sonatype

2026

State of the SOFTWARE SUPPLY CHAIN®

TABLE OF CONTENTS

Foreword 3

Executive Summary 4

Registries, Models, and the New Software Infrastructure Burden: When Growth Meets Gravity 7

Open Source Scale Has Become a Structural Risk 7

Registry Consumption 9

Real Innovation vs. Synthetic Volume 13

The Commons is Cracking 13

What Responsible Consumption Looks Like 18

The Evolving Software Supply Chain Attack Surface: Malware at The Gate 20

A Turning Point for Open Source Malware 20

The Threat Taxonomy: What Open Source Malware Does Today 22

How North Korea Weaponizes Open Source 24

The Open Source Malware Supply Chain 28

Emerging Threats 29

How Will Software Supply Chain Attacks Evolve? 30

The Three Layers of Failure in Modern Vulnerability Management 31

The Limits of Modern Vulnerability Management 31

The Data Layer is Breaking Down 33

Poor Consumption Patterns Sustain Avoidable Risk 36

When the Ecosystem Stops Maintaining Your Software 39

How the Three Layers Compound Each Other 42

Modernizing Vulnerability Management 44

From Guesswork to Grounded: AI Agents with Real World Intelligence 46

Version Hallucination: LLMs Hallucinate Versions at Scale 47

Security Improvement by Upgrade Strategy 49

Grounding is the Missing Link 54

The 2025 Global Software Assurance Mandate: Transparency as the New Trust 55

Transparency Has Become the Currency of Software Supply Chain Security 55

From Open Source Governance to Regulatory Mandate 56

United States Software Regulations 58

European Union Software Regulations 58

Other Key Jurisdictions 60

Regulated Industries: From Obligation to Opportunity 60

Formats and Interoperability 61

Open Source License Compliance in the New Regime 62

Bringing Policy into Reality 62

Software Assurance as Currency 64

Methodology 66

FOREWORD

For most of my career, open source has run on a simple premise: shared building blocks make everyone faster. That is still true. What is not optional anymore is everything that comes with running that premise at a global, automated scale.

Open source is now the substrate of software delivery, pulled continuously by pipelines and rebuilt across fleets that rarely stop. At machine speed, small inefficiencies and small risks do not stay small. "Just one more build" becomes billions of requests, and then everyone acts surprised when the infrastructure starts to groan.

You'll see it first in the operational reality of the commons. The same CI/CD patterns that make teams productive can generate massive redundant load when caches are cold, runners are ephemeral, or pipelines are effectively configured to re-download the world. If your build environment forgets what it did last run, the ecosystem still pays the cost.

You see it again in the security reality. Attackers target open source because it is the fastest path to developers, and developers sit closest to credentials, tokens, and build systems. Malware is steady pressure on ecosystems designed for openness. At the same time, public vulnerability intelligence is too often incomplete, late, or wrong, which turns prioritization into guesswork. That's not a tooling problem. It's a signal problem.

And now AI is entering the loop. It can accelerate good engineering, but it can also scale mistakes when it's operating from static training data instead of live reality. When a model doesn't know what versions exist or what is newly risky, it predicts and fills in the blank. That's how you end up with confident "upgrades" to versions that don't exist and recommendations that look plausible right up until they break your build or your policy. AI should not guess. AI-driven velocity will overwhelm any governance model built on "we'll review it later."

This report is about what happens when trust becomes a scaling problem. The takeaway isn't that open source is unsafe or that teams should slow down. It is that the ecosystem has matured into critical infrastructure and we need to operate it like one. That means responsible consumption, security controls that match modern development, and transparency that is produced by the build, not assembled after the fact. Regulations and buyers are moving there because the world is demanding evidence, not assurances.

Open source will keep powering innovation. The question is whether we build the practices and infrastructure to sustain it at the scale we now depend on, or whether we keep acting like the bill is someone else's problem.

> **AI CAN ACCELERATE GOOD ENGINEERING, BUT IT CAN ALSO SCALE MISTAKES WHEN IT'S OPERATING FROM STATIC TRAINING DATA INSTEAD OF LIVE REALITY. GUARDRAILS FOR AI ARE NO LONGER A NICE-TO-HAVE.**

Brian Fox
Co-founder and CTO, Sonatype

Executive Summary

Software supply chains have hit machine scale. In 2025, the world did not just build more software. It reused more of it, more often. That scale is bending the ecosystem in predictable ways. Open source registries, now largely serving as the internet's critical infrastructure, are under sustained strain. Synthetic traffic and redundant downloads inflate the commons, and attackers increasingly treat open source as a delivery channel, not an afterthought.

IN 2025, THE WORLD DID NOT JUST BUILD MORE SOFTWARE. IT REUSED MORE OF IT, MORE OFTEN.

THE KEY TAKEAWAYS

1,233,219

open source malware packages logged by Sonatype since 2019

9.8 TRILLION

downloads across Maven Central, PyPI, npm and NuGet

27.76%

recommended dependency upgrade hallucination rate observed with leading LLM

65%

of open source CVEs were left without CVSS by the NVD

Vulnerability intelligence is getting noisier and less complete just as teams need it to be faster. AI-assisted development is also introducing a new class of risk — automation can amplify bad inputs at machine speed. Against a backdrop of accelerating regulatory mandates for transparency, the message of this report is simple:

TRUST AT SCALE IS NOW THE CENTRAL ENGINEERING AND BUSINESS CHALLENGE OF MODERN SOFTWARE.

Growth Meets Gravity: Automated builds, ephemeral environments, and larger dependency graphs drive repeat pulling at enormous scale. Registry infrastructure is now critical plumbing, and the cost of operating the commons rises faster than most stakeholders realize.

Synthetic Growth is Not the Same as Innovation: Spam publishing, malware floods, and CI/CD misconfigurations can inflate downloads and releases without adding value. The result is wasted bandwidth, higher operating costs, noisier signals, and a larger attack surface.

Open Source Malware is a Nation-State Business Model: Attackers are exploiting high-trust open source ecosystems. Malware campaigns are increasingly optimized for developer workflows, targeting credentials, CI secrets, and build environments. State-linked activity shows that these tactics are not just opportunistic, they are strategic.

Vulnerability Intelligence is Failing at the Moment it Matters Most: Teams are trying to prioritize risk, but basic vulnerability data is often missing, late, or wrong. That creates triage failure, false confidence, and wasted effort. When the intelligence layer breaks, security programs cannot reliably separate what is urgent from what is noise.

Avoidable Vulnerability Consumption Persists: Even when fixes exist, vulnerable versions continue to be downloaded at scale. Set-and-forget dependencies, transitive sprawl, and upgrade friction keep old risk flowing into new builds. The problem is not awareness. It is workflow inertia and unclear ownership.

AI Accelerates Both Productivity and Security Risk: AI-assisted development is increasing the speed of dependency changes, but it can also introduce errors, such as selecting non-existent versions or unsafe packages. Without guardrails and verified sources of truth, AI turns small data quality issues into large-scale operational risk.

Transparency is Now a Mandate: Regulators and buyers are turning transparency into a requirement through SBOMs, attestations, and provenance expectations. Compliance is shifting from policy documents to build outputs. Organizations that operationalize transparency in CI/CD will move faster and face less friction.

sonatype

Registries, Models, and the New Software Infrastructure Burden

WHEN GROWTH MEETS GRAVITY

Open Source Scale Has Become a Structural Risk

Open source has entered an era where scale itself has become a structural risk. Package registries that once measured growth in millions of downloads now routinely serve trillions of requests. But this growth does not map cleanly to innovation. 2025 saw 9.8 trillion downloads across Maven Central, PyPI, npm and NuGet, but the majority of registry traffic today is not driven by new applications or meaningful reuse. It's driven by transitive dependency sprawl, unused or abandoned packages, and unsustainable tooling patterns.

9.8 TRILLION

downloads in 2025 across Maven Central, PyPI, npm and NuGet

FIGURE 1.1

Yearly Downloads over Time (Maven Central, PyPI, npm, and NuGet)

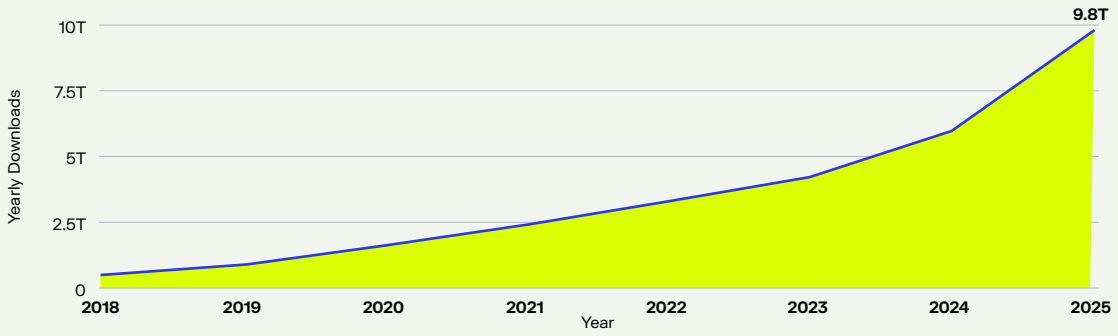


FIGURE 1.2

2025 Registry Growth

Ecosystem	2025 Total Components Added	Cumulative Total Components	2025 Total Releases Added	Cumulative Total Releases	2025 Downloads	YoY Download Growth Rate
Maven Central (Java)	260.5k	808.6k	3.3M	24.95M	839.05B	19.42%
PyPI (Python)	214.8k	821.3k	1.54M	8.85M	804.97B	50.64%
npm (JavaScript)	749.7k	5.59M	11.18M	65.56M	7.97T	65.43%
NuGet (.NET)	144.8k	760.1k	2.4M	14.02M	223.37B	17%

Modern CI/CD systems and ML pipelines are optimized for speed and convenience, not efficiency. Once configured, they pull relentlessly, often blind to redundancy, cost, or downstream impact. The result is a structural burden that registries were never designed to carry alone. Public software ecosystems are drifting toward a tragedy of the commons: a fraction of organizations and automated systems consume a disproportionate share of bandwidth and compute while registry operators and volunteer maintainers absorb the strain.

As software supply chains expand to include not just code, but models, datasets, and increasingly large artifacts, the question is no longer whether open software ecosystems can scale — but who pays for that scale, and how long the current system can hold.

Registry Consumption

maven central repository

19.42%
YoY download growth

3,312,376
releases added in 2025

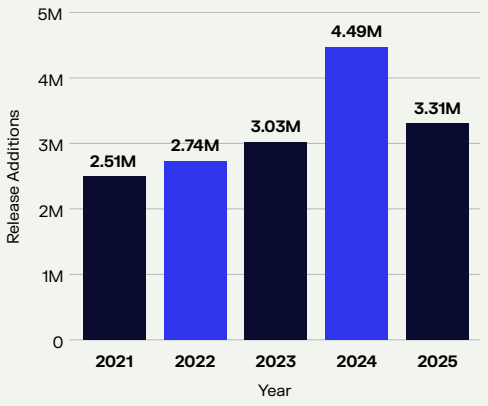
40%
of vulnerable releases were CVSS 9.0+ (Critical)

Maven Central underpins enterprise Java development, and its scale means small shifts propagate widely. In 2025, downloads grew 19.42% year over year, reinforcing Maven Central's role as a default dependency source across commercial and open source software. Maven's growth slowed slightly in 2025 due to sustainability measures put in place designed to limit usage at the highest end. At this volume, incremental growth still produces large absolute increases in consumption: new releases, regressions, and vulnerabilities can affect thousands of organizations quickly.

That impact is driven as much by release velocity as by new library creation. In 2025, more than 3.3 million releases were added, creating sustained upgrade and governance pressure for consuming teams. The operational challenge is less "what exists" and more how to evaluate and manage constant version change across dependencies already embedded in production portfolios.

Security data reinforces the need to prioritize vulnerabilities in dependencies and to steer toward the safest, fastest upgrades, not toward unused or test-only components. In 2025, 40% of vulnerable Maven Central releases carried CVSS 9.0+ scores, showing that severe issues are not rare. Teams can't control when vulnerabilities are introduced. But, at Maven Central's scale, success hinges on prioritization and speed, not additional alerts or manual reviews.

FIGURE 1.3
Maven Central Release Additions Over Time



python™ Package Index

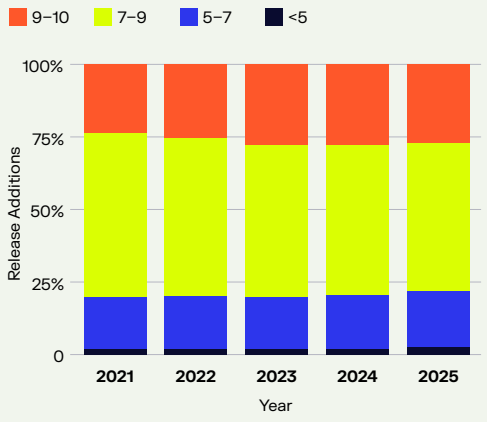
50.64%
YoY download growth

1 in 5
2025 releases were tied to CVSS 7.0+

~26%
of the total catalog made up by 2025 component additions

PyPI's growth underscores where developer adoption and dependency sprawl are accelerating most quickly. With 50.64% year-over-year download growth, PyPI reflects the surge of modern workloads tied to AI and cloud development. That velocity brings scale benefits, but it also shows early signs of stress.

FIGURE 1.4
Vulnerable PyPI Release by Severity Over Time



In 2025 alone, new component additions accounted for 26% of PyPI's total registry catalog, a striking indicator of how quickly the universe of available dependencies is expanding. Each new package increases choice and innovation, but it also multiplies evaluation and enforcement challenges. More components mean more potential entry points for risk and greater transitive exposure as teams pull in deep dependency trees they may not fully understand or monitor.

This level of growth and breadth comes with a clear security signal: risk is not an edge case. In 2025, one in five PyPI releases was associated with a CVSS 7.0+ vulnerability, showing that serious issues regularly flow through everyday pipelines. For organizations relying on PyPI, this makes proactive controls essential.

65.43%
YoY download growth

838,778
CVSS 9.0+ releases in 2025

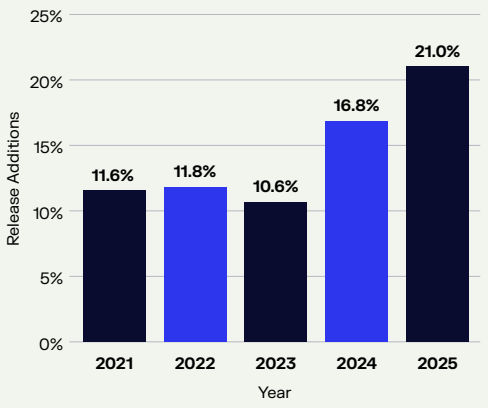
>60%
of all new releases (across these 4) were npm in 2025

In 2025, npm downloads grew 65.43% year over year, and the software ecosystem produced over 60% of all new releases across major registries. This combination of rising consumption plus dominant release volume means npm's impact is less about catalog size and more about release velocity: constant updates, republishing, and forks increase the rate of dependency change that consuming teams must evaluate and absorb. At this pace, traditional manual review and approval models do not scale.

In 2025 alone, npm recorded 838,778 releases associated with CVSS 9.0+ vulnerabilities, a number that reframes "rare" events into everyday realities. This scale is what enabled watershed incidents like React2Shell, discussed later in *The Three Layers of Failure in Modern Vulnerability Management* chapter, and Shai-Hulud to have ecosystem-wide impact. As detailed in the next chapter, *Malware at the Gate*, npm faced a number of self-replicating malware campaigns, which ultimately added 171,740 malicious packages to the registry over the span of a few months.

The takeaway is not blame, but scale awareness: when hundreds of thousands of 'Critical' releases exist in a single year, teams cannot rely on manual review or reactive patching. Automation, prioritization, and rapid upgrade motion are essential to keeping pace with an ecosystem where critical risk can now propagate as quickly as the code itself.

