IBM **X-Force Red**

The real applied risk of AI in vulnerability discovery and exploit development

How AI assisted vulnerability research is reshaping the front of the attack chain, and what defenders should do about it.

Patrick Fussell
Global Head of Adversary Simulation
IBM X-Force Red

Table of contents

Executive summary

Where AI driven vulnerability discovery sits on the capability curve today, and what that means for an attacker's kill chain.

The public conversation about artificial intelligence and offensive security has spent the last two years bouncing between two positions. On one end is the claim that large language models will end software security as we know it. On the other is the assertion that this is all hype and AI will go away soon. The reality, as measured by published research and disclosed CVEs in the last twenty-four months, sits between those poles and deserves deeper discussion.

This paper assesses where AI driven vulnerability discovery and exploit development sit on the capability curve today in pragmatic application to real world attack chains. In other words, how AI powered vulnerability research applies directly to the perspective of a threat actor's attack chain.

It examines two representative classes of work. The first is static audit of open-source codebases that underpin production infrastructure, the kind of code that runs Linux based load balancers and Java application frameworks such as Apache Struts, and what this means for an attacker's ability to establish unauthorized access to a network. The second is active discovery of vulnerabilities through iterative interaction with a running service, with server-side request forgery as a worked example.

The goal is to give defenders a grounded view of what the technology can do, what it cannot do yet, and what that means for risk posture in real environments.

Introduction

Vulnerability discovery has always been bottlenecked by skilled human attention. AI is changing that economics.

Vulnerability discovery has always been bottlenecked by skilled human attention. The group of people with the requisite skillset needed for deep vulnerability research is not large. A talented researcher with a fuzzer, a debugger, and extended time for focused work can produce results that no team of generalists can match. That asymmetry is what has made zero-day discovery one of the most expensive labor markets in security, with brokers publishing payouts in the millions for premium iOS and Chrome exploit chains.

The thesis behind AI driven vulnerability research is straightforward. Some meaningful portion of that human attention can be replicated, scaled, and turned loose against software that no human team has the bandwidth to audit.

Recent results suggest the thesis is partially correct. Google's Big Sleep agent, a joint project between DeepMind and Project Zero, identified an exploitable memory safety bug in SQLite in late 2024, the first public case of an AI agent finding a previously unknown memory safety issue in widely used real world software [1]. In mid 2025 the same agent identified CVE 2025 6965 in SQLite before threat actors who had been tracking the same flaw could weaponize it [2].

DARPA's AI Cyber Challenge final competition in August 2025 saw seven cyber reasoning systems run autonomously for roughly 143 hours against 53 challenge projects derived from critical infrastructure software. Across those runs the systems found 54 of 63 synthetic vulnerabilities and generated 43 working patches, and they surfaced 18 previously unknown real-world bugs that are now in responsible disclosure [3]. Independent researchers have replicated similar results at smaller scales, including disclosures of multiple CVE assigned issues in widely deployed frameworks for the cost of a few dollars in API spend [4].

These results are real but for those not versed in applied attack chains they only tell a small part of the story. The remainder of this paper looks at two specific shapes of the problem and what each shape implies for an attacker.

Class one

Static audit of open-source infrastructure code

Consider a typical enterprise load balancer. It runs on Linux, terminates TLS through a userspace daemon, parses HTTP at scale, and routes traffic to a backend pool. Its attack surface includes the kernel itself, the network stack, the parsing libraries used by the load balancing daemon, and any management interface exposed for operations.

Apache Struts, in service across enterprise Java frontends for the better part of two decades, is structurally similar. It sits in front of business logic, accepts untrusted input, deserializes it, and dispatches it through layers of framework code. Both targets share two properties that matter for our purposes. They are written in memory unsafe or weakly typed languages with long histories of subtle bugs, and their source is fully public.

Why this code shape is tractable for an agent

The combination is what makes them tractable for an AI driven workflow. A modern LLM agent can ingest a function, a translation unit, or a call graph and reason about it the way a junior reverse engineer would, without the biological challenges every human must contend with and across a much larger surface.

