snyk

PLAYBOOK

The Automated AI Attacks Playbook

Your AppSec program was built on a quiet agreement: you cannot fix everything, so you prioritize, and you accept the rest. Automated attacks just repriced everything you accepted. This is what a program has to do now.

01 The attacker side already went agentic

Reasoning got cheap. Both attackers and defenders got it at the same time, but only one side is running continuously.

The evidence for AI-powered attacks is no longer speculative.

In one state-sponsored cyber espionage campaign disclosed in late 2025, as much as 90% of the operation was executed by AI rather than by human operators (*Anthropic Threat Intelligence, November 2025*). Note that it wasn't AI-assisted, but AI-executed, with humans supervising a campaign rather than running it.

The tempo followed. One campaign compromised more than 600 FortiGate firewalls across 55 countries in five weeks. Average AI-driven breakout time now sits at 34 minutes, and the fastest observed is just four. In June 2026, the Five Eyes Alliance warned that AI is on track to bypass current cybersecurity capabilities in months rather than years.

This shows up in the defensive data, too. In the first six months of 2026, valid AI-generated vulnerability reports to HackerOne rose 210%, with prompt-injection reports up 540%. That surge is concentrated in exactly the flaws that depend on reasoning rather than pattern matching, which is eminently telling: **the capability that is finding those flaws for researchers is the same capability finding them for everyone else**.

> "The attacker side of this equation has already gone agentic. The question is whether you get there first [to your own flaws]."
>
> MANOJ NAIR — CHIEF INNOVATION OFFICER, SNYK

The AI labs are getting better at stopping their own models from being weaponized, and that work matters… but stops at the model's edge. It has nothing to say about whether your invoice endpoint leaks another customer's data, or whether a five-year-old medium in your backlog is now two steps from an account takeover.

That is not a model problem; it's yours.

02 What automation repriced

Every AppSec program produces the same artifact at the end of every quarter, which is nothing but a list of the things you decided not to fix.

Nobody calls it that, though. It's announced as the universe of risk narrowing: everything that exists, then what tooling identified it, then what got prioritized, and then what actually shipped a fix. What is left at the end has a name, and the name is **accepted risk**.

Accepted risk was never a judgment about safety, but a judgment about capacity, wearing safety's proverbial clothes. Developers have outnumbered security professionals by something like a hundred to one for over two decades. No program was ever going to fix everything, so the discipline did the rational thing: it ranked, it fixed the top, and it quietly agreed that the rest would probably never get to be exploited.

For twenty years, that agreement has held. But that didn't happen because the residual risk was safe, rather because reaching it cost more than it returned. Nobody was going to spend a human pentester's fortnight enumerating a five-year-old medium in an internal expense tool to see whether it chained into anything.

That is the calculation that now broke: an autonomous attacker will spend exactly that on every finding, indefinitely, because the marginal cost of trying is now close to zero.

> **Nothing was added to your backlog; it was the economics of attacking it that changed.**

Two things moved at once.

Volume

The inflow is accelerating faster than any team's throughput. As of June 2026, Snyk sees roughly six new vulnerabilities introduced for every one remediated, against +108% growth in new issues between Q4 2024 and Q1 2026. Manoj Nair's own figure from RSAC puts AI-generated code at two to ten times more vulnerabilities per developer.

A program that finds six times faster than it fixes is not getting safer by finding more. It is building inventory for somebody else.

The backlog got repriced

The dormant lows and mediums were safe because they were *boring*. Boring is not a security control, though; it's an economic condition, which just expired.

Worse, the backlog is the least-examined part of the estate by design. Everything in it got there by being ranked below the cut line, which means nothing has ever gone back and asked whether the cut line was right, or whether two items below it add up to something well above it.

Which raises the question of nothing in the standard stack was built to answer: **what happens when they do?**

03 The attacks that nothing was built to see

Deterministic engines are very good at what they were built for, and that is not going anywhere: static analysis matches patterns in source code, dynamic testing throws payloads at a running application and reads the response, and between them, hundreds of vulnerability classes are caught reliably and cheaply.

None of them adds findings together. And the thing that does the damage is not a class of vulnerability at all, but a *composition*: an authorization gap on one endpoint, a logic flaw in how identifiers get issued, and a stale session the cleanup job missed. Every engine will score all three correctly, and on its own merits, each one is minor flaw. Severity is a property of a single finding, and a chain is a property of a system of findings. No rule describes it, because the rule you would need depends on how components relate rather than on what any one of them contains.

That is a capability boundary, and it is why chained paths have always been human pentesting work. It is also where the accepted pile comes back, because each of those three scored low enough to be deprioritized. **Every chain is assembled from exactly the findings you accepted.**

And the bar is lower than most programs assume. In 2019, First American exposed roughly 885 million financial documents by changing a single number in a URL: no chain, no exploit code, one missing check, every scanner clean. If a single oversight is worth 885 million records, nothing in the standard stack has ever been asked to price a *deliberately assembled* path through three.

What changed is that frontier reasoning can now hold an entire system in view and do the part only a human could do before, repeatedly, at a cost that permits it to run *continuously* rather than *annually*.