FIGURE 1.5
Rate of Vulnerable npm Releases Over Time



nuget

16.5
releases per new component in 2025

~0.8%
of 2025 vulnerable releases were below CVSS 5

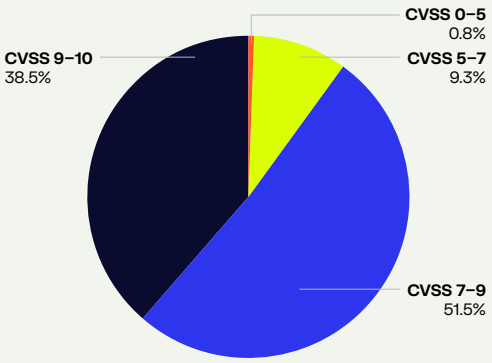
38.5%
of vulnerable releases in 2025 were CVSS 9.0+ (Critical)

NuGet may not generate the same headline-grabbing download spikes, but its cadence is distinctive. In 2025, NuGet averaged 16.5 releases per new component, pointing to rapid iteration and steady maintenance rather than pure catalog expansion. This level of churn signals active maintenance, frequent fixes, and continuous refinement, especially common in enterprise and platform-oriented .NET development. For consumers, the operational burden isn't discovering "new," it's tracking version changes across dependencies already in production.

The nature of risk within NuGet further raises the stakes of that churn: in 2025, less than 1% of vulnerable NuGet releases fell below CVSS 5, indicating the vast majority of flaws are not noise. At the extreme end, 38.5% of vulnerable NuGet releases were associated with CVSS 9.0+ vulnerabilities. Paired with rapid version turnover, ad hoc patching and manual decision-making quickly break down. What NuGet demands instead is fast, reliable remediation mechanics, including clear prioritization and automated upgrade workflows.

We are no longer just measuring growth; we are evaluating its impact. As consumption across this unified software supply chain accelerates, it forces a critical question. How much of this massive consumption is productive, driving genuine innovation and business value? And, more importantly, how much is unproductive waste that the software ecosystem can no longer afford to ignore?

FIGURE 1.6
2025 Vulnerable NuGet Releases



Real Innovation vs. Synthetic Volume

As software supply chains scale, the impacts of organic growth compared to synthetic growth are increasingly distinct. Understanding this difference helps organizations focus on what truly advances their capabilities and avoid unintentionally contributing to systemic strain.

ORGANIC GROWTH reflects real shifts in how software is built: AI adoption, cloud migration, and proliferating languages/frameworks increase dependency usage because teams are adding capabilities and moving faster. It raises complexity, but the added dependencies generally map to delivered functionality and business outcomes.

SYNTHETIC GROWTH inflates volume without comparable value. Spam publishing, incentive gaming, malware, and typosquatting can spike project and download metrics, while CI/CD misconfigurations (cold caches, always-clean builds, non-expiring mirrors) repeatedly re-download the same artifacts. The result is higher bandwidth and infrastructure cost — and more risk — without improving software quality.

For organizations trying to manage risk and cost at scale, the distinction matters. Synthetic volume obscures real signals, overwhelms governance processes, and amplifies exposure without delivering benefits. It also shifts burden onto public software ecosystems that were not designed to absorb limitless, redundant traffic.

The Commons is Cracking

Public registries are global distribution systems with real costs: bandwidth and CDN delivery on every download; storage and replication for every release; and ongoing investment in abuse response, malware scanning, moderation, incident handling, and security investigations. As open source expands beyond apps into software infrastructure, AI platforms, and model hubs, these operational demands keep rising.

THE SUSTAINABILITY PROBLEM ISN'T "TOO MUCH OPEN SOURCE," BUT RATHER CONSUMPTION AT MACHINE SCALE.

Automation multiplies load: CI pipelines repeatedly pulling the same dependencies, build systems re-resolving dependency graphs, and large organizations running thousands of parallel jobs. Similar patterns are emerging in AI and model hubs, where large artifacts are repeatedly fetched by automated workflows. Defaults built for convenience can turn routine activity into sustained, high-volume demand.

And the demand isn't evenly spread. A small number of consumers, tools, and patterns drive a disproportionate share of traffic, compounding costs, reliability strain, and exposure to abuse. When registries slow down, pause services, or absorb malicious floods, the impact ripples across entire ecosystems — from application development to critical software infrastructure and downstream AI platforms that assume constant availability.

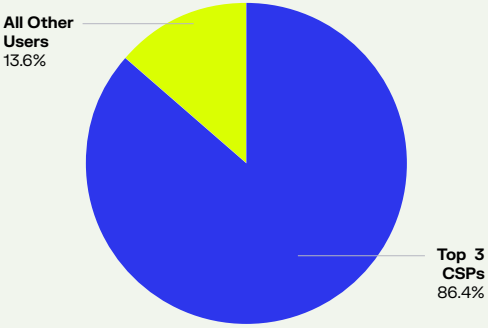
This isn't a story about one bad actor or one registry failing. It's an ecosystem-level mismatch between yesterday's defaults and today's machine-speed reality. Preserving the commons means updating consumption norms and shared responsibility. Ecosystem health now depends as much on how software is consumed as on how it's created.

THE IMPACT OF CLOUD PROVIDERS: WHERE THE LOAD CONCENTRATES

Cloud provider traffic now defines what "normal" looks like on Maven Central. In the latest snapshot, the top three cloud service providers (CSPs) accounted for more than 108 billion requests, while every other user combined represents around 17 billion. Taken another way, CSPs represent just 32.5% of IPs on Maven Central, yet account for more than 86% of downloads.

FIGURE 1.7

CSPs vs All Users: Breakdown of Maven Central Downloads



WHEN A SMALL SET OF CSPS BECOMES THE DOMINANT ACCESS PATH TO THE ECOSYSTEM, MAVEN CENTRAL EFFECTIVELY SERVES AS SHARED PRODUCTION INFRASTRUCTURE FOR CLOUD-NATIVE BUILD, DEPLOY, AND RUNTIME WORKFLOWS.

At that volume, small changes in cloud build behavior (ephemeral runners, cache churn, region replication, image rebuild loops, cold-start fleets) can translate into outsized swings in total registry load.

The implication for the commons is straightforward: registry strain is increasingly driven by automation at hyperscale, not broad-based organic growth. Improving cache persistence, tightening redundant fetch patterns, and designing "download once, reuse everywhere" behaviors inside cloud delivery pipelines becomes one of the highest-leverage ways to reduce systemic load — because the biggest consumers aren't "more developers," they're a few platforms operating at machine speed.

RE-DOWNLOAD OFFENDERS: THE BIGGEST AVOIDABLE BILL

47.5%
of the top 200 re-downloading organizations contained a single IP

20.5%
operate from more than 5 IPs

17%
exceed 1,000 re-downloads in one week

6%
exceed 5,000 re-downloads in one week

Re-downloads are where open source sustainability becomes concrete, because they represent repeat fetches that add load without adding new value. In the last seven days, the heaviest re-download activity is tightly concentrated: a large share of the top re-downloaders operate behind just one or a handful of IPs, pointing to centralized CI runners, shared egress gateways, or build fleets behaving like cold-start machines.

The sustainability implication is that avoidable strain on Maven Central is not evenly distributed across the ecosystem. It's driven by a relatively small set of automation patterns that scale — often inside a single organization — into repeated pulls of the same dependencies. That makes the problem unusually tractable: improvements like durable caching, correctly configured proxies/mirrors, and less "always-clean" dependency resolution can reduce outsized load quickly. Fixing one pipeline can remove pressure that would otherwise be multiplied across thousands of builds.

Overall, the story isn't "more developers are downloading more." It's that modern software delivery is optimized for speed and rebuildability. When cache persistence breaks down, the cost is externalized onto shared infrastructure. The path to sustainability is aligning build defaults with commons realities so the ecosystem can keep moving fast without turning every rebuild into unnecessary traffic.

HOW TO REDUCE REDUNDANT TRAFFIC WITHOUT SLOWING DELIVERY:

- ☑ Routing CI through repository managers or caching proxies
- ☑ Making build and dependency caches durable across runs
- ☑ Pinning and reusing dependencies where appropriate

FIGURE 1.8

Comparing Build Tools on Maven Central

	Maven	Gradle
Default "download behavior"	More cache-trusting for pinned versions leads to fewer repeat fetches	More cache-correct + frequent revalidation leads to more repeat GETs
Where it runs (typical)	Benefits from long-lived machines or / build nodes with warm local repos	Common in ephemeral CI/containers with cold caches each run
Why this matters at scale	Naturally dampens redundant traffic over time	Can amplify redundant traffic unless caching/CI reuse is strong
Best mitigation lever	Persist local/CI caches ("download once, use many times")	Durable build cache + CI artifact reuse to cut re-downloads

WHY BUILDS AMPLIFY LOAD: THE IMPACT OF TOOLS LIKE MAVEN AND GRADLE

Traffic patterns in large registries are not evenly distributed across countless clients — they are highly concentrated. In the case of Maven Central, just two build tools, **Maven and Gradle, account for 81.1% of all traffic**. This concentration creates outsized implications: small improvements in default behavior, caching strategies, or CI integration for these tools can materially reduce ecosystem-wide load without requiring millions of individual developers to change how they work. When the majority of consumption flows through a narrow set of tools, system-level optimizations become far more effective than relying on per-project best practices alone.

Maven and Gradle amplify registry load in different ways, not because they consume different artifacts, but because their configuration and caching models differ in practice. Gradle is engineered to be cache-correct and CI-friendly: it aggressively revalidates metadata, resolves dependencies in parallel, and is commonly run in short-lived agents or containers where caches start cold. Under normal circumstances, much of that extra traffic would be absorbed by a local caching proxy.

4.08X

more frequent re-downloads in Gradle than Maven

Inserting such a proxy consistently is nearly impossible to do at scale in Gradle because it lacks a strong, hierarchical inheritance model for repository configuration. That makes it difficult to centrally enforce a single caching endpoint without modifying every build or risking breakage. As a result, many Gradle builds effectively (and unintentionally) bypass local caches and hit upstream registries directly, amplifying repeat GETs for the same artifact URLs even when versions are pinned.

Maven, by contrast, has a simpler and more centralized settings model that makes proxying and mirroring straightforward. Combined with Maven's more cache-trusting behavior for fixed versions and its frequent use on long-lived machines with warm local repositories, this naturally reduces repeat downloads over time.

At scale, redundant downloads don't just consume bandwidth — they increase load on the services that keep registries safe and reliable (indexing, scanning, abuse detection, and incident response capacity). The practical goal is simple: download once, reuse many times. Teams can cut repeat fetches while improving build speed and reliability by adopting durable caches, shared artifact proxies, and CI patterns that preserve dependencies across runs.

DO NOW (FAST WINS)

- ☑ **Make CI caches durable** persist Gradle caches between runs
- ☑ **Add a shared artifact proxy / repository manager**
- ☑ **Stop "always-clean" defaults** keep dependency caches even if outputs are cleaned

DO NEXT (HIGHER LEVERAGE)

- ☑ **Standardize cache strategy across runners** consistent paths/keys
- ☑ **Instrument and enforce** track re-download rate; set guardrails
- ☑ **Reduce metadata churn** pin versions; use lockfiles where applicable

AI REGISTRIES AS THE NEXT STRESS TEST

AI registries and model hubs are the next major stress test for shared distribution infrastructure. They inherit package-registry behaviors such as automation, repeat pulls, and reuse, but with a much heavier cost profile. Models, datasets, and checkpoints are large by default, often hundreds of megabytes to several gigabytes, come in multiple variants, and change frequently as teams iterate. This drives higher bandwidth, storage, and replication demands.

The risk is not just artifact size. If AI usage follows today's norms such as CI re-downloads, weak cross-environment caching, and hotspot automation, load will escalate quickly. Inefficiencies that are tolerable for small packages become expensive and destabilizing at model scale, threatening availability and reliability.

> **THE TAKEAWAY IS THAT SCALE AMPLIFIES DEFAULT BEHAVIORS, SO SUSTAINABILITY MUST BE DESIGNED IN EARLY.**

Durable caching, artifact reuse, provenance-aware distribution, and AI guardrails to prevent unnecessary pulls are critical now, before AI ecosystems reach package-registry levels of global dependency.

What Responsible Consumption Looks Like

Growth across package ecosystems continues, along with the security and operational pressure that scaling creates. As registries grow, more responsibility shifts to consumers to reduce unnecessary load, limit exposure, and keep risk manageable. Responsible consumption is about maintaining developer velocity without increasing supply chain risk.

The biggest lever is architectural. Private repositories and intelligent caching should be the default. Letting CI pipelines pull directly from public registries on every build amplifies traffic, increases failure risk, and creates avoidable exposure during outages or tampering events. Centralizing dependency access through controlled repositories that cache, vet, and reuse artifacts across teams reduces churn, improves build determinism, and narrows the impact of upstream changes.

Architecture also needs guardrails. Organizations should set and enforce consumption policies that reflect real usage at scale, including limits on redundant downloads. SCA and repository management tools help by prioritizing used dependencies, de-duplicating artifacts across projects, and reducing noise from unused or unreachable components. The goal is focus, clearer signals, fewer alerts, and faster remediation.

Responsible consumption is also a shared software ecosystem issue. The heaviest consumers benefit most from public registry reliability, so long-term sustainability requires shared responsibility.

CHECKLIST: ARE YOU SUPPORTING THE SOFTWARE COMMONS?

Do you have internal policies or guidelines in place to minimize unnecessary artifact publishing or republishing?

Do you intentionally batch or optimize releases to avoid unnecessary registry strain?

Do you have policies or guidelines around publishing internal-only items to registries?

Do your CI systems use local caching or private repositories by default?

Do you know which registries your org depends on most, and how much traffic you generate?

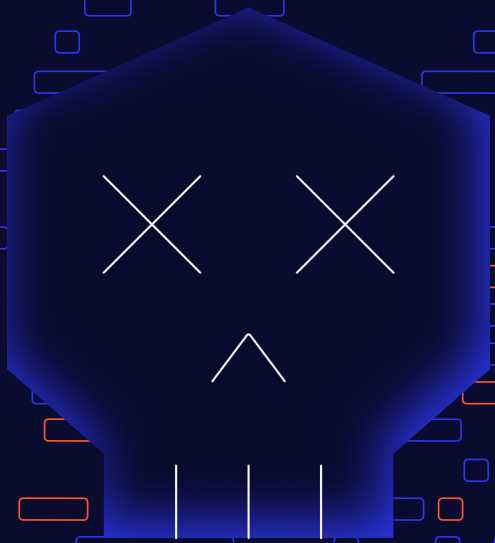
Do you distinguish between used and unused dependencies in your security and governance workflows?

Do you contribute (financially or in-kind) to the registries and OSS projects that are critical to your builds?

sonatype

The Evolving Software Supply Chain Attack Surface

MALWARE AT THE GATE



A Turning Point for Open Source Malware

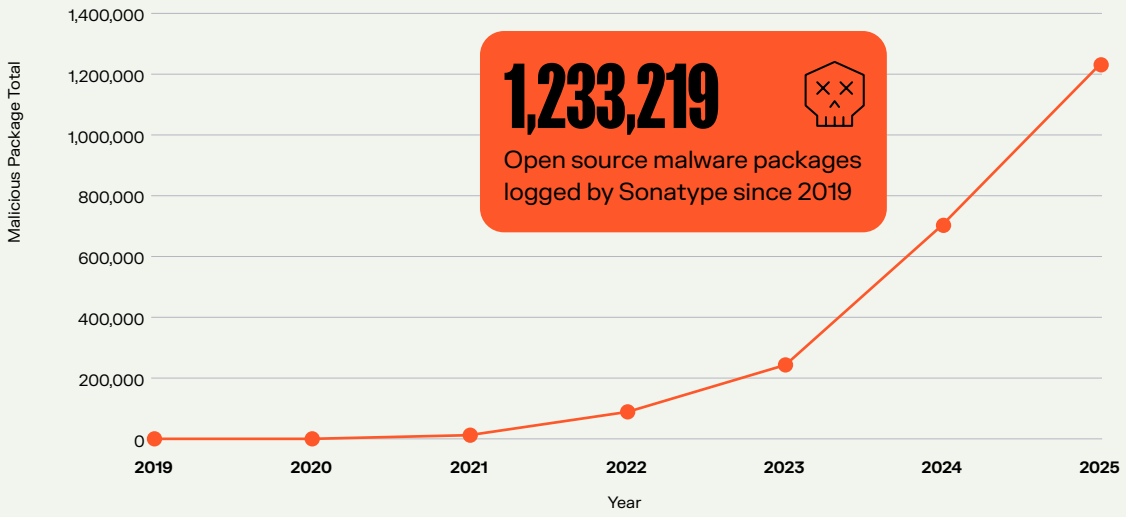
Throughout 2025, Sonatype identified more than 454,600 new malicious packages, bringing the cumulative total of known and blocked malware to over 1.233 million packages across npm, PyPI, Maven Central, NuGet, and Hugging Face. This year, we observed that the evolution of open source malware crystallized, evolving from spam and stunts into sustained, industrialized campaigns against the people and tooling that build software.

