# The New Vulnerability Landscape

Understanding, Researching, and Mitigating Software Vulnerabilities in the Age of AI and Forever-Day Vulnerabilities (2026 Edition)

# The New Vulnerability Landscape

Understanding, Researching, and Mitigating Software Vulnerabilities in the Age of AI and Forever-Day Vulnerabilities (2026 Edition)

Javier Perez

# CONTENTS

# 1 EXECUTIVE SUMMARY

This ebook examines the new realities of software vulnerabilities, what any organization running open source software should know, and reaches three conclusions.

First, artificial intelligence has permanently changed the economics of vulnerability discovery. Frontier AI models can now read a codebase and hypothesize breaking points. Discovery is accelerating to machine speed, and every published fix becomes a starting point for exploit development.

Second, end-of-life (EOL) open source software is where this acceleration does the most damage. When a framework, runtime, or library stops receiving security fixes, every vulnerability disclosed against it from that day forward remains permanently unpatched, a class of exposure this ebook calls “the forever-day vulnerabilities.” Compounding the problem, most vulnerability scanners cannot see this risk because they track known CVEs, not lifecycle status, and most open source packages never formally announce end-of-life.

Third, the exposure is manageable, but only deliberately. Organizations should inventory their software components with lifecycle status displayed alongside vulnerability status, then apply a four-option decision framework: migration as the long-term destination, commercial extended support or compensating controls as the bridge, and risk acceptance only for narrowly bounded exceptions with expiration dates.

From basic concepts to vulnerability lifecycle and recommendations, this ebook provides an up-to-date state of vulnerabilities in open source software.

# 2 INTRODUCTION

For most of the past two decades, vulnerability management followed a comfortable rhythm. A researcher or vendor found a flaw or vulnerability, a CVE was assigned, security scanners flagged it, a patch shipped, and organizations worked through the backlog to deploy patches. The process was never fast, but now there are more disclosed vulnerabilities than ever.

The composition of that volume matters as much as its size. The CVE Numbering Authority (CNA) ecosystem has shifted. With specialized CNAs focused on third-party ecosystems and open source, vulnerability disclosure is no longer dominated by a handful of large vendors with mature security programs; it is a distributed, high-volume pipeline driven heavily by AI-assisted discovery.

The median time between vulnerability disclosure and confirmed exploitation has fallen to just five days according to CSA. Attacks targeting website vulnerabilities reached 6.29 billion in 2025, a 56 percent year-over-year increase according to The State of Application Security 2026. Meanwhile, the compliance clocks most organizations operate under, such as the 30-day remediation window in PCI DSS Requirement 6.3.3, were written for an era when attackers took weeks or months to weaponize a disclosure. Attackers now routinely move faster than the fastest mandated remediation timeline. For open source software that is abandoned, unsupported, or reached the end-of-life (EOL), there is no patch window because there is no patch. For practical purposes, this document refers to all such unmaintained software as EOL software: software that receives no security fixes, no new releases, and no patches of any kind.

This ebook is about operating inside this new reality: understanding vulnerabilities well enough to shape strategies for any organization with open source software.

# 3 VULNERABILITIES, WEAKNESSES, AND EXPLOITS

Effective risk management demands a shared taxonomy across engineering, security, and governance teams. Different terms can map to different objects across databases and systems, and confusing them can lead to misunderstandings, misallocated resources, false confidence, and ultimately, security risks.

## 3.1 Bugs, weaknesses, vulnerabilities, and exploits

A **bug** is any defect that causes software to behave incorrectly. Most bugs have no security consequence. A **weakness** is a category of mistake that can lead to security problems, such as failing to validate input length. Weaknesses are cataloged in the Common Weakness Enumeration (CWE), maintained by MITRE. A **vulnerability** is a specific, concrete instance of a weakness in a specific software and version that an attacker could actually take advantage of. Vulnerabilities get CVE identifiers. An **exploit** is working code or a working technique that turns a vulnerability into a compromise. A vulnerability can exist for years with no public exploit, and a published exploit changes its risk profile (severity) overnight. Understanding these distinct concepts is critical for prioritizing remediation spend.

## 3.2 Key Security Acronyms

**CVE (Common Vulnerabilities and Exposures)** is the global dictionary of publicly disclosed vulnerabilities, operated by MITRE under sponsorship of the U.S. government (cve.org). A CVE record establishes identity: one ID, one vulnerability, so that every tool, advisory, and conversation refers to the same flaw. CVE IDs are assigned by CVE Numbering Authorities (CNAs), a federated set of vendors, open source projects, security companies, and coordinators authorized to assign IDs within their scope. HeroDevs is a CNA.