Reasoning extends the engines, but does not replace them. The deterministic layer still owns the commodity classes, because it does that work exhaustively for pennies. Reasoning goes where reasoning is the only thing that works. A program needs both, and any vendor telling you otherwise is selling you just half of it.

04 The checkpoint that stopped being a checkpoint

The pull request scan was a good control for as long as one assumption held: that code arrived at a rate a review could absorb.

That assumption is gone, and the response has been to distribute security across three control points rather than concentrate it at one:

- **In real time, inside the agent's inner loop**: guidance at the moment code is authored, where latency budgets are measured in milliseconds, and only deterministic engines can keep up.
- **Deterministic scans in CI**: the exhaustive, repeatable pass over what actually got committed.
- **Deep contextual analysis across the codebase**: the slower, more expensive look at how components relate, where reasoning earns its cost.

Each is necessary, while none by itself ends up being sufficient. And the interesting question is what falls in the cracks between them.

Authorization logic, for instance, is invisible in the inner loop because a single file does not reveal a trust boundary, and is invisible in CI because no rule describes it. Chained paths also fall between them, because every control point evaluates a change, and a chain is a property of the whole. And then, the backlog falls between all three, because every one of them is looking at what is arriving, not at what is already there.

Three control points built for the flow of new code do not, between them, constitute a program. They are just the front door, while the playbook below gives you the rest of the house.

05 The Playbook: Discover, Remediate, Validate, Prevent

Four moves, running as a loop, where each one makes the others sharper.

01 DISCOVERY	02 VALIDATION	03 REMEDIATION	04 PREVENTION
Points validation at the right targets.	Tells remediation what actually matters, rather than what merely scores high.	Clears the ground, so validation is not rediscovering the same debt every quarter.	Stops the loop from running backward.

Run one without the others, and it degrades. Discovery without remediation is just a longer list. Remediation without prevention is bailing while the tap runs. Validation without discovery tests the part you already knew about.

Here is what each move requires, how Snyk delivers it, and the question you can ask any vendor, including us.

Discover

You cannot test, prioritize, or govern an estate you cannot see. That means the applications nobody owns, still serving traffic and absent from the last three scoping conversations, and it means the models, agents, MCP servers, and skills now running inside them, along with what each one can actually reach. Shadow AI is the default state rather than the edge case, so any process that asks you to declare your attack surface in advance will only cover the part you already knew about.

Then there is the harder half. You cannot reason about authorization without knowing what the application is supposed to do. That means a living model of architecture, data flows, trust boundaries, and which components hold what. And you need that maintained continuously, not reconstructed by a consultant every eighteen months or so.

This is the prerequisite for everything else on this list. It is also the step most programs skip, because it produces no findings and therefore no metrics to keep track of.

HOW SNYK DELIVERS IT

AI-SPM and the AI-BOM map the AI estate, including the shadow usage that never went through review. Underneath that, the platform carries accumulated application context from Snyk Code, Snyk Open Source, and Snyk API & Web: not a list of findings, but the structure every later move reasons from.

ASK

What does your tooling know about my application beyond its files and dependencies, and how does that understanding stay current as the application changes?

Remediate

Finding was never the bottleneck. But merging was.

Every program already holds more findings than it can fix. Adding detection capacity to that is not a strategy. Clearing a backlog at this rate is a throughput problem, and throughput has three requirements: fixes good enough that a developer merges them without rewriting, coverage across both new code and everything already shipped, and something independent confirming the fix actually holds.

The coverage requirement is the one to press on. Most remediation tooling is aimed at the inflow, because that is where the clean test cases are. The backlog is where the chains are.

HOW SNYK DELIVERS IT

The Remediation Agent, in public preview through the CLI and developer environments, pairs frontier reasoning with the platform's intelligence layer: reachability, exploitability, package health, and a decade of fix-outcome data. That intelligence is the difference between a proposed fix a developer merges and one they rewrite, which, at this rate, is the only difference that matters.

ASK

What proportion of your fixes merge without human rework, and what independently confirms the fix held, as opposed to the ticket closing?

Validate

Frontier reasoning alongside deterministic engines, grounded in that context. Not instead of them, and not pointed at a target cold.

There are two separable claims in any reasoning-based detection pitch, and they need testing separately. The first is that it finds things scanners cannot find, and the second is that what it finds is real.

Which is the same problem one level up, because everything above is a claim until someone (or something) tries to break it.

The principle underneath is the one the whole discipline now rests on: **the generator cannot be the validator.** A system that finds a flaw, proposes the fix, and then certifies its own work has a conflict of interest baked into its architecture, and no amount of prompting resolves a structural problem. The fix for unreliable AI is not more AI. It needs reasoning bounded by independent verification, from something with no stake in the answer.

Applied to a program rather than a finding, that means adversarial testing against your own controls, continuously. And the requirement stands independent of who supplies it.

> **A guardrail you have not attacked is not a guardrail; it's an assumption.**

HOW SNYK DELIVERS IT

Evo Continuous Offensive Security runs offensive assessment against the running application on the cadence of development actually moves, grounded in what the platform already found, rather than starting cold. Findings arrive as exploit chains with runnable proof, and a dedicated validation model acts as an independent judge, confirming exploitability before any finding surfaces. That is the generator-validator separation shipped as architecture rather than asserted as a principle.