454,648

new malicious packages Sonatype identified in 2025

FIGURE 2.1

Annual Open Source Malware Growth



What stands out most about 2025 is not just the scale of the threat, but also the sophistication. Where 2024's XZ Utils incident was groundbreaking, demonstrating how a single compromised maintainer could imperil global infrastructure, 2025 saw software supply chain risk evolve dramatically.

This year, over 99% of open source malware occurred on npm. State-linked entities such as the Lazarus Group advanced from simple droppers and crypto miners to five-stage payload chains that combined droppers, credential theft, and persistent remote access inside developer environments. The first-ever self-replicating npm malware (Shai-Hulud, quickly followed by Sha1-Hulud) proved that open source malware can now propagate autonomously through open source ecosystems. IndonesianFoods created more than 150,000 malicious packages in just a couple of days. And a series of offensive hijackings of trusted packages like `chalk` and `debug` showed that established maintainers of high-profile packages are being targeted as entry points for mass distribution.

Taken together, these developments mark 2025 as a grim year for open source malware: the moment when isolated incidents became an integrated campaign, and bad actors proved software supply chain attacks are now their most reliable weapon.

The Threat Taxonomy: What Open Source Malware Does Today

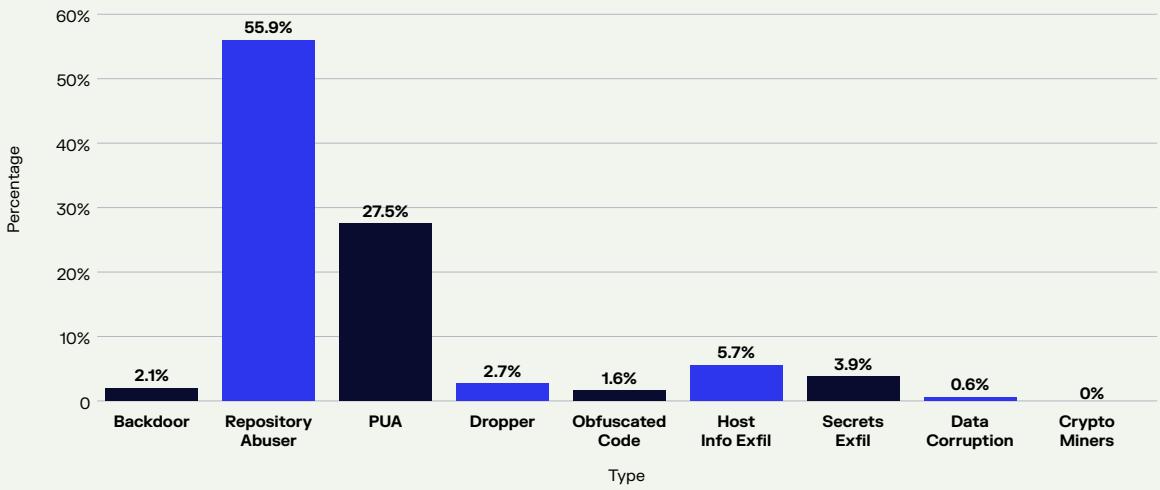
Open source malware is best understood less as a set of isolated "bad packages" and more as a set of repeatable behaviors that exploit how modern software is built and shipped. Public registries provide a low-friction distribution channel, while developer machines and CI/CD pipelines provide an execution environment that often sits close to sensitive data and production access. As a result, the malicious package is increasingly not the whole attack, but the first step in a larger supply chain intrusion.

REGISTRIES ARE BEING USED AS DISTRIBUTION PLATFORMS

In 2025, the dominant pattern is operational scale through ecosystem mechanics. Repository abuse shows up in 55.9% of all logged malicious packages, indicating actors are treating registries like platforms: automating publication and iterating quickly to maximize reach. Repository abuse packages have been observed harvesting TEA tokens or seeking clicks on spam links. Alongside that, Potentially Unwanted Application (PUA) appears in 27.5% of packages, which include items like empty packages, demos with hardcoded credentials, or messaging app spam bot orchestration frameworks. These are packages that don't necessarily compromise the developer who installs it or the application it is bundled into, but are still unwanted in developer environments.

FIGURE 2.2

2025 Landscape: Open Source Malware by Threat Type



DEVELOPER AND BUILD ENVIRONMENTS ARE THE PRIZE

A consistent objective is harvesting valuable data from where software gets built. Host information exfiltration appears in 5.7% of packages, and secrets exfiltration in 3.9%. These aren't the largest categories by volume, but they're high-leverage: packages run inside developer machines and CI/CD environments where tokens, API keys, and CI credentials are commonly present and reusable.

ATTACKS ARE ENGINEERED AS CHAINS, NOT SINGLE PAYLOADS

Sonatype observed clear signs of staged delivery and follow-on capability. Droppers/loaders appear in 2.7% of packages, and backdoors in 2.1%, with obfuscated code in 1.6% acting as a force multiplier that helps these chains persist and evade inspection. Even lower-volume disruption behaviors matter for impact: data corruption appears in 0.62% and targets build outputs and release workflows where compromise can propagate downstream.

DEVELOPERS ARE THE ATTACK VECTOR

Software supply chain attackers are perfecting social and technical mimicry to target and exploit developers making development decisions fast and with incomplete information.

Attackers increasingly rely less on individual mistakes and more on scale, momentum, and volume. They know developers under deadline pressure are unlikely to pay detailed attention on every dependency. If a package "looks right" with mostly comprehensible code, a legitimate seeming README.MD, and a reasonable amount of downloads, it is likely to get installed.

SOCIAL AND TECHNICAL MIMICRY TECHNIQUES

- **Typosquatting** and **namespace confusion** remain staple techniques, but they operate differently. Typosquatting relies on minor spelling variations of legitimate package names, counting on human error during installation. Namespace confusion exploits how package managers resolve dependencies across public and private scopes. This allows attackers to publish public packages with the same name as internal or expected dependencies, so they are inadvertently pulled into builds.
- **Toolchain masquerading** is accelerating. Rather than posing as generic utilities, malicious packages increasingly impersonate the everyday tools developers install reflexively: framework add-ons, build plugins, linters, scaffolding utilities, and migration helpers. These packages are designed to look like routine workflow dependencies, making them more likely to be installed without close inspection.
- **Front-end workflow lures** are especially common. Attackers cluster package names around high-velocity ecosystems and popular tooling where dependency decisions are frequent, repetitive, and time-boxed. In these environments, developers often add or swap dependencies rapidly, creating ideal conditions for malicious lookalikes to blend in.

How North Korea Weaponizes Open Source

The Lazarus Group, or APT38, epitomizes the 2025 malware shift from opportunistic to industrialized. Building on earlier research, Sonatype identified more than 800 Lazarus-associated packages this year, concentrated overwhelmingly in npm (97%). In practical terms, npm provides the fastest path from package publication to developer workstation because it does not require namespace validation and tooling prefers the latest versions. By concentrating activity there, Lazarus maximizes the likelihood that poisoned dependencies will be installed quickly, propagate through transitive dependency chains, and spill into build pipelines, CI/CD systems, and downstream production environments with minimal friction.

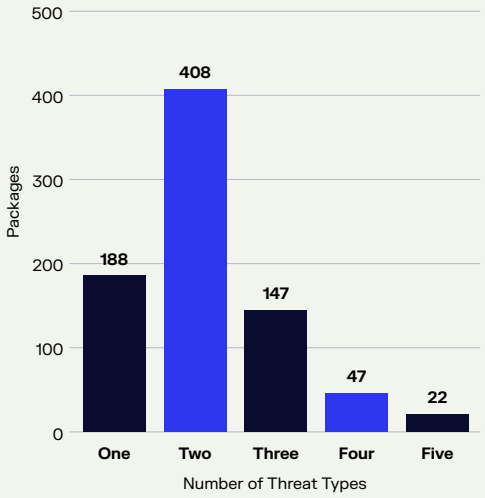
This level of sustained activity aligns with broader public reporting that cyber operations, including theft, espionage, and cryptocurrency-related crime, have become significant sources of revenue for the North Korean government. As a result, Lazarus now operates as one of the most prolific and successful state-sponsored cybercriminal enterprises in operation today. Lazarus is investing in ecosystems where speed, scale, and reuse combine to maximize the downstream impact of each compromised dependency.

HYBRID MALWARE DOMINATES THE LAZARUS PLAYBOOK

Lazarus packages are distinguished by how they integrate multiple threat behaviors into a single component. These aren't single-purpose nuisances; they're multi-function packages designed to support a staged intrusion chain. Sonatype Security Research observed that most

FIGURE 2.3

Lazarus Group Packages by Number of Threat Type

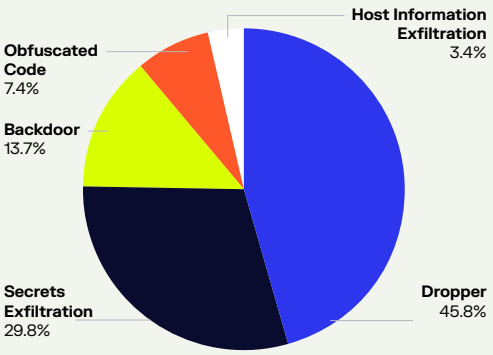


Lazarus packages carry multiple threat behaviors: roughly 77% include two or more threat types, and nearly 9% include four or more. In plain terms, the "package" is often just stage zero.

Behaviorally, the profile is dropper-led and credential-first: droppers appear in ~98% of packages, secrets exfiltration in ~64%, and backdoor functionality in ~29%. That combination matters. Droppers keep the published artifact small and less obviously malicious; exfiltration turns a single install into stolen tokens and credentials; and backdoor capability reflects investment in persistence and post-compromise control. The Lazarus pattern demonstrates repeatable intrusion tooling that is built to land quietly, harvest access, and remain useful after the initial foothold.

FIGURE 2.4

Lazarus Group Campaign Threat Types



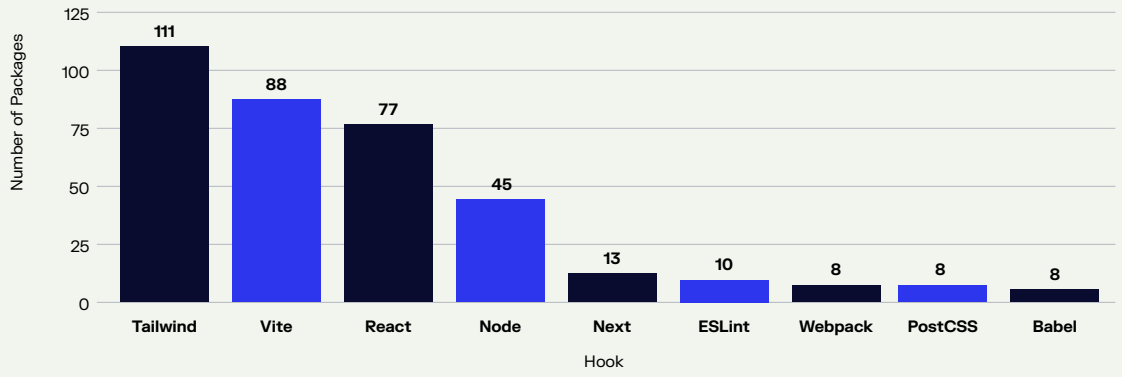
TARGETING IS OPTIMIZED TO EXPLOIT MUSCLE MEMORY

Lazarus targeting is engineered around how developers actually pick dependencies: familiar names, familiar ecosystems, familiar moments of need. These packages do not resemble overt threats; rather, these packages present as the routine glue of front-end workflows, such as framework add-ons, build helpers, plugin utilities, and configuration packages that developers install reflexively.

The naming patterns show deliberate clustering around high-velocity toolchains, such as Tailwind, Vite, and React. Zooming out, nearly 43% of Lazarus-linked packages reference common developer framework or tool keywords.

FIGURE 2.5

Top Lazarus Group Developer Lures



This is an intentional distribution strategy. These ecosystems have high dependency churn, many "one more plugin" installs, and constant troubleshooting under deadlines. That's the ideal environment for lookalike packages to blend in and get pulled into both workstations and CI. Sonatype's prior research showed that modern applications routinely contain hundreds of dependencies — averaging around 180 — making it unrealistic for developers to closely scrutinize every package they consume.

EXECUTION IS MODULAR AND REPEATABLE

One of the most important operational signals in Sonatype's analysis is how scalable the campaign was. The data shows strong indicators of templated reuse and rapid variant generation as opposed to one-off, bespoke malware. The distribution is sharply concentrated: Sonatype Security Research mapped 341 packages to a set of just 32 anchor packages, and the largest anchor clusters fan out into dozens of related variants.

That concentration is a direct indicator of manufacturing capacity: Lazarus can iterate quickly, generate families of near-neighbors, and keep publishing even as specific packages are identified and removed. In other words, this is not a handful of malicious uploads. It's a production line.

SHAI-HULUD: A NEW ERA OF SELF-REPLICATING MALWARE

The Shai-Hulud software supply chain attack in September 2025 marked a turning point: the first known self-replicating npm malware observed spreading autonomously across developer environments and packages, more like a traditional network worm than a passive library.

Hidden deep within duplicate files and nested directories, Shai-Hulud evaded superficial scans and leveraged maintainer credential theft to publish poisoned updates. The worm compromised more than 500 packages in days, spreading autonomously across registries and developer machines.

EACH SHAI-HULUD PACKAGE CARRIED A PAYLOAD DESIGNED TO:

- **Steal** npm authentication tokens,
- **Replicate** by infecting other locally linked projects, and
- **Exfiltrate** environmental credentials via encrypted payloads.

To support this, the attackers used public code-hosting services as dead drops, helping the traffic blend in with normal developer workflows.

The result was a rapidly self-propagating software supply chain worm, capable of infecting projects downstream without any manual publication step. This was quickly followed by another self-replicating npm malware in November, named "Sha1-Hulud: The Second Coming." These campaigns escalation illustrates the next phase of open source malware — one that behaves more like network worms than passive implants.

In contrast to traditionally-understood malware, which needed to be downloaded and installed before the malware would execute, open source malware executes pre-install, meaning developers only need to download in order to become a victim.

SELF-REPLICATING MALWARE IN 2025

September 16, 2025

Shai-Hulud | npm

500+ packages

The first documented self-replicating open source malware; demonstrated innovative use of automation by attackers to hijack accounts and publish new, malicious versions of legitimate packages.

October 17, 2025

Glassworm | OpenVSX and Microsoft VSCode

12 packages

Impersonated popular developer tools to steal credentials, drain cryptocurrency wallets, and use the Solana blockchain for command-and-control communication.

November 9, 2025

Glassworm | OpenVSX and Microsoft VSCode

3 packages

New malicious packages uncovered with 10,000 downloads using new extensions and publisher accounts to bypass cleanup efforts.

November 11, 2025

IndonesianFoods | npm

169,538 packages

This campaign was designed to self-replicate every seven seconds. While some packages abused the TEA protocol, most appeared designed to overwhelm detection and exploit ecosystem trust at scale.

November 24, 2025

Sha1-Hulud: The Second Coming | npm

49 packages

The hijacking campaign surged a second time with a new name and slight tweaks to evade detection; the attackers also introduced the use of Bun to deploy the payload.

December 1, 2025

Glassworm | OpenVSX and Microsoft VSCode

24 packages

In this third wave, the threat actors artificially inflated download counts of the packages to increase discoverability.

The Open Source Malware Supply Chain

Modern open source malware is modular, resilient, and designed to bypass both static and human inspection.

- **Multi-stage payloads:** Droppers download encrypted payloads from C2 servers or embed secondary stages locally.
- **Obfuscation layers:** Increasing use of eval(), encoded scripts, or disguised binaries within legitimate file trees.
- **Legitimate infrastructure for C2:** Slack, GitHub, Dropbox, cryptocurrency blockchain, and even logging services (like Better Stack) are co-opted for command-and-control traffic.
- **Local project propagation:** Recent attacks weaponize developer machines to infect all other projects they find and pushing infected versions upstream.
- **Multi-process behavior:** Telemetry from Sonatype's behavioral analysis indicates a rise in "multi-process modular malware," particularly in npm and PyPI.
- **Install-time execution:** The latest malicious packages run during installation, dropping payloads before builds.