**NVD (National Vulnerability Database)** is NIST's enrichment layer on top of CVE (nvd.nist.gov). NVD adds CVSS scores, CWE mappings, and affected-product (CPE) data. NVD has experienced significant enrichment backlogs in recent years, which is one reason many programs no longer treat it as their sole source of truth.

**EUVD (The European Union Vulnerability Database)** is a centralized platform maintained by the European Union Agency for Cybersecurity (ENISA) to track, categorize, and address software and hardware vulnerabilities relevant to the EU. Created to support the NIS2 Directive and the Cyber Resilience Act, it aggregates global feeds (like CVE and NVD) while directly ingesting reports from European national CSIRTs (Computer Security Incident Response Teams) and regional vendor disclosures.

**CVSS (Common Vulnerability Scoring System)**, maintained by FIRST, expresses the technical severity of a vulnerability on a 0 to 10 scale based on exploitability and impact metrics (first.org/cvss). Scoring differs between CVSS v3.1 and v4.0. It says nothing about whether anyone is actually exploiting the flaw or whether your deployment is reachable.

**EPSS (Exploit Prediction Scoring System)**, also from FIRST, estimates the probability that a vulnerability will be exploited in the wild within the next 30 days (first.org/epss). EPSS is a probability, not a severity. A low-severity flaw can carry a high EPSS score and vice versa.

**KEV (Known Exploited Vulnerabilities Catalog)** is CISA's authoritative list of vulnerabilities with confirmed exploitation in the wild, each carrying a required remediation deadline for U.S. federal agencies (cisa.gov KEV catalog). KEV is the floor of any prioritization scheme: if a finding is in KEV and in your environment, it should go to the front of the queue.

**SBOM (Software Bill of Materials)** is a machine-readable inventory of the components inside a piece of software, typically in SPDX or CycloneDX format. CISA maintains the central collection of SBOM guidance and community resources (cisa.gov/sbom).

**VEX (Vulnerability Exploitability eXchange)** is the companion format that lets a supplier state whether a product is actually affected by a given CVE in its components. VEX exists because SBOMs generate enormous numbers of theoretical matches, and someone has to communicate which ones matter.

## 3.3 The Advisory Landscape

CVE records and NVD entries are only part of the picture. Vendor advisories often carry the earliest and most accurate affected-version data. Open source ecosystems maintain their own advisory databases, including the GitHub Advisory Database and the OSV.dev aggregation layer, and these frequently disagree with NVD on version ranges and severity.

## 3.4 Zero-day and Forever-day Vulnerabilities

A **zero-day** is a vulnerability with an exploit available before the vendor or maintainer knows about it or has shipped a fix. Some vulnerabilities were so impactful they earned their own names: Spring4Shell, Dirty Pipe, Log4Shell, Shellshock, Heartbleed, though several had fixes available within days of public disclosure.

A **forever-day** is a vulnerability in software that has reached EOL. Because abandoned, unsupported, and EOL software receives no security fixes, no further releases, and no patches, a forever-day has a higher risk of exploitation. It remains exploitable in every deployment, indefinitely, unless the operator does something other than wait for a patch. **Determining which components in an inventory have actually crossed that line is itself nontrivial, because most open source packages never formally announce EOL.**

# 4 ROOT CAUSES: HOW VULNERABILITIES GET INTO CODE

Vulnerabilities rarely enter code out of malice; software complexity, legacy dependencies, and modern delivery speeds inevitably introduce systemic risk. The overwhelming majority of vulnerabilities fall into a small number of durable classes that have been understood, documented, and taught for a long time.

**Memory corruption.** Buffer overflows, use-after-free, double free, and out-of-bounds reads and writes, concentrated in C and C++ codebases. These remain the dominant class in operating systems, browsers, and native libraries, and they are the primary target of the industry's memory-safety push toward languages like Rust (championed by CISA, NSA, and others).

**Injection.** SQL injection, command injection, LDAP injection, and XML injection all share one root cause: attacker-controlled data crossing into an interpreter without adequate separation between data and code. Injection has appeared in every edition of the OWASP Top 10 since the list began over 20 years ago.

**Cross-site scripting (XSS).** XSS accounted for more than 8,000 CVEs in 2025 alone. Three decades after its discovery, XSS persists because output encoding is easy to get wrong in any one of the thousands of places a web application writes user data into a page, and because ecosystems with low barriers to entry keep producing new code that repeats old mistakes.