ASK

What proportion of your findings are confirmed exploitable rather than flagged as suspicious, and what grounds the reasoning: my code and architecture, or a model's general knowledge of what applications usually look like? And then: what proves the controls work, who validated the validator, and how often?

Prevent

Guardrails at authoring time, secrets caught before commit rather than rotated after exposure, and malicious packages blocked at install rather than discovered in an audit.

Prevention is not a *nice-to-have* alongside remediation. At a six-to-one ratio, remediation alone cannot outrun the inflow. You can staff, automate, and prioritize as well as anyone in the industry and still lose ground, because the rate is the problem, and prevention is the only lever that acts on the rate.

HOW SNYK DELIVERS IT

Snyk Secrets, generally available since August, enforces leaked-secret prevention from the IDE and pre-commit through pull request and CI. Snyk Studio puts the platform's deterministic engines inside AI coding workflows, so the same standard applies whether a human or an agent authored the change, which increasingly it is.

ASK

What stops a new issue from entering, as opposed to finding it faster once it has?

06 Readiness: six questions

This is not a maturity model, but six questions that surface the gap the fastest.

MOVE	QUESTION	IF THE ANSWER IS NO
Discover	**Does anything in your stack know what your application is supposed to do?**	Then nothing can reason about authorization, and the highest-impact category stays invisible regardless of how much you scan.
Remediate	**How fast is your backlog growing against your fix rate?**	If you do not know the ratio, you cannot tell whether the program is gaining or losing. It is the single most diagnostic number you are probably not tracking.
Remediate	**What proportion of proposed fixes merge without rework?**	Remediation throughput is your real constraint, and it is being measured as a detection problem.
Validate	**Do you know which of your findings chain together?**	You are prioritizing by severity, which cannot see combinations. Your critical path is likely already in the backlog as three separate mediums.
Validate	**When was your risk register last re-examined against an automated attacker rather than a human one?**	Everything on it was accepted under economics that no longer apply. The register is a record of decisions made in a different market.
Prevent	**What stops a new issue from entering, rather than finding it faster afterward?**	You are fighting the rate with volume, and the rate wins.

Underneath all six: **when something is fixed, does anything independently confirm the path is closed?** If the answer is "*The ticket was marked resolved,*" that is the cheapest thing on this list to change, and it is the difference between measuring activity and measuring risk.

07 The first ninety days

None of this requires a reorganization. All of it is measurable by the end of a quarter.

Days 1–30: Baseline the rate

DISCOVER

- **Measure inflow against outflow.** New findings entering versus findings closed, weekly, for your top ten applications. You are looking for your own version of six-to-one.
- **Re-examine the accepted pile.** Not all of it. Take the internet-facing applications and ask what is in their accepted-risk list, when each item was accepted, and against what assumption.
- **Find the applications nobody owns.** Still serving traffic, still holding data, absent from the last three scoping conversations. That is where the untested authorization logic lives.

Deliverable: your ratio, and a list of what you accepted when reaching it was expensive.

Days 31–60: Unblock the merge path

REMEDIATE, PREVENT

- **Instrument the fix pipeline.** Where does a fix actually stall: generation, review, regression risk, or ownership? Programs assume triage, but it's usually merge.
- **Test remediation throughput on real debt**, not on a demo. Take a hundred backlog items and measure what proportion reaches production without a human rewriting the patch.
- **Put gates on the highest-volume inflow** you have, whether that is secrets, dependencies, or a single noisy class.

Deliverable: a merge rate, and one inflow stream measurably reduced.

Days 61–90: Turn the rate around

VALIDATE, PREVENT

- **Run adversarial testing against what you have fixed**, and against the controls you just installed. Confirm paths are closed rather than tickets.
- **Re-measure the ratio.** This is the number that tells you whether the program changed or the activity did.
- **Set the reporting line.** Two numbers, monthly: the rate of new risk entering, and the rate of proven exploitable paths closing.

Deliverable: two defensible numbers, and the first period where outflow beat inflow.

After 90 days, the work is maintaining a growing ratio of fixes to issues, while the volume keeps climbing. That is a different job from the one most AppSec teams were hired for, but it's the essence of the job now.

Defense at the speed of offense

The attacker's side already went agentic. The defensive side is still running an operating model designed around a constraint that has quietly changed.

Accepted risk was the rational response to being outnumbered a hundred to one with human hands and human hours. It's not a law of nature. It was a capacity decision, and capacity is exactly what changed. For the attacker first, and now for you.

So the question is not how to find more or how to triage better. Both of those are refinements to a model that assumed you would never get through the entire list. The real question is whether your program can run at the rate the problem now runs at: **discovering** what you actually have and what it is supposed to do, **remediating** at machine rate, **validating** that any of it holds by attacking it, and **preventing** the next wave from rebuilding what you just cleared.

You do not have to accept the backlog. You have to outrun it.

Talk to a Snyk expert today