The throughline shows malware is adopting the same modular architecture that makes open source so powerful. In 2025, software supply chain attacks mirrored the software supply chain itself. The risk is not theoretical. It's structural.

This phenomenon is especially visible in ML and DevOps contexts. MLOps is still a newer, less mature discipline, and it has not yet absorbed many of the supply chain lessons that became standard practice in traditional software development. Combined with intense pressure for rapid experimentation and deployment, teams often default to convenience-driven workflows that bypass normal governance.

In practice, that shows up as ungoverned "shadow downloads" that pull artifacts directly from wherever they are easiest to access. Examples include precompiled Python wheels and CUDA libraries fetched from unofficial sources, Hugging Face models loaded directly through package installs or runtime calls, and internal scripts or agents that silently retrieve dependencies from places like GitHub or Pastebin.

This mirrors the "Complacency and Contamination" model from the 10th Annual State of the Software Supply Chain report. Shadow downloads are the modern form of contamination, created when enforcement gaps intersect with developer convenience and automation.

HOW SHADOW DOWNLOADS COMPOUND OPEN SOURCE MALWARE RISK

- **Invisible:** Shadow artifacts often never appear in SBOMs or inventory systems.
- **Unscanned:** Because they bypass governed repositories, these artifacts frequently evade security scanning and policy enforcement altogether.
- **Unattributable:** With no verified origin or provenance, organizations have no reliable way to trust, trace, or audit what they've pulled in.

Emerging Threats

As AI becomes core to modern pipelines, attackers are following the trend, embedding malicious payloads into container images, AI models, and helper binaries distributed through trusted platforms.

MALICIOUS AI MODELS IN HUGGING FACE

Although many quarantined models observed to-date are not overtly nefarious, the underlying pattern reveals a structural weakness in model registries: model artifacts are being treated like data and scanned as single items, but in reality, most behave more like code and can be treated much the same way.

Sonatype's research into picklescan vulnerabilities underscored why this is uniquely dangerous in ML: widely used serialization formats can execute code during deserialization, turning a routine "load model" step into an execution path.

It's important to note the shape of the malicious activity observed on Hugging Face: many of these repositories appear consistent with security research or proof-of-concept demonstration uploads rather than fully operational criminal campaigns. Some are transparently labeled as unsafe, and several show low download counts. That doesn't reduce the underlying software supply chain risk, but rather highlights it. In a model registry, even a "demo" artifact can be copied, repackaged, or pulled into the wrong environment, and the consequences play out at runtime.

Two examples illustrate why this matters:

- **Backdoored model artifacts enabling remote access.** A cluster of models published under the same account exhibited behavior consistent with establishing a reverse shell to an external host, granting an attacker interactive access to any machine that loads the model. Even when download counts are low, the risk is disproportionate: models are frequently pulled into shared environments (developer workstations, notebooks, CI runners, GPU boxes) where credentials and tokens are plentiful.
- **Embedded malicious code in serialized model files.** In another case, a model artifact (a serialized file) contained embedded malicious logic that invoked common system tooling to exfiltrate local files (for example, transmitting /etc/passwd to a remote endpoint). The key point isn't the specific file targeted — it's the mechanism: a "model download" can become code execution at load time if organizations treat model artifacts as inherently safe.

MODEL REGISTRIES NEED THE SAME SUPPLY-CHAIN GUARANTEES AS PACKAGE REGISTRIES, BECAUSE THE BLAST RADIUS OF A COMPROMISED MODEL OFTEN INCLUDES THE VERY SYSTEMS THAT HOLD THE HIGHEST-VALUE SECRETS.

AI AGENTS AS SOFTWARE SUPPLY CHAIN ATTACK MULTIPLIERS

AI development assistants and autonomous agents have rushed into developer workflows, but the integration of those agents into their security models has not happened. Experiments show agents have a knowledge cut-off date well in the past, resulting in them happily installing whatever dependency resolves a build error without checking provenance, policy, or known-malicious indicators.

In the *From Guesswork to Grounded* chapter, we will show that AI code assistants like Claude or ChatGPT can fetch and install malicious code automatically when prompted to fix dependency errors or install missing libraries. The developer's intent may be harmless, but the result can be catastrophic.

Attackers are increasingly preying on this. Sonatype's 2025 malware research continues to document deceptive naming patterns — including typosquatting and new evasion tactics that mimic legitimate dependencies to trick developers into installing malware. As organizations integrate AI coding assistants into production workflows, they must recognize that these systems are not neutral intermediaries. They are potential infection vectors.

How Will Software Supply Chain Attacks Evolve?

The next frontier of software supply chain attacks is not limited to package managers. AI model hubs and autonomous agents are converging with open source into a single, fluid software supply chain — a mesh of interdependent ecosystems without uniform security standards.

Malware authors already understand this convergence. They are embedding persistence inside containers, pickled model files, and precompiled binaries that flow between data scientists, CI/CD systems, and runtime environments.

> **DEVELOPERS ARE NO LONGER AT THE PERIMETER. THEY ARE THE PERIMETER.**

sonatype

The Three Layers of Failure in

MODERN VULNERABILITY MANAGEMENT

The Limits of Modern Vulnerability Management

Modern vulnerability management is struggling to keep up with the rapid evolution of the software it aims to protect. It's not a single tool, team, or workflow that's failing, but the entire system that allows open source vulnerabilities to exist.

Despite major investment in scanning tools, disclosure pipelines, and security automation, organizations continue to operate with blind spots large enough for systemic risk to take root. Our analysis shows this failure compounds across three breakpoints in the software ecosystem, each breaking in its own way, and each amplifying the others.

THE GROWING INTEGRATION OF AI INTO SOFTWARE DEVELOPMENT IS ONLY EXACERBATING THIS CHALLENGE.

THE DATA LAYER

The Data Layer consists of various elements within the global vulnerability intelligence system: CVE (the Common Vulnerabilities Enumeration program), NVD (the National Vulnerability Database), and the advisory pipelines around them. Elements in this layer are increasingly incomplete, inconsistent, and slow. Coverage gaps, inaccurate version data, and long scoring delays distort how risk is understood and prioritized by both humans and AI.

THE CONSUMPTION LAYER

The Consumption Layer describes any activities related to importing open source software. Even when accurate data and patches exist, organizations continue to download and deploy vulnerable components. Dependency pinning, sprawling transitive graphs, outdated CI images, and ungoverned AI-generated component selection all reinforce the reuse of insecure versions. AI tools can only make recommendations that are as up-to-date as their training data. Much of today's risk arises not from new exploits, but through persistent poor consumption habits.

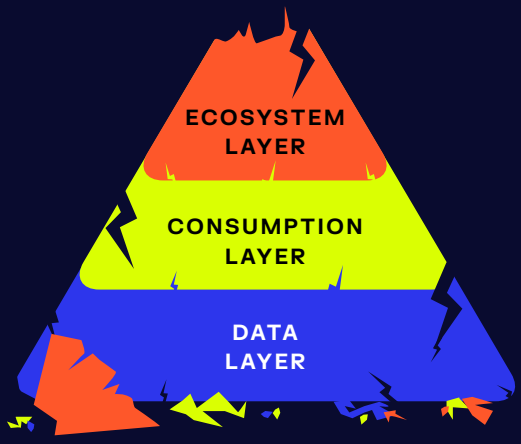
THE ECOSYSTEM LAYER

The Ecosystem Layer encapsulates the myriad of decisions that must be made for long-lived projects with open source dependencies. A growing share of software now depends on unsupported or end-of-life (EOL) releases. These components receive no patches, making vulnerabilities permanent. Legacy frameworks, abandoned libraries, and orphaned versions accumulate as long-term technical debt, leaving organizations dependent on software that cannot be secured through traditional remediation.

This chapter quantifies where the system breaks down, and outlines what a modern vulnerability management model must look like in a world where software moves far faster than the legacy processes designed to safeguard it.

ACCUMULATED VULNERABILITY DEBT

- **Data Layer:** Incomplete, inaccurate, and delayed public vulnerability intelligence
- **Consumption Layer:** Developers, AI, and pipelines keep pulling vulnerable components
- **Ecosystem Layer:** Dependence on EOL and abandoned components locks in permanent risk



The Data Layer Is Breaking Down

Modern threat and vulnerability management relies on the global intelligence ecosystem, anchored by the CVE program, NVD enrichment data, and upstream advisory pipelines that feed them. But that foundation is no longer consistently reliable. Coverage gaps, inconsistent metadata, delayed scoring, and missing ecosystem context now distort the very signals organizations depend on to assess and prioritize risk. When the underlying data is incomplete or wrong, every downstream decision, whether by humans, scanners, or AI, starts from a flawed premise.

Sonatype Security Research analyzed more than 1,700 open source CVEs throughout 2025 to understand where the gaps lie, and how they are impacting software development and security teams.

COVERAGE COLLAPSE

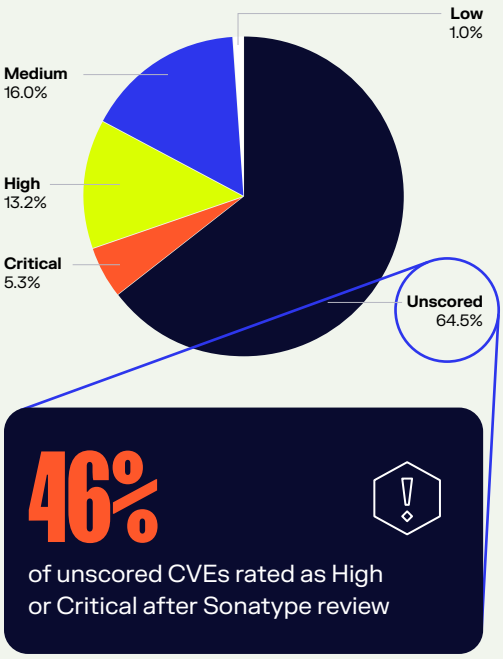
The first warning sign is the growing gap in basic CVE coverage. Nearly 65% of open source CVEs lack an NVD-assigned CVSS score, leaving most open source vulnerabilities without an official severity rating. That means that only about 600 open source vulnerabilities last year could effectively be triaged. When Sonatype assigned scores to these unscored CVEs, 46% were actually High or Critical, meaning many serious vulnerabilities enter the ecosystem without any meaningful prioritization signal.

IN 2025, ENTERPRISES COULD ONLY TRIAGE 35% OF VULNERABILITIES IF RELYING ON PUBLIC CVE DATA.

This problem is accelerating. In just five years, the global CVE count has doubled, yet the number of unscored CVEs has increased 37×, overwhelming a system built for manual processing and slower software cycles. As volume grows, the gap widens — leaving defenders without the baseline CVE data they rely on to triage risk effectively.

FIGURE 3.1

NVD-Assigned Severity of 2025 Open Source CVEs

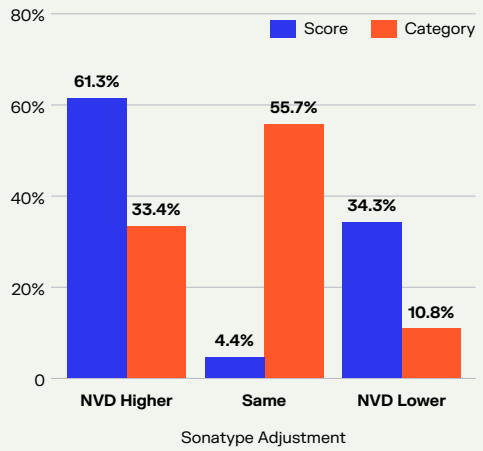


ACCURACY FAILURES

Even when scores exist, they're inconsistent enough to drive different outcomes depending on which feed you trust. Compared to Sonatype scoring and analysis, exact CVSS score matches are rare (4.4%), and severity categories align only 55.7% of the time. This means 44% of CVEs land in a different bucket in NVD versus Sonatype. The direction of the drift is usually upward in NVD: 61.3% of NVD scores are higher than Sonatype, compared with 34.3% that are lower.

FIGURE 3.2

Severity Score and Category Adjustments



1 IN 7

NVD-scored CVEs differ from Sonatype by 3+ CVSS points

20,362

false positives identified by Sonatype

167,286

false negatives identified by Sonatype

Sonatype identified 20,362 false positives, or packages incorrectly marked as vulnerable, creating noise in vulnerability management workflows and wasting developer time, and 167,286 false negatives, meaning exploitable components went unflagged entirely. The result is a vulnerability intelligence ecosystem that misleads both developers and security teams, forcing organizations to spend time on issues that don't exist while overlooking those that do. Inaccurate data also biases AI-driven tools, which use this information to determine dependency selection, upgrade paths, and remediation recommendations.

DELAYS THAT BREAK DEFENSES

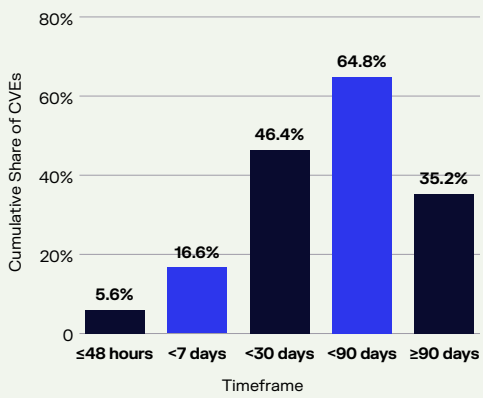
In 2025, the NVD's median time-to-score for open source CVEs was 41 days, with some taking up to a year. Meanwhile, exploit proof-of-concepts and maintainer patches frequently appear within hours. This growing lag renders "official" vulnerability information increasingly stale. By the time a CVE receives a severity score, the vulnerability may already be exploited in the wild, patched upstream, or both. Organizations relying exclusively on NVD data become effectively blind during the period when fast action matters most.

35%

took more than 3 months to receive a complete NVD record

FIGURE 3.3

NVD Time-to-Analysis of 2025 Open Source CVEs



EVEN MINOR METADATA INACCURACIES CREATE OUTZIDED REAL-WORLD CONSEQUENCES:

- **Incorrect vulnerable version ranges** generated thousands of false positives, overwhelming downstream scanners.
- **Wrong component identifiers** resulted in silent false negatives — packages with real vulnerabilities passed security checks unflagged.
- **EOL versions omitted from advisories** gave organizations a false sense of security, masking risks that upstream maintainers no longer track.
- **These cases reveal a systemic issue:** the CVE system excels at naming vulnerabilities, but struggles to describe them reliably enough for automated decision-making.

AI AS A FORCE MULTIPLIER FOR BAD DATA

AI-assisted development tools — increasingly embedded across coding, build, and remediation workflows — amplify the weaknesses of the data layer. Large language models are trained on public CVE and NVD data and treat it as authoritative even when it is incomplete, outdated, or incorrect. This impact is compounded when using an older model. As a result, AI does not fix bad data, but rather distributes it faster, which is examined closer in the *From Guesswork to Grounded* chapter.

The data layer is the foundation of threat and vulnerability management, yet today it is the least reliable part of the system. Incomplete coverage, inaccurate metadata, long scoring delays, AI amplification, and shadow download blind spots collectively undermine the ability of organizations to recognize and respond to real risk. When the data layer fails, every subsequent decision — what to fix, when to fix it, and how to prioritize it — begins from the wrong premise.

Poor Consumption Patterns Sustain Avoidable Risk

Even when vulnerability data is accurate and patches are readily available, risk persists because of how organizations actually consume open source. Dependency pinning, transitive pull-ins, outdated build images, and AI-generated manifests all keep vulnerable components in circulation long after fixes exist. In practice, a large share of modern vulnerability exposure is not created by new flaws — it is sustained by repeated reuse of old ones.

LOG4SHELL: THE CASE THAT SHOULD HAVE CHANGED EVERYTHING — BUT DIDN'T

Log4Shell was expected to be the turning point: the moment the industry collectively learned to upgrade quickly, retire vulnerable components, and modernize dependency practices. Four years later, the data tells a different story: the remediation path is well-understood and non-breaking, the open source vulnerability is universally recognized, and yet vulnerable versions continue to circulate at scale.

> **IN 2025 ALONE, DEVELOPERS DOWNLOADED MORE THAN 42 MILLION VULNERABLE VERSIONS OF LOG4J, REPRESENTING 13% OF ALL LOG4J DOWNLOADS WORLDWIDE.**

Regional patterns make the problem even clearer. While some markets have driven vulnerable Log4j usage down to single digits, others continue to pull 20–45% vulnerable versions, suggesting deeply uneven adoption of safe releases and persistent reliance on outdated build templates, pinned versions, or ungoverned transitive dependencies.