**Deserialization.** Unsafe deserialization of attacker-supplied data leads to some of the most severe remote code execution vulnerabilities on record, particularly in Java and .NET ecosystems where rich object graphs can be turned into gadget chains.

**Authentication and access control failures.** Broken access control is the top category in the current OWASP Top 10. Missing trust boundaries, incomplete authorization design, unsafe defaults, and unsafe API use produce vulnerabilities that no memory-safe language will ever prevent, because they are design flaws, not implementation flaws.

**Cryptographic misuse.** Rarely a broken algorithm; almost always a correct algorithm used incorrectly: hardcoded keys, disabled certificate validation, homegrown constructions, weak randomness.

## 4.1 Language and Ecosystem Risk Profiles

Every major ecosystem fails in its own recognizable way. C and C++ codebases are known to produce memory corruption: buffer overflows and out-of-bounds access vulnerabilities.

Java's vulnerability signature is unsafe deserialization and complex framework-level flaws, and the Spring ecosystem illustrates the volume: June 2026 alone brought 67 Spring CVEs, 27 of them high severity.

Many of the vulnerabilities for JavaScript and the npm ecosystem fail through supply chain exposure and prototype pollution, amplified by extreme dependency depth. The May 2026 Mini Shai-Hulud campaign is the defining recent example: attackers published 84 malicious versions across 42 TanStack packages in a six-minute window by poisoning a GitHub Actions cache.

PHP code is prone to produce injection and file inclusion flaws, concentrated in the plugin ecosystems of WordPress and Drupal. Python produces deserialization (pickle), dependency confusion, and, increasingly, vulnerabilities in AI and ML tooling.

Knowing your stack's signature vulnerabilities lets you weigh your review effort where your ecosystem tends to fail; nonetheless any type of vulnerability can be discovered on any language or framework at any time.

## 4.2 Where to Go Deeper

Two references cover the prevention side of this chapter far more thoroughly than an ebook can. The OWASP Application Security Verification Standard (ASVS) is the working checklist for verifying application-level controls. The NIST Secure Software Development Framework, SP 800-218 defines the development-lifecycle practices that regulators increasingly reference by name.

# 5 THE VULNERABILITY END-TO-END LIFECYCLE

Every vulnerability follows a lifecycle, from creation and discovery through disclosure and, ideally, remediation. Understanding this path, including where it can stall or break down, is a fundamental aspect of application security knowledge.

## 5.1 Discovery Paths

Vulnerabilities surface through four main channels, each with different characteristics.

**Internal research.** Vendor and open source project security teams reviewing their own code, running static analysis, and fuzzing their own targets. Highest signal, earliest in the lifecycle.

**Bug bounty and external researchers.** Independent researchers reporting through programs on platforms like HackerOne, or directly to maintainers. Quality varies enormously; for instance, AI-generated low-quality reports have recently strained this channel to the breaking point for some open source projects including Node.js. This was discussed in a recent HeroDevs webinar.

**Fuzzing at scale.** Coverage-guided fuzzing infrastructure such as Google's OSS-Fuzz continuously exercises hundreds of critical open source projects and has produced tens of thousands of findings over its lifetime.

**AI-assisted analysis.** The newest channel and the fastest growing. Frontier language models are now capable of reading a codebase, hypothesizing where it might break, experimentally validating those hypotheses against the running software, and producing confirmed, reproducible findings. The Anthropic, Mythos Preview assessment provides evidence in detail from just one of the LLMs.

## 5.2 Validation and Reporting

A finding is not a vulnerability until it is validated: reproduced against the actual software, with a clear statement of impact and affected versions. The gap between "my tool flagged this" and "here is a reproducible security issue with demonstrated impact" is where most low-quality reports die, and where reporter credibility is earned.

A good report includes the affected component and version range, reproduction steps or a proof of concept, an impact assessment, and a suggested severity with reasoning. Anthropic's published operating principles for AI-discovered vulnerabilities are a useful modern template: every report reflects human review and confirmation, AI-originated findings are labeled as such, and candidate patches are included where possible (Anthropic, Coordinated Vulnerability Disclosure).

## 5.3 Coordinated Disclosure, Embargoes, and Timelines

Coordinated vulnerability disclosure (CVD) is the negotiated process by which a reporter gives a vendor or maintainer time to develop and ship a fix before details become public. The definitive process reference is the CERT/CC Guide to Coordinated Vulnerability Disclosure, and the underlying standards are ISO/IEC 29147 (disclosure) and ISO/IEC 30111 (handling processes).