Big Sleep's SQLite work is the prime example. The agent navigates the codebase using a set of specialized tools, forms hypotheses about where memory corruption is likely, generates targeted inputs in a sandboxed Python environment, and observes program behavior under a debugger [1]. None of these steps is novel in isolation. Fuzzers and static analyzers have done them for years. What is new is that an agent can move between code reading, hypothesis formation, and dynamic confirmation in a tight loop, the way a human auditor does when working through unfamiliar code.

For load balancer and framework style targets the same loop applies. An attacker working against the Linux kernel TCP stack, an Nginx parsing module, or a Struts dispatcher can ask an agent to enumerate functions that handle attacker-controlled input, identify the ones where that input crosses a trust boundary without effective validation, and generate proof of concept inputs that confirm the suspicion. The AIxCC results show that this is not theoretical. In the results we see the teams from the competition cyber reasoning systems found and patched bugs across Jenkins, the Linux kernel, Nginx, SQLite3, and Apache Tika under timed conditions with no human in the loop during the run [3].

The Struts lineage as a case study

It is worth grounding this in the history of the Struts framework itself. CVE 2017 5638, the Jakarta Multipart parser flaw that drove the Equifax breach, was a textbook example of attacker-controlled input flowing through error handling into OGNL expression evaluation [5]. CVE 2018 11776 exploited namespace handling in a more subtle way that required reasoning across configuration and dispatcher code [6]. CVE 2023 50164 abused case sensitive parameter handling to enable path traversal in file upload processing [7].

Each of these bugs has the same family resemblance. Untrusted data crosses a boundary, gets interpreted by code that assumed a stricter contract, and produces unintended behavior. This pattern is exactly the kind of thing an agentic workflow is well suited to find. It is also exactly the kind of thing that signature based static analysis has historically missed, because the bug is in the semantics of the data flow rather than in any single suspicious construct.

What the agent gives the attacker is automation and power to move through the most expensive part of the work. Reading code and forming an accurate model of intent has always been where vulnerability researchers spend their hours. An agent that can produce a credible first pass at that work, even with a false positive rate that requires human triage, changes the economics of who can credibly target widely deployed open-source infrastructure. It does not eliminate the need for a skilled operator. It does shift the cost curve in a way that defenders should plan for.

An applied scenario

Consider an applied scenario rooted in the Struts situation discussed above. A threat actor targets a financial services firm whose external posture, identified through banner enumeration and JSP error fingerprinting, indicates Apache Struts in production. The attacker is not waiting on the next public CVE. They point an agent, or an already running fleet of agents, at the Struts source tree with focused attention on the regions that have historically produced trouble. Multipart parsing, file upload handling, and namespace dispatch.

The agent surfaces a candidate bug that carries the family resemblance to previous bugs but the attacker can confirm it has no advisory attached and no signature in any commercial WAF. The attacker stands up a vulnerable instance locally, confirms exploitability, and crafts a working payload before any maintainer is aware the bug exists. From that point the chain is conventional. Reconnaissance had already mapped the target. The exploit lands a memory resident web shell. Lateral movement proceeds through the application server's effective permissions and into the backend data tier.

What the AI changed was not the post exploitation tradecraft, which is worth noting here still rewards an operator with strong tradecraft skills. It changed the front of the chain. Initial access pivoted from waiting on someone else's disclosure to producing the disclosure themselves without the type of investment in research or zero days needed in the past, on a timeline measured in days of compute rather than months of skilled human attention. The defender sees the same web shell they have always seen. What they no longer see is the months long window

between a maintainer's patch commits and an exploit campaign in the wild, because that window is the part the attacker removed.

> *One important caveat. The Big Sleep team itself noted that for many targets a well-tuned, target specific fuzzer remains at least as effective as the agent. AI driven static audit is not a replacement for the existing toolchain. It is a new layer that runs alongside it, and its outputs still require validation before they are usable.*

Recent benchmarks confirm this. A systematic study of LLM driven proof of concept generation for web vulnerabilities found success rates between 8 and 34 percent when working from public disclosure information alone, rising to 68 to 72 percent only when supplemented with code context and adaptive prompting strategies [8].

