time stunt.**

03 The Safe Executor & the Allowlist

A playbook step names a tool; it never contains a command. Translating a tool name into an actual process is the job of a single chokepoint — the **safe executor** — and it is deliberately small and boring.

- **Fixed allowlist.** Every runnable tool is an entry in a table. Each entry hard-codes its binary, its fixed flags and its timeout. There is no path by which a caller supplies a flag, a pipe or a shell string.
- **No shell, ever.** Processes are spawned by argument vector (`argv`), not through `/bin/sh`. Shell metacharacters are not "escaped" — they are never interpreted, because no shell is involved.
- **Container and target binding.** A tool may only run against a pre-validated target. A step that names a non-allowlisted destination is refused before anything spawns.
- **Tiered capability.** Most allowlisted tools are read-only probes that prove a door opens without walking through it. Exactly one is a vetted exploit, with fixed parameters, gated behind the authorisation checks below.

WHY THIS MATTERS

An allowlist of argv-locked binaries has a **finite, enumerable action space**. You can read it, test it and reason about its worst case. A general shell has an infinite one. The entire security argument of the tool rests on keeping that space finite.

04 Authorisation: Scope-Lock, Target-Lock, Kill-Switch

Before any step executes, the engine passes three independent gates. They are checked **per step**, not once at the start, so a long run cannot drift out of bounds halfway through.

GATE	GUARANTEES
SCOPE LOCK	Every engagement carries an explicit authorisation record — what is in scope, the rules of engagement, and whether intrusive actions are permitted. A target outside the authorised scope is rejected; the engine simply cannot be pointed at it.
TARGET LOCK	Immediately before a tool spawns, the resolved destination is re-validated against the locked scope. Detection-only engagements never reach the exploit step; the intrusive tool is gated behind a signed-authorisation flag that the read-only path never sets.
KILL SWITCH	An out-of-band abort that the engine checks at the head of every step. Tripping it stops the run cleanly between steps — no orphaned processes, no half-finished exploit.

DESIGN RULE

The signed-exploit gate is a property of the **code path**, not a config value a busy operator might flip. The only path that can fire an exploit is the one that runs against the throwaway lab. There is no field on a client engagement that turns intrusion on.

05 From Output to Evidence: Detected vs. Confirmed

A scanner banner is a claim, not a finding. A credible report distinguishes "the version string suggests this is vulnerable" from "we proved it." The engine encodes this as an **evidence tier** that a finding can only climb by producing real proof.

01 Extract

DETECTED

Findings are parsed deterministically from real scan output — open ports, service versions, known-vulnerable banners. Nothing is invented; every finding traces to a line the scanner actually produced.

02 Enrich & score

CVSS 3.1

Each finding is scored with a pure, spec-exact CVSS 3.1 base-metric calculator. Severity has to be defensible to a client's auditor, so the maths follows the published FIRST.org equations — including the round-up rule — rather than a hand-waved label.

03 Actively validate

CONFIRMED · PROBE

Benign, active probes attempt to prove the exposure: anonymous login actually succeeds, real shares and exports are listed, an injectable parameter responds. A Detected finding is promoted to Confirmed only on a definitive positive — and a validator binds to its finding by the exact port it probed, so one proof can never confirm an unrelated service.

04 Exploit, lab only

CONFIRMED · EXPLOIT

For the flagship finding on the disposable target, a single vetted exploit pops a real shell, followed by read-only post-exploitation to demonstrate genuine impact. This is the strongest tier of evidence and is never available on a client path.

The validation stage is pure and deterministic: `(findings, output) → new findings`. It never invents a finding, never downgrades one already proven by exploitation, and never mutates its input — it returns new objects. These properties are not aspirations; **they are pinned by tests**, because the credibility of every report depends on them.

WHAT THIS BUYS YOU

A finding is only as strong as the evidence tier it earned. **Encoding "detected" versus "confirmed" structurally is what separates a real pentest report from a vulnerability-scanner dump.**

06 Where the Language Model Earns Its Keep

Removing the model from the execution path does not make it useless — it makes it safe to use for the thing it is genuinely good at: turning a structured pile of findings into clear, human prose. The narrator is handed the finished, scored, validated findings and asked for an executive summary. Its role is deliberately and provably limited:

- **It explains results the deterministic core already produced.** It does not decide what to run.
- **It is never on the execution path.** It cannot trigger a command even in principle.
- **It is optional.** If the model endpoint is down, the engine falls back to a templated summary and the report is still produced in full.

This is the inversion that makes the whole thing work. The dangerous capability — deciding and acting — lives in deterministic, auditable, allowlisted code. The model contributes only the capability it is safe to give it: narrating after the fact. Neither half can do the other's job, **and that separation is the product.**

A USEFUL TEST

Ask of any "autonomous security tool": if the language model were swapped for a random-text generator, what unsafe thing could happen? In this architecture the answer is **"the summary reads badly"** — never "it attacked the wrong host." If your answer is worse than that, the model has too much power.

07 What It Takes to Build One — A Checklist

01 **A methodology to encode.** A recognised framework (PTES here) gives the playbook its phases: recon, scanning, vulnerability analysis, validation, exploitation, reporting. The framework is the spec; the playbook is its executable form.

02 **Playbooks as authored data.** Ordered, version-controlled step lists. Each step names one allowlisted tool and carries an `intent` so the report can explain *why* a step ran, not just that it did.

03 **A safe executor with a fixed allowlist.** Argv-spawned, shell-free, per-tool fixed flags and timeouts, target-bound. The single chokepoint through which any action must pass.

04 **Layered authorisation.** Per-step scope lock, target lock and a kill switch — plus a code-path-level gate that confines intrusive actions to disposable, self-owned targets.

05 **Deterministic findings and spec-exact scoring.** Parse real output into findings, score with a faithful CVSS 3.1 implementation, and track an evidence tier.

06 **An active-validation stage.** Pure, deterministic promotion of findings on definitive proof only — bound to the exact asset probed.

07 **A post-hoc narrator.** The model, off the execution path, optional, explaining results it could not cause.

08 **Reporting as a first-class output.** A methodology narrative, a confirmed-versus-detected split, per-finding status and CVSS, and a client-grade deliverable.

09 **Tests as the safety contract.** The guarantees above are only real if a test suite fails the build when they break.

08 Conclusion

It is tempting to measure an autonomous security tool by how much freedom you give the AI. *That is exactly backwards.*

The hard, valuable engineering is in how much you *take away* — and in doing it structurally, so that safety survives a bad model, a bad prompt, or a bad day.

Put the methodology in version-controlled playbooks. Put every action behind a finite allowlist and a layered authorisation gate. Make findings earn their evidence tier with real proof. And keep the language model where it can only ever explain — never act.

Do that, and "autonomous" stops being a euphemism for "uncontrolled," and starts meaning what it should: **a disciplined, repeatable, auditable test that runs itself and hands you a report you can stand behind.**

ABOUT REED SENTINEL

Reed Sentinel is a penetration testing and security consulting practice in Benoni, Gauteng, providing penetration testing and POPIA readiness to South African businesses. Every engagement runs under a signed authorisation and an agreed scope, and returns a plain-English report with the remediation for each finding — followed by a re-test that proves it closed.

Derek Reed

Principal Security Consultant · CEH
082 800 1008
derek@derekreed.co

Engagements

Fixed fee, agreed up front and scoped in a twenty-minute call. No system is touched without your signed authorisation.
derekreed.co