Log4Shell should have eliminated any doubt about the cost of running outdated open source. Instead, it revealed how ingrained consumption habits can be — and how long vulnerable code can persist, even when every incentive exists to move away from it.

THE BROADER PATTERN: JAVA'S TOP UNNECESSARY RISKS

Log4Shell remains the most visible example of "avoidable" vulnerability exposure, but it is not the dominant driver. Taking a broader look at the Java ecosystem, Sonatype analyzed the most frequently downloaded components that contained a vulnerability, despite a fix for that vulnerability already existing. The same consumption pattern repeats across the ecosystem: the vast majority of vulnerable components being downloaded already have a safer version available. The 10th Annual State of the Software Supply Chain Report found that roughly 95% of vulnerable component downloads had a fix on the shelf, while only ~0.5% represented true edge cases with no upstream path forward.

The most concerning signal is how frequently well-known vulnerable releases persist years after fixes are released. The Java ecosystem provides clear examples: widely used libraries with long-available patches still see substantial (and in some cases overwhelming) consumption of vulnerable versions. This is "unnecessary risk" in its purest form: risk that organizations continue to import into new builds even when safer versions are readily available.

Sonatype took a closer look at four vulnerable component versions with released fixes that, combined, represent a total of nearly 1.8 billion avoidable vulnerable downloads in 2025.

These packages share three characteristics: (1) at least one disclosed vulnerability, (2) a published fix, and (3) low adoption of the fixed line. The reasons are rarely dramatic. They're structural: pinned versions copied across services, transitive dependency blind spots, upgrade friction (especially across major versions), and selection signals that reward familiarity over maintainability.

FOUR VULNERABLE COMPONENT VERSIONS WITH RELEASED FIXES

Component	Vulnerable version(s) still widely consumed	Fixed version available	% of 2025 avoidable vulnerable downloads	Representative CVE(s)	Why it persists (consumption drivers)
commons-compress	1.21	1.26 (Feb 2024)	46.32%	CVE-2012-2098, CVE-2024-26308, CVE-2020-1945, CVE-2024-25710, CVE-2021-36374	Deeply embedded in build/packaging workflows; low "visibility" dependency; upgrades deferred unless forced.
commons-lang	2.6 (legacy major line)	3.18.0 (Jul 2025)	99.88%	CVE-2025-48924	Major-version migration is non-trivial (2.x → 3.x); older enterprise stacks remain pinned to legacy APIs.
snappy	0.4	0.5 (May 2024)	99.58%	CVE-2024-36124	Common in distributed platforms (e.g., Hadoop/Spark ecosystems) where low-level compression deps are pinned for stability/performance.
jdom2	2.0.6	2.0.6.1 (Dec 2021)	57.73%	CVE-2021-33813	Widely reused XML utility; upgrade inertia and "if it isn't broken" maintenance norms keep vulnerable lines circulating.

WHY TEAMS KEEP DOWNLOADING OPEN SOURCE VULNERABILITIES

If patches exist and the risks are well-known, why do vulnerable components continue to flow into modern software at such scale? The answer lies not in malicious intent, but in the quiet, structural habits of software development. Collectively, these patterns mean vulnerable components remain in circulation, not because teams are unaware of the risk, but because the system makes unsafe choices easier than safe ones.

THE SYSTEM MAKES UNSAFE CHOICES EASIER THAN SAFE ONES.

SET-AND-FORGET DEPENDENCIES

A version gets pinned once and then copied forward across services for years.

THE RESULT:
Changing dependencies feels risky; leaving them alone feels “safe.”

TRANSITIVE DEPENDENCIES + UNCLEAR OWNERSHIP

Vulnerabilities arrive via the dependency tree, not direct installs.

THE RESULT:
No single team feels accountable for buried upgrades.

TOOLING THAT SHRIEKS BUT DOESN’T STEER

Scanners generate long CVE lists without clear prioritization or safe upgrade paths.

THE RESULT:
Teams hit alert fatigue and avoid “break the build” upgrades.

INCENTIVES FAVOR FEATURES OVER HYGIENE

Maintenance work is deferred unless there’s a fire drill.

THE RESULT:
Delivery is rewarded; dependency upkeep is invisible..

AI EXACERBATES VULNERABLE CONSUMPTION

AI-assisted development tools are increasingly embedded across modern software workflows — from code generation and dependency selection to build configuration and remediation guidance. While these tools can accelerate delivery, they also inherit and amplify the same consumption patterns that already sustain vulnerability risk. AI amplifies vulnerable consumption in several predictable ways:

1. AI suggests "popular" (historically common) versions, not secure ones.
2. AI generates manifests with outdated/vulnerable components.
3. Training data lags, so even after fixes exist, AI keeps suggesting vulnerable versions.
4. Without governance, AI increases component sprawl.

AI does not introduce new vulnerability classes, but it accelerates existing consumption behavior. When unsafe versions are already easier to consume than safe ones, AI makes those unsafe choices faster, more repeatable, and harder to unwind. Most vulnerability risk is no longer a vulnerability discovery problem. It's a consumption behavior problem, and AI scales that behavior by default.

When the Ecosystem Stops Maintaining Your Software

Even with accurate vulnerability intelligence and disciplined dependency practices, some risks cannot be mitigated because the software itself is no longer maintained. A growing share of open source components now lives on EOL, or abandoned release lines, where no patches will ever be issued and new open source vulnerabilities may never be disclosed. These dependencies create permanent exposure: organizations inherit flaws that cannot be remediated upstream, locking long-term risk into the foundation of their software.

To analyze how End-of-life (EOL) dependencies turn vulnerabilities into persistent risk, we partnered with HeroDevs to examine the security impact of EOL software across modern software supply chains.

AI DOES NOT INTRODUCE NEW VULNERABILITY CLASSES, BUT IT ACCELERATES EXISTING CONSUMPTION BEHAVIOR.

EOL SOFTWARE IS NOT AN EDGE CASE

EOL software is often discussed as something a mature program will eventually "clean up." But data and analysis from HeroDevs suggests the opposite: EOL dependencies are a structural flaw of modern enterprise stacks, showing up consistently across ecosystems and persisting over time.

EOL changes the risk model. A measurable share of open source vulnerabilities now fall into a category that traditional remediation workflows cannot resolve. For these components, "scan → ticket → patch" stops being a workflow and becomes a backlog generator.

5–15%

of components in enterprise dependency graphs are EOL, meaning EOL exposure is present even when teams believe they are only using supported top-level libraries.

81,000+

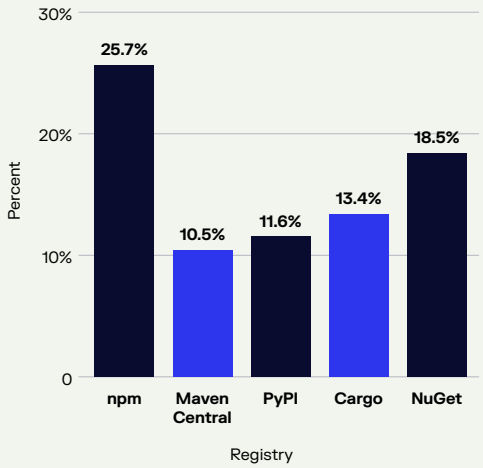
package versions with known CVEs are both EOL and unpatchable. HeroDevs estimates this number may actually be 400,000 across all registries.

ALL EXPOSED

EOL exposure appears across all major ecosystems (Java, Python, npm), with little variation in long-term persistence, suggesting this is not limited to one language community or a single package manager.

FIGURE 3.4

Breakdown of EOL Components by Registry



WHY EOL ALLOWS "FOREVER VULNERABILITIES"

Most vulnerability programs assume a predictable lifecycle: issues are disclosed, fixes are released, and risk declines as organizations patch and upgrade. EOL status breaks that logic. Once a release line is out of maintenance, upstream fixes stop, and a vulnerability can persist indefinitely — not simply because teams are slow to respond, but because the ecosystem no longer provides a patch path. At the same time, advisory coverage often degrades for unsupported versions, creating blind spots where EOL exposure is undercounted or missed entirely. And because abandoned code is reviewed less, fewer issues may be found or disclosed, so "no CVE" can indicate low scrutiny rather than safety.

In practice, EOL turns ordinary defects into "forever vulnerabilities": liabilities that cannot be resolved through routine patching and instead require major upgrades, replacements, or commercial backports. AI-assisted development can amplify this effect by steering teams toward what is most common in historical code rather than what is currently supported. EOL components often appear "popular" in public corpora, making them more likely to be suggested and adopted as defaults in AI-generated manifests. Once introduced, those patterns can replicate across services through reuse, reinforcing dependence on software that has no viable long-term remediation path.

AI REINFORCES EOL RISK IN PREDICTABLE WAYS:

1. AI models recommend EOL components because training data reflects **historical prevalence**, not current support status.
2. EOL packages often appear "popular" in public code corpora, creating **insecure defaults** in AI-generated manifests.
3. Once introduced, EOL dependencies are **self-replicating**: AI reuses prior code patterns, deepening organizational reliance on abandoned software.

EOL IN THE WILD: LOG4SHELL AND OTHERS

EOL is not just a theoretical lifecycle concern. It has measurable real-world impact during major incidents. Log4Shell illustrates how EOL status can prevent closure even when a fix exists in maintained branches. Real-world cases show how EOL obstructs remediation:

- **14% of Log4j artifacts affected by Log4Shell are now EOL, representing more than 619 million downloads in 2025,** preventing closure even four years later.
- Widely deployed major versions of Java, Node.js, Python frameworks, and .NET libraries continue to see **active download volume despite being unsupported.**
- CVE coverage for these versions is often **incomplete or missing,** reinforcing misleading "clean" scan results, especially when advisories and scanners focus on supported release lines.

This is how "known vulnerabilities" become "persistent exposure." Even if engineering teams upgrade where they can, long-tail EOL usage can keep a vulnerability class alive in production fleets, especially in large enterprises with diverse portfolios, legacy workloads, and inherited dependency trees.

THE BACKPORT ECOSYSTEM

As EOL exposure becomes unavoidable, a secondary market has emerged to provide what upstream maintainers no longer can: security patches for unsupported release lines. This ecosystem is both a pragmatic mitigation path and a signal of structural fragility in open source lifecycle guarantees.

These programs can reduce risk when modernization is not immediately feasible. But they also underscore a core shift: for a meaningful share of enterprise dependencies, patchability is no longer guaranteed by the open source ecosystem itself. Organizations must plan for lifecycle continuity as a security requirement, not a best practice.

A GROWING RESPONSE ECOSYSTEM INCLUDES:

1. **Commercial extended-support providers** that backport security fixes (and sometimes ship compatible, maintained forks).
2. **Smaller specialist vendors and consultancies** that produce targeted patches for older release branches.
3. **Community-maintained forks** that temporarily sustain patching.

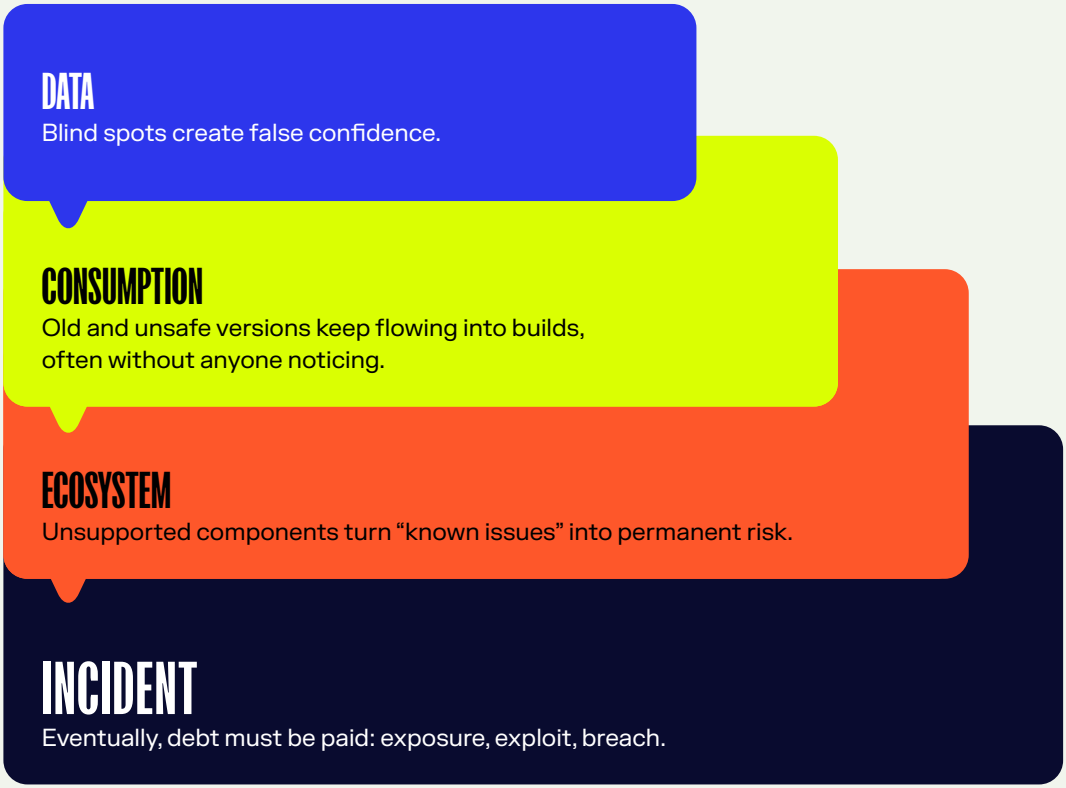
How the Three Layers Compound Each Other

Together, these failures create structural vulnerability debt, or risk that accumulates faster than it can be identified, triaged, or patched. Traditional "find and fix" workflows, centered on CVE identifiers and remediation queues, cannot keep pace with this reality. When the data is incomplete, consumption is undisciplined, and the ecosystem is aging, security becomes a reactive discipline rather than a strategic one.

Modern vulnerability risk is not the product of a single failure point. It is systemic, emerging from the way multiple weaknesses interact across the SDLC. When viewed in isolation, each layer appears manageable. When combined, they create a feedback loop that sustains risk even in organizations with mature security programs. The result is not a backlog problem but a structural one:

- **Long-term residual risk** persists across software lifecycles, surviving refactors, rebuilds, and even organizational change.
- **Attack windows widen** as vulnerable and EOL components accumulate faster than teams can identify, prioritize, and remove them.
- **Remediation pipelines fall behind** dependency sprawl, generating more work than existing security and engineering capacity can absorb.
- **Compliance artifacts drift from reality.** SBOMs, audit reports, and scan results increasingly reflect what tools can see, not what software actually runs, especially when shadow downloads, or artifacts that are pulled into development without the use of a repository manager, bypass formal governance.

HOW VULNERABILITY DEBT ACCUMULATES



This is why vulnerability management feels increasingly ineffective, even as tooling improves. The system is optimized to find and fix individual vulnerabilities, while the risk itself is produced by how software is sourced, reused, and aged over time. When bad data feeds unsafe consumption, and unsafe consumption feeds unpatchable software, remediation alone cannot catch up. Organizations accumulate vulnerability debt, not because teams are inattentive, but because the system allows risk to enter faster than it can be retired.

Modernizing Vulnerability Management

The issues outlined in this chapter are not the result of insufficient effort or tooling, but the product of workflows designed for a slower, simpler software ecosystem. Addressing modern vulnerability risk requires modernization, not acceleration of legacy "find-and-fix" models.

To meaningfully reduce vulnerability debt, organizations need to move beyond CVE-by-CVE remediation toward lifecycle-based modernization and governance. In practice, reducing risk increasingly means addressing structural weaknesses: improving the fidelity of vulnerability intelligence, making safe dependency intake the default, and proactively migrating away from EOL components that have no future patch path.

This shift is necessary because vulnerability risk is now systemic rather than isolated. Modern vulnerability management often fails at the system level, constrained by weak data quality, inefficient consumption patterns, and the compounding effects of aging software foundations. The data layer, in particular, is increasingly misaligned with real-world exposure: coverage gaps, inaccurate metadata, and delayed scoring distort prioritization, waste remediation effort, and obscure material risk.