Class two

Active discovery through service interaction

The second class of work looks different. Here the attacker has a running service, often a web application with limited or no source access and needs to discover and confirm a vulnerability through interaction.

Server-side request forgery is a useful illustration because it has been an underrepresented vulnerability class for years despite being implicated in some of the most consequential incidents on record. Even given it being a less commonly exploited vulnerability, in attack chains there are several breaches that can be traced back to an SSRF.

Why SSRF is hard for traditional tools

SSRF is often hard to find through pattern matching. The vulnerability lives in how a backend service handles a URL or hostname that an attacker influences, and confirming it usually requires probing internal network behavior, parsing nuanced error responses, and chaining the initial request with follow on probes to validate impact.

A traditional automated scanner sends a fixed payload, looks for a fixed indicator in the response, and reports a finding. It can identify obvious cases where a URL parameter triggers an outbound HTTP request to an attacker-controlled listener. It struggles with the rest of the class. Blind SSRF where the response is suppressed, SSRF through unusual protocol handlers such as gopher or file, SSRF that only shows up after the application performs a series of internal lookups, and SSRF chained against cloud metadata endpoints all require interpretation that a fixed signature scanner cannot perform.

Why the agent loop fits

This is precisely the kind of work that benefits from an agent loop. The agent can send a request, read the response, reason about what the response implies, and choose a next request informed by that reasoning.

Daniel Kang's group at the University of Illinois demonstrated in 2024 that an LLM agent given a description of a vulnerability class could autonomously exploit a meaningful fraction of zero day web vulnerabilities in test environments, including SSRFs that required multi step reasoning to confirm [9]. A follow up systematic evaluation in 2025 compared twenty agent frameworks across categories including SQL injection, SSRF, and XSS, finding that top performing systems such as OpenHands, SWE agent, and CAI could reliably reproduce real SSRF exploits when given disclosure information as a starting point [10].

Static analysis variants of the same idea are also producing results. The Artemis taint analysis system, which uses an LLM to extract source and sink definitions before running interprocedural path sensitive analysis, reported 106 confirmed SSRFs across 250 PHP applications in 2025, of which 35 were previously unknown and 24 received CVE assignments [11].

The mechanism is worth examining. When a human tester pokes at a suspected SSRF, they read the response time, the body, and the error messages, and they form a working model of what the backend did with their input. They then craft a follow-on request that disambiguates. Did the response come from an internal address. Did the application follow a redirect. Did the backend resolve a DNS name the tester controls.

An agent does the same thing, just faster and at request rates the human cannot match. For the operator, the practical payoff is in the long tail. Most production web applications have been scanned by traditional tools for years and the easy findings are gone. What remains are vulnerabilities that require interpretation, chaining, and contextual reasoning. SSRF is one example. Insecure direct object reference, race conditions in business logic, and second order injection are others. These are the vulnerabilities where an agent earns its compute cost.

An applied scenario

The exploit path here is operationally distinct from Class one but no less consequential. Vulnerabilities that survived years of human review and signature-based scanning often did so because confirming them required interpretation that humans get tired of performing or may miss due to sophisticated contexts.

An agent that can read a response, reason about what the response implies about backend behavior, and select the next request based on that reasoning closes that gap. Consider a URL preview feature on a customer facing application, the kind that fetches a remote page to render an inline card. An agent probing that endpoint can submit a sequence of carefully chosen hostnames, observe response timing and body length, infer that one of those requests reached an internal address, and pivot from confirmation to IMDS credential theft within the same session.

No payload upload occurs and no memory resident web shell is needed. The compromise is complete the moment the temporary credentials are in attacker hands, and from there the chain proceeds through whatever IAM role the application server happened to assume. What changed is who can produce that pattern at scale, against which targets, and on what timeline. A vulnerability class that historically rewarded patient skilled testers is now reachable by anyone with API budget and the operator skill to interpret what comes back and understand what steps should be taken next.

What this means for risk posture

A few conclusions follow from the current state of the field.

The threat is real but bounded

AI driven vulnerability discovery works best where the target has reviewable code or a responsive interactive surface, where the bug class is reasonably well understood, and where the operator running the system can validate findings. It is not a magic box that produces exploit chains on demand. It is a force multiplier for operators who already understand the problem space.