The de facto industry norm is a 90-day disclosure deadline where the reporter publishes after 90 days or after a patch ships, whichever comes first, with extensions and accelerations negotiated case by case. Embargo periods exist to coordinate multi-vendor fixes, especially for vulnerabilities in shared components where dozens of downstream products need to patch simultaneously.

Disclosure norms assume there is someone on the other end with the capacity to build a fix. For a healthy and active project, the lifecycle completes: report, embargo, patch, advisory, adoption. For an end-of-life open source project, the lifecycle terminates early. There is no one to report to, or the maintainers who receive the report will state, accurately, that the release line is unsupported. **The disclosure completes; the remediation never begins. This is the mechanical reason unsupported and end-of-life software accumulates permanently unpatched, publicly documented vulnerabilities: disclosure keeps happening, and fixing has stopped.**

Clear visibility of long-term support and EOL dates is a recommended best practice to provide visibility to software developers. The framework HeroDevs publishes for open source projects through its Open Source Sustainability Fund addresses this from the maintainer side: communicate end-of-life events through official channels, document which versions are supported versus unsupported, and point users toward migration or a support solution before the EOL date arrives.

## 5.4 Patch Release and Public Disclosure

When the lifecycle works, the endgame is a patch release accompanied by a security advisory: the fix, the affected versions, the severity, and credit to the reporter. Unfortunately, from that disclosure, attackers can go after unpatched environments and exploit them. The five-day median time from disclosure to exploitation window according to CSA, opens against every deployment that has not yet been updated. And for EOL open source software that patch never arrives unless you opt for a commercial extended support option.

# 6 VULNERABILITIES ON EOL OPEN SOURCE SOFTWARE

Vulnerability management, as practiced and as regulated, is built on a hidden assumption: that for any given vulnerability, a fix exists or will soon exist, and the operator's job is to apply it quickly. EOL software breaks that assumption completely.

Abandoned, unsupported, or EOL open source software refers to projects that are no longer actively maintained or supported by their original developers or community. These projects no longer receive security patches, bug fixes, or compatibility updates, leaving organizations exposed to known vulnerabilities and increasing the risk of security and operational issues. Every CVE published against it is permanently unfixed through normal channels.

## 6.1 Finding EOL Components in Your Own Inventory

You cannot manage this risk class without knowing where it lives, and most vulnerability scanners do not answer the question, because they are keyed to CVE data rather than lifecycle status. A component with zero known CVEs and zero remaining maintainers looks clean to a security scanner and is a time bomb in reality.

Lifecycle-aware tooling exists to close this gap. The HeroDevs EOL Dataset checks packages against a dataset tracking more than 19 million package versions across every major ecosystem (npm, Maven, etc), returning an EOL determination per component. And for a full complete platform functionality HeroDevs Evergreen automatically covers every EOL dependency in applications with secured, engineer-built replacements. Replacements arrive as pull requests to review and merge. It keeps covering more as dependencies lose support later get picked up.

## 6.2 The Decision Tree: Migrate, Mitigate, Accept, or Buy Extended Support

Once an EOL component is identified, exactly four options exist.

1. **Migrate.** Move to a supported version or a replacement technology. This is the permanent fix and the right long-term answer, but real migrations of framework-level dependencies are measured in engineer-sprints and forcing them on an emergency timeline after a critical CVE is the most expensive possible way to do them. The operation of the organization or business is also impacted by forced migrations.

2. **Mitigate.** Apply compensating controls: WAF rules, network segmentation, feature disablement, and even environment isolation. Legitimate as a bridge, risky as a destination, because each control addresses known exploitation paths of known CVEs while the underlying flaw, and every undiscovered flaw, remains.

3 **Accept.** Document the risk and consciously carry it. Sometimes defensible for genuinely isolated, low-value systems. Rarely defensible for anything internet-facing or in a regulated industry. Several regulatory regimes effectively remove this option.

4 **Third-Party Extended support.** Commercially maintain EOL open source software through dedicated security engineering. HeroDevs employs open source experts and core contributors who continuously monitor upstream advisories and evaluate CVEs against customer EOL versions. They develop, test, and ship backported fixes as drop-in replacements, without requiring application code changes. The same expertise cuts the other way when a headline CVE does not apply to EOL software. When Drupal core disclosed a highly critical SQL injection in 2026, the upstream advisory simply did not evaluate Drupal 7, and it took a HeroDevs expert audit of core and contributed modules to confirm that Drupal 7 deployments were not exposed (HeroDevs blog, Is Drupal 7 affected by CVE-2026-9082?).