At the same time, the ecosystem itself is aging in ways that create durable exposure. EOL and abandoned components transform open source vulnerabilities into long-term liabilities that cannot simply be patched away; they must be modernized out of the environment or supported through alternative maintenance models. AI increases the urgency of this modernization agenda.

> **WITHOUT GOVERNANCE, AI CAN AMPLIFY EACH FAILURE MODE, MAKING LIFECYCLE MODERNIZATION, NOT CVE TRACKING ALONE, THE ONLY SUSTAINABLE PATH FORWARD.**

To reduce structural vulnerability debt, organizations must correct weaknesses across all three layers of the system: **data**, **consumption**, and **ecosystem**. And, with increasing integration of AI into software pipelines, reducing this risk has never been more critical.

Layer	Key Actions	Primary KPI
DATA LAYER	• **Enrich CVE/NVD:** leverage data from OSV.dev, GitHub Security Advisories, upstream maintainers, and commercial intel. • **Add decision context:** accurate version ranges, exploitability signals, and EOL status. • **Improve identification:** fingerprint shadow-downloaded artifacts and feed curated data into AI systems.	**False negative rate / coverage gaps** (missed vulnerable or EOL components due to incomplete intelligence).
CONSUMPTION LAYER	• **Block by default:** repository firewall + policy controls for known-vulnerable versions and shadow downloads. • **Standardize safe inputs:** golden images, dependency templates, internal catalogs/allowed versions. • **Automate hygiene:** PR bots + continuous refresh with compatibility-aware upgrades; govern build agents/AI to approved sources.	**Avoidable exposure** = % of downloads/builds using vulnerable versions when a fix exists.
ECOSYSTEM LAYER	• **Treat EOL as critical:** detect, prioritize, and remove unsupported components. • **Define exit paths:** major upgrades, framework transitions, retirement plans. • **Reduce provenance risk:** eliminate unsupported shadow binaries; use extended-support backports only as transitional controls; surface lifecycle status in SBOM/risk scoring.	**EOL footprint** = % of components (or builds) on unsupported release lines.
AI WITHIN CONTROLS	• **Constrain recommendations:** limit AI to approved catalogs and sources. • **Steer the model:** retrain/condition on enriched, policy-aligned metadata (not popularity). • **Verify outputs:** monitor AI-generated manifests for vulnerable/EOL/shadow patterns and enforce dependency-aware guardrails in workflow.	**AI policy violation rate** = % of AI-generated dependency changes that introduce vulnerable, EOL, or unapproved components.

sonatype

From Guesswork to Grounded:

AI AGENTS WITH REAL WORLD INTELLIGENCE

As organizations increasingly delegate critical security decisions to AI systems, we face a fundamental challenge: even state-of-the-art language models lack access to real-time vulnerability databases, supply chain intelligence, and breaking change data. As a result, AI agents are confidently recommending nonexistent versions, introducing known vulnerabilities, and even suggesting malware-infected packages. The model is doing all of this while appearing authoritative.

THIS ISN'T AN INDICTMENT OF AI CAPABILITIES. IT'S A RECOGNITION THAT AUTOMATION WITHOUT LIVE INTELLIGENCE IS DANGEROUS AT SCALE.

Traditional upgrade strategies expose similar blind spots. Most Recently Published Version (Latest) heuristics, which software developers simply upgrade open source components whenever a newer version is available, assume newer means better, ignoring CVE disclosures, stability signals, and the cascade of breaking changes that transform simple updates into multi-week migrations. Meanwhile, ungrounded AI recommendations, regardless of the sophistication of the underlying model, operate on theoretical patterns rather than live security intelligence. Both approaches share a critical flaw: they make decisions without the data that actually matters and without the guardrails to guarantee code is compliant.

PROMPT ▼

```
You are helping a production engineering team decide on a dependency
upgrade path.

Based on your best knowledge, recommend the version they should target.
If newer releases may exist beyond your knowledge, still provide a specific
version and explain any uncertainty.

Dependency context:
- Package: {namespace}/{name}
- Current production version: {version}
- Ecosystem: {format}

Return JSON matching the schema.
```

In this research, Sonatype demonstrates a different path: **AI that is grounded in live intelligence, validated against real registries, and guided by breaking-change analytics governed by policy**. When AI operates with this foundation, its capabilities shift from theoretical suggestion engines to trusted, production-grade decision systems.

This chapter analyzes nearly 37,000 real dependency upgrades across Maven, npm, PyPI, and NuGet to quantify how ungrounded AI coding agents behave in practice and how security-intelligent governance closes the gap.

LLMs Hallucinate Versions at Scale

27.76% of dependency upgrades were hallucinations

Across 36,870 upgrade recommendations, 27.76% referenced non-existent versions including over 10,000 hallucinated package releases that would never resolve in a live repository.

FIGURE 4.1

Hallucination Rates by Confidence Level

Confidence	Hallucinated	Valid	Total	Hallucination Rate	Share of Hallucinations
High	23	1,336	1,359	1.69%	0.22%
Medium	4,504	18,959	23,463	19.20%	44.01%
Low	5,708	6,340	12,048	47.38%	55.77%

PERFORMANCE ANALYSIS OF THE LLM STRATEGY

The performance analysis of the LLM strategy (detailed in the "Grounding AI Agents In Real-World Intelligence" section of the *Appendix*) reveals an interesting finding regarding confidence:

- **GPT-5 was 98% accurate when it expressed high confidence**
- **It expressed high confidence in just 3.68% of recommendations**
- **Nearly half of all "low confidence" answers were incorrect**

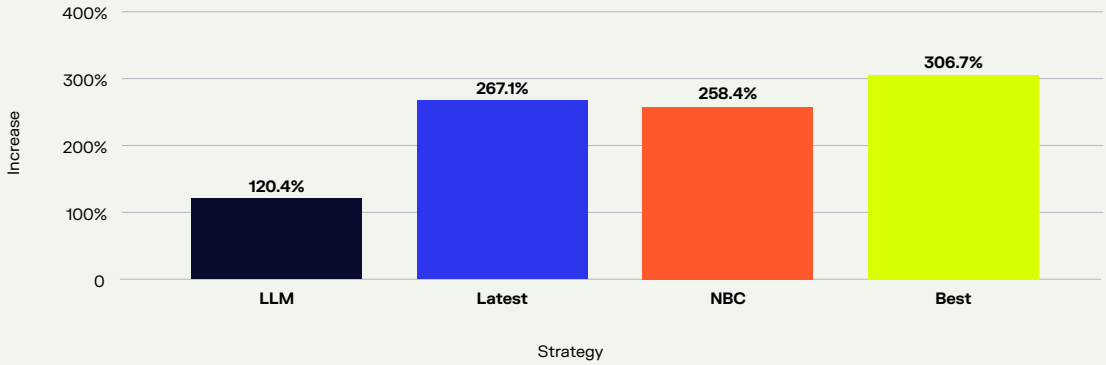
This confidence pattern was observed in a sample of real-world enterprise applications. While production AI systems might decline to answer when uncertain, the core issue remains: package ecosystems evolve constantly. New versions ship hourly. Security vulnerabilities are constantly emerging. **No training dataset, however comprehensive, can predict tomorrow's CVE or next week's breaking change.**

Sonatype's approach doesn't compete with agentic AI — it completes it. By grounding recommendations in live package registries, proprietary vulnerability and malware data, and breaking change calculations, we achieved zero AI hallucinations across the same 36,870 components. Every recommendation is verified against real repositories. Every upgrade is assessed for actual security impact.

The future isn't choosing between AI and traditional tools. It's AI agents operating with real-time intelligence that teams can trust in production.

FIGURE 4.2

Mean security score improvement per application by strategy



Security Improvement by Upgrade Strategy

Software ages like milk, not wine. As new vulnerabilities are discovered and disclosed, older package versions accumulate security debt while newer releases incorporate patches. Every day without upgrading increases exposure. Yet, not all upgrade paths are created equal. We compared four upgrade strategies across 856 enterprise applications. All strategies improved security, but not equally.

Figure 4.2 outlines the mean security score improvement for each application by strategy. Percent improvement is calculated as (total target security - total baseline security) / total baseline security × 100, averaged across vulnerable components from 856 enterprise applications. Security scores aggregate the severity and count of known vulnerability types on a 0–100 scale. For example, an application with 450 baseline points improving to 614 target points represents +36.4% security gain.

COMPARING FOUR UPGRADE STRATEGIES

We compared four upgrade strategies across 856 enterprise applications. All strategies improved security, but not equally

- **LLM-generated versions (LLM)** Lowest improvement of the strategies analyzed; 345 components became less secure
- **Most Recently Published Version (Latest)** Results in strong security outcomes but with extreme engineering costs
- **Sonatype 'No Breaking Changes' (NBC)** Chooses highest safe version without breakage; high security gains with minimal refactoring
- **Sonatype Best (Best)** Chooses highest version score regardless of breaking changes; highest security improvement overall

Overall, it is generally a good idea to remediate vulnerabilities. All upgrade strategies improve security outcomes, but not equally. LLM-generated (LLM) upgrade recommendations show the smallest uplift, recommending generally newer versions without proper guidance. Sonatype 'No Breaking Changes' (NBC) sees a significant improvement while identifying versions that minimize or eliminate breaking changes.

Then we have the Latest version strategy, with a significant improvement in security, but with a high engineering cost, as we will see later. The overall best improvement comes from the Sonatype Best (Best) strategy, which more holistically considers the security of the components (severity in combination) when identifying the best upgrade path.

LLM recommendations present a troubling paradox. While showing an improvement overall, **the model degraded security posture for 345 components**, ecommending newer versions that introduced more vulnerabilities than they resolved. This occurred when the model unknowingly chose versions that:

- Were compromised after its training cutoff
- Carried additional CVEs
- Were newer, but were also riskier

MALWARE AND PROTESTWARE RECOMMENDATIONS

The LLM strategy did more than hallucinate versions. It recommended **`sweetalert2 11.21.2`**, which is confirmed protestware executing political payloads, as well as **`color 5.0.1`** and **`color-string 2.1.1`**, which were compromised in a major supply chain attack. These packages were not obscure edge cases. They were widely downloaded and part of a high-profile security event that occurred *after the model's training data cutoff.*

PROMPT ▼

```
{
  "color": {
    "recommended_version": "5.0.3",
    "confidence": "high",
    "rationale": "Latest per Sonatype MCP; MIT licensed, no known vulnerabilities."
  },
  "sweetalert2": {
    "recommended_version": "11.26.3",
    "confidence": "high",
    "rationale": "Latest stable; fixes prior malware flag and known CVE."
  }
}
```

THIS IS THE CORE PROBLEM: AI CANNOT DETECT THREATS THAT HAPPENED AFTER IT WAS TRAINED. AI NEEDS REAL-TIME INTELLIGENCE.

While security improvements justify upgrades, the practical question remains: what does it cost? Breaking changes drive developer effort, transforming version bumps into multi-day refactoring projects. The following analysis quantifies these costs across strategies, revealing trade-offs between security gains and implementation burden.

BREAKING CHANGE COST ANALYSIS

Security improvements come at a price measured in developer hours and refactoring effort. Across 856 enterprise applications with representative dependency footprints, upgrade strategies impose dramatically different implementation costs.

Figure 4.3 below compares median per-application upgrade budgets across the four strategies. NBC delivers the lowest-friction path: roughly ~1 engineer-week to modernize an entire app while avoiding destabilizing work. Best still holds the costs under $20K and under 200 hours per app, yet it absorbs the additional change needed to drive higher security scores.

FIGURE 4.3

Upgrade Cost & Effort per Application

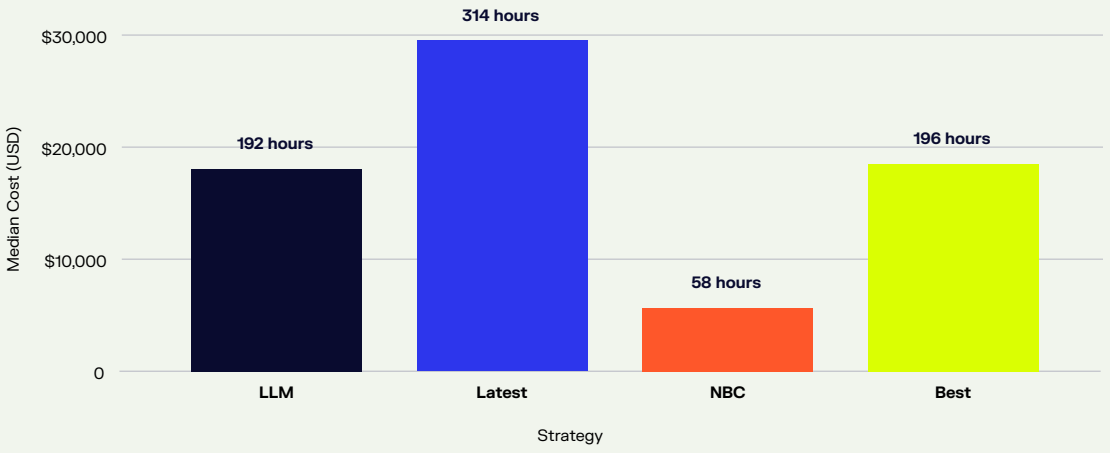
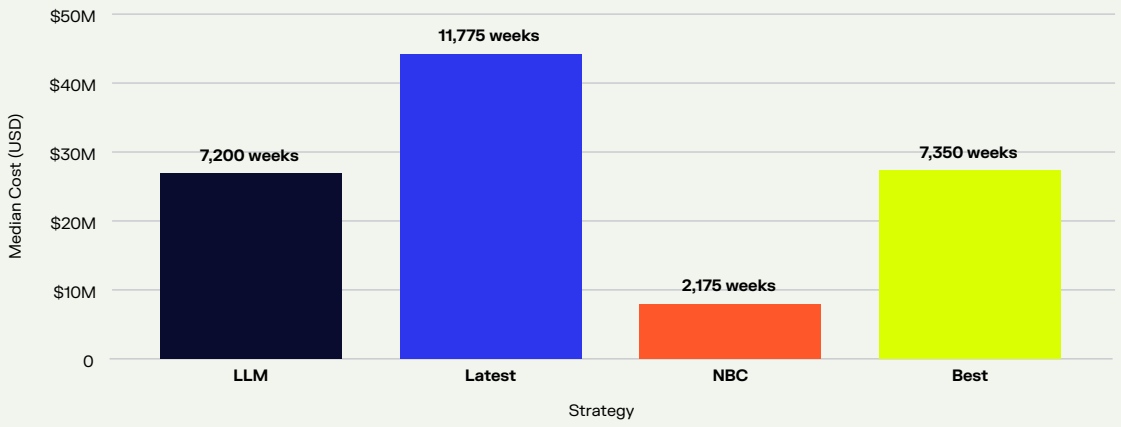


FIGURE 4.4

At Enterprise Scale: Upgrade Cost & Effort



Both outclass the unmanaged options: unconstrained Latest upgrades result in nearly 5x the median spend versus NBC, and LLM-only selections land in the same cost bracket as Best without the significant risk reduction.

Applying a generic ~8% copilot uplift to the same per-app upgrade totals, NBC still modernizes an app for a little over $5K and ~53 hours, while chasing Latest upgrades soaks up nearly $27K and 288 hours — over five times the spend and the engineering time.

That gap isn't just a bookkeeping line; it's opportunity cost. Every extra week poured into unmanaged upgrades is a week not spent on security hardening, paying down tech debt, or feature delivery. LLM-only picks land in the same budget band as Best yet lack its curated risk reduction, reinforcing that disciplined Sonatype strategies are the only way to keep upgrade budgets predictable without cannibalizing roadmap work.

HOW COSTS SCALE: ORGANIZATIONAL IMPACT

This projection scales each strategy's median per-application effort across a representative large enterprise portfolio. It illustrates the cumulative impact of decentralized upgrade decisions over time.

In practice, organizations don't upgrade every dependency in every application all at once. Instead, they perform ongoing dependency maintenance — small, continuous updates that, across hundreds or thousands of applications, represent a near-constant workload. Without a cohesive strategy, these distributed efforts can quietly accumulate into multimillion-dollar annual costs.

NBC automation keeps portfolio-level upgrade effort roughly an order of magnitude lower than unmanaged Latest adoption, while achieving a similar security posture. Teams targeting the most secure baseline can adopt Best selectively, reserving deeper migrations for critical systems where maximum vulnerability reduction warrants the additional investment.