The threat scales differently than human research does

A skilled zero-day researcher targets a small number of high value codebases. An agent fleet can audit thousands of repositories in parallel for a marginal compute cost. This is the change that matters most for organizations responsible for software supply chains.

The historical assumption that low profile open-source dependencies will not be audited by sophisticated actors is no longer reliable. The economic floor for who can credibly run such an audit has dropped from a small number of nation state aligned teams to anyone with API budget and operator skill. Reporting from Google's Threat Intelligence Group on AI generated zero-day exploits surfacing in live operations [12] suggests that some of this is already happening in practice.

The same tooling is available to defenders

The agent that finds an exploitable bug for an attacker finds the same bug for a maintainer or a red team. AIxCC was structured around this symmetry. Systems were judged not only on discovery but on producing working patches, and the leading systems were required to release their code as open source after the competition [3]. Organizations that build production capacity for AI assisted code review and AI assisted application testing will close the timing window that attackers are otherwise free to widen.

> *The honest summary is that we are past the proof of concept stage and into the early deployment stage. The question for defenders is not whether the capability exists. It is how to integrate it into the same operational*

> *workflows that fuzzers, static analyzers, and DAST tools occupy today, and how to update threat models to reflect that an attacker who could only afford a junior researcher last year may be able to afford the equivalent of a full audit team this year.*

AI driven vulnerability discovery threat model

A structured view of attacker capabilities, access requirements, vulnerability classes, success bounds, and defensive controls across four tiers.

Attacker tier	Access assumptions	Vulnerability classes discoverable	Expected success bounds	Defender controls that reduce yield
Tier 1 Commodity operator	Chat interface access to frontier models. Public PoCs and CVE disclosures. Black box interaction. No source review.	N day reproduction from public disclosures. Reflected XSS and basic SQLi via guided probing. Surface level SSRF.	8 to 34 percent PoC reproduction on disclosed CVEs from description alone. Effectively zero on undisclosed targets.	Patch SLA enforcement on disclosed CVEs. WAF signatures for high profile classes. Input validation as baseline. Asset inventory tied to vulnerability management.
Tier 2 Skilled individual or small team	API budget in tens to hundreds of dollars. Agentic frameworks such as OpenHands, SWE agent, or CAI. Public source for OSS targets. Iterative black box testing.	Novel SSRF with multistep reasoning. Logic flaws and IDOR. Function or file level memory safety bugs in smaller OSS projects. CVE weaponization within hours of patch publication.	68 to 72 percent PoC success when code context plus adaptive prompting is supplied. Individual researchers producing CVEs for single digit dollar API spend.	Pre disclosure code audit using the same agent tooling. Internal red team adoption of agent workflows. Behavioral detection on probing patterns. Hardened SDLC including taint analysis on dependencies.
Tier 3 Well resourced criminal group or commercial offensive vendor	Custom cyber reasoning system pipelines combining static and dynamic analysis. Fleet scale compute for parallel audit. Target reconnaissance feeding the hunt.	Novel zero days in widely deployed OSS infrastructure. Memory corruption in C and C++ code paths. Chained vulnerabilities across components. Zero days in obscure dependencies.	Comparable to AIxCC finalist performance. 54 of 63 synthetic vulns found and 18 real world zero days across 143 hours of autonomous run time. Time to first finding in days, not weeks.	Continuous AI assisted audit of high value dependencies. Memory safe languages where feasible. SBOM with active monitoring. Bug bounty economics scaled to attacker compute cost. Tabletop exercises that assume a zero day exists.
Tier 4 Nation state or peer adversary	Purpose built systems analogous to Big Sleep. Custom or fine tuned models. Threat intelligence integration.	Predisclosure zero days where intelligence informs which code to audit. Cross system attack chains. Vulnerabilities in	Demonstrated capability to identify and disrupt threat actor exploitation operations before the exploit is used, as in Big Sleep and	Equal footing AI deployment in defensive research. Intelligence sharing across ISAC and government channels. Predisclosure research collaboration

Attacker tier	Access assumptions	Vulnerability classes discoverable	Expected success bounds	Defender controls that reduce yield
	Coordinated human and AI workflows with skilled operators on the loop.	proprietary code obtained through supply chain access.	CVE 2025 6965. Contesting on equal footing with any attacker fleet.	with maintainers. Assumption of breach posture funded as a peer capability to prevention.