The four options are not mutually exclusive; the common mature pattern is extended support or mitigation as the bridge, migration as the destination, and acceptance only for explicitly bounded exceptions with expiration dates.

## 6.3 Compliance Exposure

A dedicated compliance analysis is warranted. HeroDevs' white paper, The Developer's Guide to Security Compliance, examines global standards, frameworks, and regulations, concluding that EOL software inherently fails their core requirements on three primary counts:

1 **Remediation Timelines:** Standard compliance windows (such as mandated 30-day remediation mandates for critical vulnerabilities) presuppose a patch exists.

2 **Inventory & Asset Visibility:** Generating a Software Bill of Materials (SBOM) immediately exposes active, unsupported components to regulatory scrutiny.

3 **Documented Governance:** Required risk-management processes cannot be satisfied for vulnerabilities lacking a defined remediation path.

Consequently, an auditor discovering an EOL component with an unpatched, critical CVE within scope is likely to record a non-compliance finding.

# 7 AI-ASSISTED VULNERABILITY DISCOVERY AND EXPLOITATION

The strongest public evidence comes from Anthropic's security research team, which published a technical assessment of its Claude Mythos Preview model in April 2026 after roughly a month of internal testing. The model reads code to form hypotheses, runs the software to confirm or reject them, uses debuggers as needed, and outputs validated findings. Using this approach the model discovered genuine zero-day vulnerabilities, primarily memory safety flaws, across widely used open source software, and demonstrated the ability to build working exploits, including chaining multiple browser vulnerabilities together (Anthropic, Assessing Claude Mythos Preview's cybersecurity capabilities).

The scale is the key point. Claude Mythos and comparable LLMs from OpenAI, Google, and others can rapidly scan codebases and identify vulnerabilities. At the same time, AI-assisted tools can help researchers accelerate remediation, but not at the same pace. And that is only the coding part of the process. Fixes still need to be triaged, packaged, documented, tested, and ultimately shipped.

## 7.1 Vulnerability Exploits

Analysis of the Mythos results by Wiz notes that the model can take a CVE identifier and a commit hash as input and autonomously produce a working exploit within hours at low cost, where the same work historically took skilled researchers days to weeks. The practical consequence: more CVEs disclosed and exploited faster. With AI-assistance every published vulnerability remediation can become an exploit-development starting point.

Now apply that to abandoned, unsupported or EOL software, and the asymmetry becomes stark. For supported software, faster exploitation is answered by faster patching; the defender's window shrinks but still exists. **For abandoned, unsupported and EOL software, there is no defender's window at all, because there is no patch to race toward.**

AI has permanently inverted the defender's advantage. Frontier models now perform autonomous vulnerability research and exploit generation at machine speed and marginal cost. For active software, this shrinks the patching window; for EOL and unmaintained software, it eliminates defense entirely, turning legacy technical debt into active breach potential.

If the industry's discovery rate is about to be multiplied by machine-scale analysis, the population of software that structurally cannot receive fixes is where that multiplication does the most damage. This is the single strongest argument for treating EOL inventory as an urgent, board-level topic rather than background technical debt.

# 8 FINAL THOUGHTS

The old vulnerability playbook rested on three assumptions: disclosures arrive at a manageable pace, a patch exists or soon will, and defenders have time to apply it. All three are now false. In the new vulnerability landscape, disclosure volume is heading past ~66,000 per FIRST's mid-year forecast, AI-assisted discovery is converting codebases into findings at machine speed, and the median exploitation window has collapsed below every compliance clock.

For supported software, this is a hard operational problem. For EOL software, it is something categorically worse: a growing class of vulnerabilities that are publicly documented, permanently unpatched, and increasingly discovered by AI. Every acceleration described in this ebook, more CNAs, more findings, faster exploits, lands hardest on the software that structurally cannot receive a fix.

The response is not panic; it is inventory and sequencing. Know which components in your stack have crossed the EOL line (most never announce it). Put lifecycle status next to vulnerability status in the same view. Then apply the decision tree deliberately: migrate as the destination, extended support or mitigation as the bridge, and acceptance only for bounded exceptions with expiration dates.

The frequency of AI-discovered vulnerabilities is multiplying the risk of EOL software. Forever-days only stay forever if no one is shipping fixes. HeroDevs Never-Ending Support restores security patching for 40+ EOL open source frameworks, runtimes, and libraries, delivered as drop-in replacements under SLA. If any component in your inventory has crossed the EOL line, talk to our team, before the next disclosure cycle finds it first.