Our analysis of 36,870 dependency upgrade recommendations exposes a critical divergence between the promise of autonomous AI agents and the reality of software supply chain security. The data suggests that without access to real-time package registry intelligence, both state-of-the-art LLMs and traditional Latest heuristics fail to balance security risk with engineering effort.

SWEETALERT2 VERSION 11.21.2

Data corruption & protestware

This package creates a 'noWarMessageForRussians' banner on any Russian website using this component that is running in a browser using Russian.

COLOR VERSION 5.0.1 & COLOR-STRING VERSION 2.1.1

Cryptostealer & hijack

Taken over as part of the chalk/debug campaign, color and color-string were manipulated to extract victims' cryptocurrency from browser wallets.

THE "INTELLIGENCE" GAP IN GENERATIVE AI

The most alarming finding is not merely that ungrounded AI makes mistakes, but that it makes dangerous ones with high confidence. **The observed 27.8% AI hallucination rate in GPT-5 recommendations confirms that language models, when isolated from live repositories, struggle to distinguish between existing and non-existent software.**

More critically, the "hallucinations" were not only harmless version number errors, but also data corruption, protestware, and hijacked packages. This illustrates a fundamental limitation: training data cuts off, but supply chain attacks operate in real-time. A model trained before a package compromise cannot "know" a version is unsafe without a live feed of vulnerability intelligence.

Furthermore, the LLM strategy delivered the lowest security improvement (+120.4%) of all methods tested. In 345 specific instances, following the AI's advice actually degraded the component's security posture by introducing more vulnerabilities than it resolved.

THE FALSE ECONOMY OF "LATEST VERSION"

While the industry often defaults to "always upgrade to latest" as a best practice, our cost analysis reveals this to be a financially inefficient strategy. While Latest achieved strong security gains (+267.1%), it did so at a brute-force cost: approximately $29,516 and 314 developer hours per application. When scaled to a portfolio of 1,500 applications, the Latest strategy demands nearly $44.3 million in estimated labor costs.

HOW WE GET TO $44M AT THE ENTERPRISE SCALE

- 180 dependencies per app × ~$161 per app = **~$29.5K per app**
- 1,500 apps × $29K = **~$44.3M**

This 5x cost multiplier, compared to intelligent automation, represents a massive opportunity cost; every hour spent resolving breaking changes from an unnecessary major version jump is an hour lost to feature development or debt reduction.

Grounding is the Missing Link

The high accuracy (98%) of GPT-5 in the rare instances (3.68%) where it expressed "High Confidence" suggests that the reasoning capabilities of modern models are sound, but their context is insufficient.

The path forward is not to choose between AI and traditional tools, but to ground autonomous AI agents in verified intelligence. By feeding the model real-time data — including computed breaking changes and enhanced vulnerability and malware intelligence, Sonatype's approach eliminates AI hallucinations entirely while empowering teams to choose the upgrade path (Best vs. No BC) that aligns with their risk tolerance and budget.

SONATYPE SECURITY HYBRID

You can also take a hybrid approach that puts security first in the version scoring algorithm. When a version has a perfect security score, it recommends NBC; otherwise, it defaults to the Best recommendation. This results in:

327%

security gain from remediating vulnerable components

2.1X

lower dependency upgrade cost and effort compared to Latest Version

sonatype

The 2025 Global Software Assurance Mandate:

TRANSPARENCY AS THE NEW TRUST

Transparency has become the currency of software supply chain security.

Around the globe, policymakers and regulatory bodies have moved from rhetoric to regulation on that principle. SBOMs (Software Bills of Materials), attestations, and provenance tracking are no longer optional. They're being elevated as expressions of transparency, codified in law, and embedded into how organizations will be required to demonstrate security readiness. We estimate 90% of global organizations fall under one or more regulatory requirements to demonstrate evidence of software assurance.

In this chapter, we map the current regulatory landscape, identify key changes and enforcement deadlines as of the end of 2025, and forecast how organizations should prepare. We show how software compliance is shifting from policy to code and why teams that treat transparency as an engineering challenge will win.

UP TO 90%

of organizations around the world will fall under one or more regulatory requirements

From Open Source Governance to Regulatory Mandate

For years, SBOMs, consumption governance, and software supply chain transparency were treated as best-practice responses to technical risk. What changed in 2025 is that transparency moved from optional to required. Across regions, regulations are converging on the same basics: minimum SBOM elements, interoperable formats, and proof of secure development practices. The focus is no longer just what's in the software, but who delivered it and how it was built and shipped.

This shift is also changing how compliance works. Manual checklists are giving way to automation and "compliance as code," because procurement, audits, and enforcement increasingly demand auditable evidence. Vendors are expected to show, not just claim, that they generate SBOMs, track provenance, sign artifacts, and provide attestations when asked.

As a result, open source governance is now squarely in the regulatory spotlight. Policies that once lived as internal guidelines are becoming obligations, driven by frameworks such as the EU Cyber Resilience Act and NIS2, alongside U.S. Executive Order 14028. OSS components, forks, transitive dependencies, and license ambiguity can create exposure not only through security risk, but through procurement breach, audit failure, or product liability. The UK is moving in the same direction. The Cyber Security and Resilience Bill, recently introduced to Parliament, signals expanded scope and faster incident reporting, reinforcing that assurance has to be operational, repeatable, and provable.

Sonatype sees this evolution as a positive forcing function. The industry already knows what "good" looks like: mapped dependencies, SBOMs, signed provenance, and attestable secure practices. The work now is making those outputs default by embedding them directly into development and release workflows.

GLOBAL REGULATORY TIMELINE

May 12, 2021
US Executive Order 14028 signed.

July 12, 2021
US NTIA publishes SBOM "Minimum Elements."

January 16, 2023
NIS2 and DORA enter into force in the EU.

December 1, 2023
Australia's ASD ISM first edition released.

October 18, 2024
NIS2 compliance measures apply in the EU.

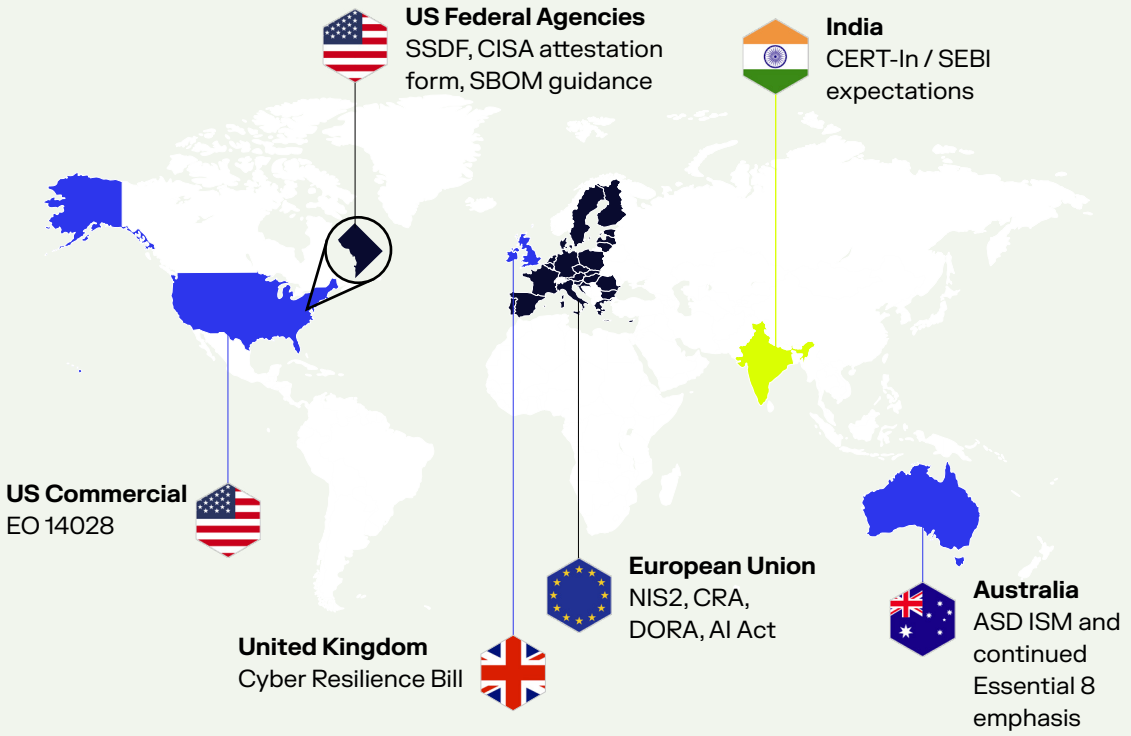
December 10, 2024
EU Cyber Resilience Act (CRA) entered into force.

January 17, 2025
DORA compliance measures apply in the EU.

July 25, 2025
CERT-in mandatory annual third-party cybersecurity audits in India.

November 12, 2025
UK CSRB introduced to parliament.

THE REGULATORY MAP: WHERE THINGS STAND AT THE END OF 2025



MANDATES IN FORCE / DETAILED GUIDANCE

- European Union
- United States (Federal)

MANDATES ADOPTED / IN TRANSITION

- United Kingtom
- United States (Commercial)
- Australia

EMERGING REQUIREMENTS / GUIDANCE ONLY

- India

Note: On January 23, 2026, OMB issued Memorandum M-26-05 rescinding the standardized secure software self-attestation approach and directing agencies to use a risk-based model for software assurance, including requesting SBOMs when appropriate.

United States Software Regulations

In the U.S., the federal approach has matured into a multi-layered regime. The foundation lies in the NIST SP 800-218 (Secure Software Development Framework, SSDF) and the self-attestation and verification regime instituted under Executive Order 14028 and related OMB memoranda (such as M-22-18 and M-23-16). The Cybersecurity and Infrastructure Security Agency (CISA) has published the Secure Software Development Attestation Form, which vendors must submit to certify adherence to secure development practices.

Federal procurement rules now condition software eligibility on those attestations, and as noted by the Government's Software Acquisition Guide, procurement agencies are being guided to require transparency via SBOMs, signed artifacts, and auditable supplier processes.

SOFTWARE VENDORS SEEKING FEDERAL CONTRACTS MUST PROVIDE:

- ☑ **Attestation** that their development practices align with SSDF.
- ☑ **Minimum Elements** for SBOMs (in one of the formats specified by the National Telecommunications and Information Administration (NTIA) and supplemented by CISA.
- ☑ **Evidence** of component provenance, vulnerability handling, and in some cases third-party assessment.

For federal agencies, this has broader implications: the procurement lifecycle now explicitly links security software assurance to procurement eligibility, renewal cadence, and supplier audits. Vendors that cannot demonstrate attestations or generate SBOMs may simply be disqualified.

In our view, organizations outside of federal scope but working in critical verticals should view this as indicative of what is coming in the private sector: procurement leverage, audit readiness, and transparency of supply chain footprint will become table stakes.

European Union Software Regulations

Europe has launched multiple landmark pieces of legislation in the software supply chain and product cybersecurity domain.

NIS2 + IMPLEMENTING REGULATION

The NIS2 Directive (Directive (EU) 2022/2555) entered into force in January 2023 and introduced higher standards for cybersecurity risk management, incident reporting, and software supply chain security for "essential" and "important" entities. The Commission Implementing Regulation (EU 2024/2690) defines more specific technical and methodological requirements. In June 2025, the European Union Agency for Cybersecurity (ENISA) published technical implementation guidance for the software regulation, mapping each requirement to evidence, frameworks, and standards.

NIS2 explicitly requires organizations to manage software supply chain security risks, incorporate secure-by-design and secure procurement principles into system acquisition and development, and demonstrate effective software governance. Organizations in scope must maintain documented risk management policies, implement incident detection and handling processes, and assess and manage the cybersecurity posture of suppliers.

CYBER RESILIENCE ACT (CRA)

The CRA (Regulation (EU) 2024/2847) establishes a horizontal regulatory framework for "products with digital elements" (PDEs). It entered into force on December 10, 2024. Its main obligations begin to apply in December 2027, after a three-year transition period. Certain vulnerability handling and reporting obligations start earlier, in September 2026.

Under the CRA, manufacturers must ensure that products are designed, developed, and produced in accordance with essential cybersecurity requirements. This includes implementing Secure by Design practices, performing and documenting risk assessments, and ensuring ongoing vulnerability handling throughout the support period they specify. When integrating third-party components, including free and open source software, manufacturers remain responsible for assessing risks and maintaining appropriate technical documentation.

The CRA requires manufacturers to maintain detailed technical documentation about security properties and supply chain dependencies.

While the software regulation anticipates greater transparency of software components, it does not explicitly prescribe an SBOM format; however, it does empower the Commission to adopt delegated acts specifying additional elements or procedures, which could include SBOM-related requirements in the future.

A notable feature of the CRA is that it brings certain actors in the open source ecosystem into scope. Specifically, "open source software stewards" are identified as individuals who play a coordinating role in the development and distribution of widely used OSS.

They may be subject to obligations such as adopting documented cybersecurity processes, providing attestations, and cooperating with market surveillance authorities. These obligations apply only where such stewards meet the criteria defined by the regulation and are not intended to cover individual volunteer contributors.

In parallel, the revised EU Product Liability framework (Regulation (EU) 2024/2853) extends no-fault liability to software and digital products. Non-compliance with the CRA's cybersecurity obligations may therefore expose manufacturers to strict product liability for damage caused by vulnerabilities or security defects in products with digital elements, irrespective of fault or negligence.

From our perspective at Sonatype, CRA and NIS2 together represent a sea-change: software and products containing digital elements are regulated from design through maintenance; transparency and SBOMs are wired in. The message: software compliance requires end-to-end visibility, not after-the-fact patching.

Other Key Jurisdictions

Beyond the U.S. and EU, several jurisdictions are aligning with this global transparency movement.

In India, the Indian Computer Emergency Response Team (CERT-In) and the Securities and Exchange Board of India (SEBI) have updated incident reporting obligations and disclosure requirements.

While SBOM mandates are less mature than in the U.S. or EU, regulated entities are being required to establish software supply chain documentation and risk management programs as an expectation.

In Australia, the Australian Signals Directorate (ASD) Information Security Manual (ISM) and the "Essential 8" framework have long influenced cyber-maturity expectations.

In 2025, those frameworks began emphasizing software supply chain transparency, SBOMs, and supplier assurance as differentiators for procurement in critical sectors.

From Sonatype's vantage point: while regulatory maturity varies, the direction is consistent globally. Firms that invest in transparency and software assurance now will gain a competitive advantage.

Regulated Industries: From Obligation to Opportunity

Historically, heavily regulated industries, including financial services, healthcare, and critical infrastructure, have been the earliest adopters of software assurance and SBOM mandates. The regulatory developments emerging in 2025 are broadening that landscape.

Under DORA, financial institutions and their ICT third-party providers must implement comprehensive ICT-risk-management frameworks, incident reporting processes, and supplier-governance controls. Requirements for documenting cyber resilience strategies and demonstrating oversight of software supply chain risk are now appearing in procurement cycles and audit practices.

In 2025, multiple regulatory bodies began explicitly treating artificial intelligence components, including models, training data, evaluation pipelines, and automated decision systems, as software artifacts subject to supply chain controls.

The AI-compliance landscape is rapidly maturing, led by the EU AI Act's staggered phase-in schedule and U.S. federal guidance following Executive Order 14110 (later replaced by Executive Order 14179).

Formats and Interoperability

The two dominant SBOM schemas are SPDX and CycloneDX. SPDX has traditionally excelled in open source license compliance and metadata governance; CycloneDX is particularly effective for vulnerability/component dependency correlation and CI/CD integration. In practical terms, organizations must evaluate when to use which schema: for licensing-governance pipelines SPDX may be the default; for live software supply chain tools, vulnerability context and runtime telemetry, CycloneDX may be preferable.

Interoperability is increasingly mandated. For example, the CRA allows the European Commission to specify by delegated acts the format and elements of SBOMs for products with digital elements. "Attestation" too has become the currency of procurement and audit readiness. In the U.S., CISA's attestation form formalized vendor self-attestation to SSDF practices. Similarly, the EU regulatory regimes expect documented evidence of risk assessments, vulnerability-handling procedures, and software assurances.

The key operational lesson here is that transparency must be engineered: organizations must treat SBOM generation, artifact signing, attestation capture, and publishing as part of the build-and-release pipeline — not as an afterthought. The enforcement regime is moving from "show us your policy" to "show us the artifact".