Reading the model

The tier boundaries are economic, not technical. The same agent framework moves between tiers depending on who is paying the API bill and who is interpreting the output. The reason this matters for risk modeling is that Tier 2 is the line that shifted hardest in the last twelve months. Work that previously required Tier 3 budget now sits inside what a motivated individual can sustain.

The success bounds column reflects published benchmarks where they exist. For Tier 3 and Tier 4 the numbers are derived from competition data and disclosed operational results rather than controlled benchmarks, so they should be read as existence proofs rather than expected values.

Key observations

Capability gaps between tiers

The jump from Tier 1 to Tier 2 is primarily about operator skill and custom tooling, not model capability. Tier 3 represents the current state of the art as demonstrated by Big Sleep and AIxCC winners. Tier 4 capabilities are partially theoretical but consistent with disclosed nation state TTPs.

Access as a forcing function

Source code access remains the primary differentiator for static analysis approaches. Test environment access enables the iterative loops that drive success rates from 8 percent to 68 percent and above. Production access is not required for discovery, only for validation and weaponization.

Defensive control effectiveness

Traditional controls such as SAST and fuzzing remain effective against Tier 1 and Tier 2 threats. Tier 3 threats require AI assisted defense to match attacker capability. Tier 4 threats require architectural controls that assume compromise.

Economic implications

Tier 1 attacks cost dollars in API spend. Tier 2 costs thousands in operator time. Tier 3 requires institutional investment in the tens to hundreds of thousands of dollars. Tier 4 represents nation state budgets in the millions. The cost floor for credible vulnerability discovery has dropped by roughly an order of magnitude in two years.

Sources

References cited in the body of this paper.

[1] Google Project Zero and Google DeepMind. "From Naptime to Big Sleep. Using Large Language Models to Catch Vulnerabilities in Real World Code." Project Zero blog, November 2024.

[2] Google Cloud. "Cloud CISO Perspectives. Our Big Sleep agent makes a big leap." Google Cloud blog, July 2025.

[3] Defense Advanced Research Projects Agency. "AI Cyber Challenge marks pivotal inflection point for cyber defense." DARPA news, August 2025.

[4] Huang, K. "Token Is All You Need. Finding Zero Days with LLMs and Agentic AI." Substack, April 2026.

[5] Apache Software Foundation. "S2 045 Possible Remote Code Execution when performing file upload based on Jakarta Multipart parser (CVE 2017 5638)." Struts security bulletin, March 2017.

[6] Apache Software Foundation. "S2 057 Possible Remote Code Execution when using results with no namespace (CVE 2018 11776)." Struts security bulletin, August 2018.

[7] Apache Software Foundation. "S2 066 File upload component vulnerable to path traversal (CVE 2023 50164)." Struts security bulletin, December 2023.

[8] Wei, Y. et al. "A Systematic Study on Generating Web Vulnerability Proof of Concepts Using Large Language Models." arXiv preprint 2510.10148, 2025.

[9] Fang, R., Bindu, R., Gupta, A., and Kang, D. "LLM Agents Can Autonomously Exploit One Day Vulnerabilities." Working paper, University of Illinois at Urbana Champaign, 2024.

[10] Zhang, L. et al. "LLM Agents for Automated Web Vulnerability Reproduction. Are We There Yet?" arXiv preprint 2510.14700, 2025.

[11] Ji, Y., Dai, T., Zhou, Z., Tang, Y., and He, J. "Artemis. Toward Accurate Detection of Server Side Request Forgeries through LLM Assisted Interprocedural Path Sensitive Taint Analysis." Proceedings of the ACM on Programming Languages, OOPSLA1, 2025.

[12] Google Threat Intelligence Group. "AI Threat Tracker. Advances in Threat Actor Usage of AI Tools." GTIG report, 2025.