SBOM & ATTESTATION FORMATS

● strong ◐ mixed

Category	SPDX	CycloneDX	Notes
Licensing & IP metadata	● strong	◐ mixed	SPDX is fundamentally license-first (SPDX expressions, compliance lineage). CycloneDX carries license data well, but SPDX remains the legal/compliance "gold standard."
Vulnerability / Dependency correlation	◐ mixed	● strong	CycloneDX was designed with security and dependency graphs in mind. SPDX supports this, but it's not the primary design center.
CI/CD Friendliness	◐ mixed	● strong	CycloneDX is more commonly generated by modern build tools, scanners, and CI jobs. SPDX is used in CI/CD, but more often post-build or for compliance artifacts.
Ecosystem Tooling & Adoption	◐ mixed	● strong	CycloneDX has stronger momentum in AppSec, SCA, and cloud-native tooling. SPDX remains dominant in regulated, supplier-driven, and government contexts — strong, but slower-moving.

Open Source License Compliance in the New Regime

Key risk patterns include transitive copyleft propagation (when combining or distributing code with copyleft-licensed dependencies can trigger downstream obligations), unclear or missing license metadata, and forked components that diverge from upstream development, making provenance and patch-tracking difficult. The compliance challenge is to define meaningful metrics. For example, the percentage of components with approved licenses, the number of license conflicts detected pre-merge, and the mean remediation time for license-non-compliant usage. While current compliance regimes rarely specify exact thresholds for these metrics, the trend is clear: organizations that cannot demonstrate open source license compliance of intake and remediation are increasingly disadvantaged in procurement, audit, and regulatory contexts.

Under the CRA, for instance, open source software stewards must put in place and document in a verifiable manner a cybersecurity policy and cooperate with market surveillance authorities and CSIRTs/ENISA in certain circumstances. They may also need to provide security documentation and, in some cases, attestations of compliance. At Sonatype, we increasingly advise clients that an open source intake policy is no longer just software governance best practice — it is rapidly becoming a compliance expectation.

The practical implication is that organizations must operationalize OSS intake, contribution, and remediation workflows; integrate open source license compliance scanning and component metadata tracking into CI/CD; ensure SBOMs capture accurate license data; and maintain audit logs of intake decisions. Downstream, procurement teams are beginning to require supplier attestations that OSS intake and license governance policies are in place and followed.

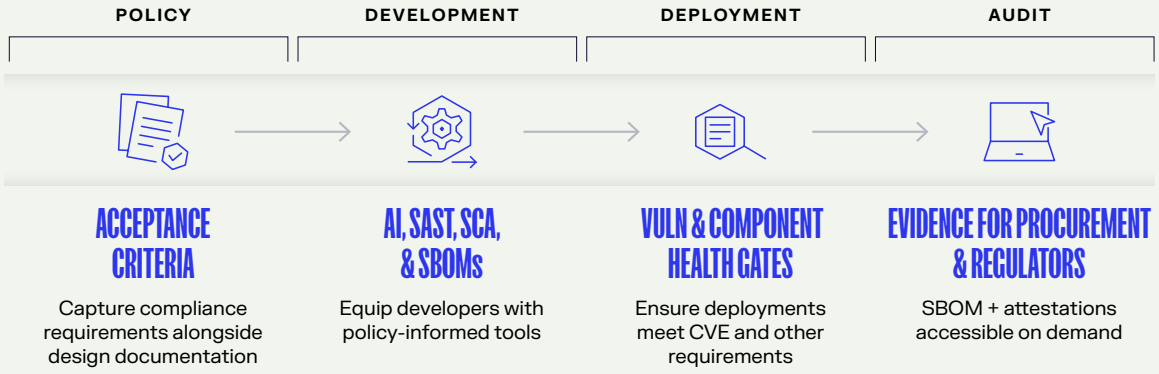
Bringing Policy into Reality

Whether your discussion is around Compliance-as-Code, Policy-as-Code, GRC Engineering, or some other umbrella term — the industry is shifting toward automated governance to keep pace with the exponential acceleration in both software development speed and malware presence.

As we can see in the CNCF's Automated Governance Maturity Model and OpenSSF's Gemara, software development lifecycles can be significantly accelerated while improving compliance outcomes by ensuring codification and automation at every opportunity:

- Writing organizational policies in a machine-optimized format reduces friction for both human and AI interactions as tools interface with structured data. Policies turn compliance into a foundational design requirement instead of being stapled on after development.
- Selection of development tools can be done with early feedback according to policy, informed by supplier onboarding workflows: standardized questionnaires, third-party assessment checklists, attested secure development practices.

COMPLIANCE-AS-CODE



- Development tools can provide early feedback both in the IDE and at pull/merge time with evaluation and enforcement of priorities such as license requirements and trusted dependency registries; blocking patterns if disallowed licenses or untrusted sources are requested.
- Deployment builds can automatically evaluate code and enforce requirements such as generation of SBOMs (SPDX or CycloneDX) with all dependencies, versions, metadata, and signatures. At release time, attachment of signed provenance data and attestation, artifacts to the release package.
- Audits become streamlined by using compiling machine-readable policies, evaluation logs, enforcement results, and relevant artifacts (e.g., via a customer-accessible portal, or public registry) to support audit, procurement, or regulator review.

The question is not whether your organization can produce an SBOM or attestation but whether it has the automation, traceability, and audit-readiness baked into the build workflow. Compliance is not an add-on; it must be part of the entire software development lifecycle.

METRICS AND MATURITY FOR SELF-ASSESSMENT

Transparency and software compliance readiness must be measured. Sonatype recommends organizations track four dimensions: Coverage, Integrity, Responsiveness, and Assurance.

- **Coverage:** What proportion of shipped artifacts include SBOMs? What percentage of components have declared licenses and an explicit policy decision?
- **Integrity:** How deep is your SBOM (i.e., dependency depth)? Are provenance records signed and traceable? Is your rebuild reproducible? What signal-to-noise ratio do your scans produce (i.e., real vulnerabilities vs false positives)?
- **Responsiveness:** What is your mean time to provide a customer or regulator an SBOM or attestation? What is your mean time to resolve a non-compliant license usage? What is the median time to apply a post-release security update?
- **Assurance:** What percentage of releases meet SSDF-defined attestation? What percentage of your suppliers provide verifiable artifacts (SBOMs, signed provenance, secure development attestation)?

SOFTWARE MATURITY ASSESSMENT

Control	Metric	Readiness
Transparency	% of artifacts with SBOM provenance	90% = Mature
Licensing	% dependencies with approved license	90%+ = mature
Security Response	Mean time to patch vulnerabilities	<15 days = mature
Attestation	Releases meeting SSDF standard	80%+ = mature

Software Assurance as Currency

2025 marked the inflection point. In 2026, software assurance becomes the standard by which software earns trust. Transparency through SBOMs, attestations, and provenance is moving from policy to regulation, from a nice-to-have to a procurement requirement, and from an audit checkbox to a competitive differentiator.

Sonatype views these mandates as catalysts for safer software. When compliance is built into the delivery process, software becomes more measurable, auditable, and secure.

Organizations that embed transparency into build pipelines, integrate supplier attestations into procurement, and treat SBOMs and provenance as first-class artifacts will be ahead of the curve. Everyone else risks procurement lockout, audit disruption, and downstream liability.

IN 2026, COMPLIANCE IS NOT JUST ABOUT AVOIDING PENALTIES. IT IS ABOUT EARNING TRUST, AND TRUST IS THE CORE ASSET OF THE SOFTWARE SUPPLY CHAIN.

ONE-YEAR SOFTWARE COMPLIANCE PLAYBOOK

Based on our experience across clients and regulatory developments, here is a recommended playbook for organizations aiming to be fully compliant (and competitive) in a world of compliance through governance.

Timeframe	Primary Goal	Key Actions	Outputs
0–3 MONTHS	ESTABLISH OWNERSHIP AND BASELINE	• Name cross-functional compliance owners • Inventory SBOM coverage • Choose SPDX/CycloneDX • Validate toolchain • Run attestation gap analysis (signing, pipeline evidence, vuln mgmt)	• Ownership model • SBOM baseline • Schema decision • Gaps list
3–6 MONTHS	OPERATIONALIZE COMPLIANCE IN CI/CD	• Require SBOM per release • Sign and (if needed) publish • Map obligations (CRA/NIS2/AI Act as applicable) • Define required artifacts • Implement compliance-as-code (license gate, SBOM at build, provenance at release) • Start metrics	• SBOM + signed provenance in pipeline • Obligations/ artifact catalog • Initial metrics
6–12 MONTHS	EXTEND TO SUPPLIERS AND AUDIT READINESS	• Standardize supplier onboarding • Require supplier attestations, SBOMs, signed provenance • Quarterly reporting dashboards • License conflict + copyleft workflows • Make artifacts accessible and auditable	• Supplier requirements in place • Dashboards • Audit-ready evidence repository

By the end of the year, your goal should be: all major releases produce SBOMs, all software vendors/ suppliers have attested secure-development practices, all major components have approved licenses, and compliance metrics are live in software governance dashboards.

Methodology

REGISTRIES, MODELS, AND THE NEW SOFTWARE INFRASTRUCTURE BURDEN: WHEN GROWTH MEETS GRAVITY

This chapter is based on Sonatype's analysis of registry consumption and infrastructure load signals drawn from aggregated telemetry across major open source ecosystems (with Maven Central used as a primary lens where noted). The study examined download and re-download behavior over the report's specified reporting windows, focusing on how automated software delivery systems (CI/CD pipelines, ephemeral build fleets, and dependency managers) amplify demand on shared registry infrastructure.

Sonatype Security Research Team evaluated registry load and sustainability pressure using four primary measures:

- Growth and concentration: overall request volume trends and the degree to which traffic is dominated by a small set of high-volume consumers.
- Re-download intensity: repeat-fetch behavior for the same artifacts, used as a proxy for cache inefficiency and rebuild amplification.
- Burstiness and hotspots: peak download behavior (e.g., 95th percentile patterns) to distinguish steady consumption from spiky traffic that strains shared infrastructure.
- Source footprint signals: directional indicators such as distinct IP counts and distribution patterns to infer automation characteristics (shared egress/NAT, centralized runners), without treating IPs as definitive identity.

While the chapter focuses on open source registry dynamics, the patterns identified (automation-driven amplification, concentrated demand, and cache fragility) reflect broader structural pressures affecting modern software supply chains. All quantitative results reflect a point-in-time snapshot as of the report's stated verification date, and are reported in aggregate to avoid attribution to specific organizations or users.

THE EVOLVING SOFTWARE SUPPLY CHAIN ATTACK SURFACE: MALWARE AT THE GATE

This chapter is based on Sonatype's analysis of malicious open source packages identified through a mix of automated detection and expert review, using publicly observable package metadata and Sonatype threat intelligence. We evaluated packages observed within the report's stated window using a consistent, multi-label threat taxonomy (one package may map to multiple behaviors), normalized duplicates/variants to avoid inflating counts, and used clustering signals (payload and code reuse, naming patterns, publisher behavior, dependency relationships, and shared infrastructure) to identify coordinated campaigns. Findings are reported in aggregate as a point-in-time snapshot as of the report's verification date.

THE THREE LAYERS OF FAILURE IN MODERN VULNERABILITY MANAGEMENT

The Data Layer: This analysis evaluates the quality and usefulness of vulnerability records for open source by comparing public advisory data with Sonatype's enriched vulnerability intelligence. We assembled a study set of 1,718 open source–relevant CVE records disclosed within the report's defined window (January 1, 2025 to December 31, 2025), drawing from publicly available sources (including NVD/CVE metadata and CVSS where present) and Sonatype Security Research. For each CVE, the Sonatype Security Research Team assessed five core dimensions that directly affect whether teams can make consistent remediation decisions: (1) coverage (whether NVD provides usable CVSS/severity and how often that aligns with Sonatype), (2) scoring consistency (magnitude and direction of CVSS score drift between NVD and Sonatype, plus resulting severity-category shifts), (3) false positives (records or affected-version claims that would trigger remediation for non-impacted software), (4) false negatives (missing, incomplete, or delayed records/metadata that would cause impacted software to be missed), and (5) timeliness (time between public CVE disclosure and availability of NVD analysis/scoring). Results are reported at the CVE level using consistent matching rules across sources, with percentages rounded for readability; all findings reflect a point-in-time snapshot verified as of the report's stated "as of" date.

The Consumption Layer: This section is based on Sonatype's analysis of Maven Central download telemetry to measure real-world consumption of known vulnerable vs. fixed component versions. We constructed a dataset of components with publicly disclosed vulnerabilities and an available remediated (fixed) release, then measured how frequently vulnerable versions continued to be downloaded relative to their fixed counterparts over the report's stated time windows. Downloads are treated as a consumption signal (what build systems actually pull), not as a proxy for unique users, and results are reported in aggregate to quantify avoidable risk—cases where vulnerable versions remain in active use even though safer versions exist.

The Ecosystem Layer:
Prevalence of EOL components
We analyzed a representative sample of more than 3,000 enterprise SBOMs. For each SBOM, we examined the fully resolved dependency graph, including all transitive dependencies, and identified the number of package versions that were end-of-life. We calculated the percentage of EOL components per SBOM and then aggregated these results across all enterprises to measure overall EOL prevalence.

Number of EOL components with unpatched CVEs
We analyzed a database of over 11 million package versions with known end-of-life status and known, unpatched CVEs. This analysis identified approximately 81,000 EOL package versions with unpatched vulnerabilities. To estimate ecosystem-wide impact, we weighted this dataset against the broader population of open-source package versions, normalizing for selection bias introduced by database coverage and sourcing constraints. This produced an estimated total of more than 400,000 end-of-life package versions with unpatched CVEs across open-source ecosystems.

Breakdown of EOL Components by Registry
We analyzed a database of over 11 million package versions with known end-of-life status and grouped them by package registry. Within each ecosystem, we calculated the percentage of package versions that are end-of-life versus those that are currently supported. This resulted in a per-ecosystem end-of-life rate, as shown in the chart.

FROM GUESSWORK TO GROUNDED: AI AGENTS WITH REAL WORLD INTELLIGENCE

We analyzed a sample of enterprise applications scanned over a three-month window (June–August 2025), filtering to valid scans (those with >10 components) to remove setup/test/incomplete results. For apps with multiple stages, we selected the most operationally mature snapshot using the hierarchy compliance > operate > release > build > develop > proxy, and then took each app's first valid scan within the period. Analysis focused on four ecosystems (Maven, npm, PyPI, NuGet) and used direct dependencies identified by Sonatype's component recognition as upgrade candidates; apps that migrated into/out of an ecosystem during the window were kept to reflect real-world complexity.

We compared five upgrade strategies: No Breaking Changes (highest version score without breaking changes), Latest (most recent by publication date), Sonatype Best (highest version score regardless of breaking changes), Sonatype Security Hybrid (use No Breaking Changes only if it achieves a perfect security score of 100, otherwise fall back to Best), and an LLM strategy where GPT-5 (reasoning_effort=medium) returned a JSON recommendation (version, confidence, short rationale) per dependency (≈37,000 components, processed asynchronously with concurrency). Breaking-change effort was modeled using four buckets (0–5, 6–20, 21–100, 101+ changes) mapped to estimated hours and cost at $94/hr (conservative lower bound), with SemVer fallbacks when telemetry is unavailable (patch→L1, minor→L2, major→L3; L4 requires explicit data).

Security outcomes were measured via a 0–100 security score derived from Sonatype vulnerability intelligence, combining the worst-severity issue with the count of distinct vulnerability types (log-transformed to reflect diminishing marginal impact). Strategy comparisons used Welch's t-tests across primary outcomes (security score change and breaking-change count) at $\alpha=0.05$.

sonatype

2026

State of the SOFTWARE SUPPLY CHAIN®

Sonatype is the leader in AI-driven DevSecOps. As the maintainers of Maven Central and creators of Nexus Repository, Sonatype has spent two decades pioneering how the world manages and secures open source software — making Sonatype the trusted authority for modern software supply chains. With unmatched open source visibility and a unified product suite built for modern software development, Sonatype gives enterprises the intelligence and automated governance they need to harness the full potential of open source and AI. Sonatype handles the complexity behind the scenes: guiding component and model selection, blocking harmful malicious code, automating dependency and vulnerability management, and ensuring faster, more reliable builds — so developers spend more time on innovation and less time on remediation and rework. Trusted by more than 15 million developers, Sonatype helps power secure, modern software development at nearly 2,000 global organizations including 70% of the Fortune 100. To learn more about Sonatype, please visit www.sonatype.com.