

Fixing vulnerabilities at the source

Shift left to catch bugs
before they ship



bugcrowd

Introduction

In 2026, Verizon reported that [software flaws had dethroned stolen credentials as the primary entry point](#) for breaches for the first time ever, accounting for 31% of breaches. Fixing these vulnerabilities isn't getting any easier. Verizon further found that the median time to remediate is [43 days](#), a 44% increase from 2025.

A major driver of this is AI; the increase in developer productivity has led to [more code being shipped](#) and an expanded attack surface. At the same time, attackers are [using AI at an unprecedented pace](#) to scan attack surfaces and execute attacks in days, not months.

To keep pace, security leaders need to embrace preemptive security, continuously scanning their attack surface for vulnerabilities and fixing them before attackers can exploit them. That takes three things working together:

- ✓ **Catching vulnerabilities before they ship (shift-left testing)**
- ✓ **Continuously monitoring the attack surface for anything that slips through**
- ✓ **Validating which vulnerabilities are actually exploitable so teams know what to fix first**

Implementing one of these pieces, let alone all of them, isn't straightforward. Traditional AppSec tools are noisy, which buries security and development teams in alerts.

This guide focuses on the first piece of that strategy: **catching vulnerabilities before they ship**. We'll walk through some of the challenges, explain how fuzzing fits in, and how Bugcrowd can help.



The current state of application development

Modern applications are often built by patching together many different components, including first-party code, open-source dependencies, third-party libraries, and APIs. A vulnerability can exist in any of these layers, giving attackers multiple entry points.

Keeping on top of everything is nearly impossible. Every dependency a developer adds can pull in its own dependencies, so it's difficult to trace the full chain without manually digging through the source of every package. Applications built with open-source software are especially vulnerable to this problem. Since it's maintained by a community of volunteers, organizations often inherit whatever security flaws it comes with.

In an [audit of open-source codebases](#), 87% contained at least one vulnerability, 78% contained high-risk vulnerabilities, and 44% contained critical-risk issues.

95% of the examined software had at least one vulnerability

One in five of those vulnerabilities was high risk

As a result, many applications have more vulnerabilities than organizations realize. A recent [analysis](#) revealed that 95% of the examined software had at least one vulnerability, and one in five of those vulnerabilities was high risk.

These vulnerabilities also tend to live in the codebase for a long time. Veracode's [2026 benchmark](#) puts the overall vulnerability fix half-life at 243 days, extending to 358 days for third-party and SCA findings.

Meanwhile, organizations are under even more pressure to ship more features. **Many are turning to AI agents to take over day-to-day coding and are skipping key parts of the development process**, such as code and security reviews. However, the accumulation of unaddressed vulnerabilities and missing security controls creates security debt, increasing the risk of a breach or attack. Security debt is similar to tech debt, but with an increased risk of a breach or attack.

Traditional AppSec tools no longer meeting the mark

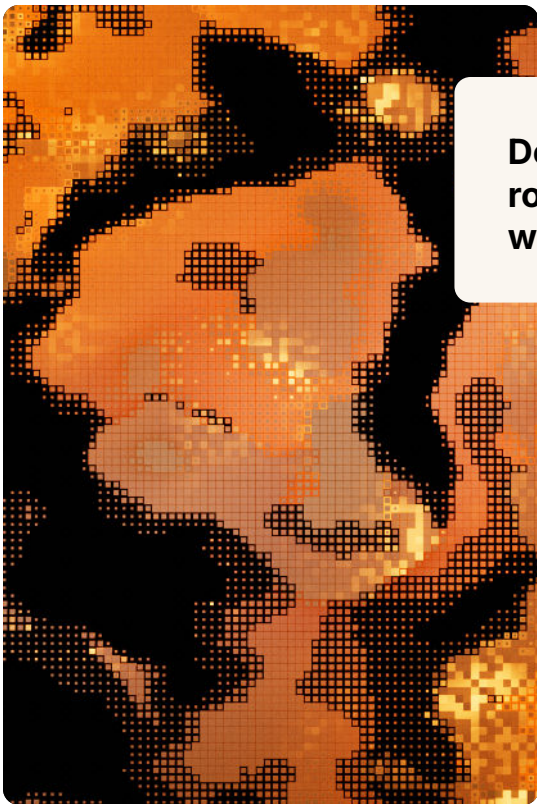
Shift-left testing only works if the tools doing the testing give developers something they can act on. Catching an issue earlier doesn't help if it still takes hours to figure out whether the issue is real. Instead, it's important to test so you can prove a vulnerability is actually exploitable before it ever reaches a developer's queue, making the fix the only work left to do.

Therein lies the issue: Most automated security tools are focused on finding known vulnerabilities and aren't always actionable for developers.

For example, Static Application Security Testing (SAST) inspects code for any known vulnerabilities, such as unsafe functions, injection risks, or insecure data flows. When used well, SAST still earns its place in the development pipeline. In GitHub's [beta](#) of a code scanning feature (i.e., a SAST tool), they found that **developers fixed 72% of security errors surfaced in pull requests before merge**, across a sample of more than 12,000 repositories and 1.4 million scans.

72% of security errors surfaced in pull requests before merge





Developers spend roughly eight hours per week testing

About 400 hours per year

55% of that time is spent chasing false positives.

200 hours per developer each year

Unfortunately, not all SAST results are high value, as SAST often returns many findings that aren't exploitable issues. That's because SAST doesn't know how data, configurations, dependencies, or application state behave at runtime, so it can flag issues that aren't actually problematic.

The time spent on false positives and low-value alerts adds up. Developers spend roughly eight hours per week (about 400 hours per year) testing, and 55% of that time (200 hours per developer each year) is spent chasing false positives.

Over time, that leads to alert fatigue, which limits the effectiveness of shift-left testing and [hurts adoption](#).

Even when developers find a high-value alert, taking action isn't straightforward. They need to reproduce the failure to debug it, but most tools don't provide the exact input, execution path, or application state that caused the failure. **Developers have to spend a lot of time on setup to reproduce and triage the issue before they can write a single line of code to fix it.** Given how quickly developers are moving, it's easy to see why they might skip investigating the alert altogether.

This friction also strains the relationship between security and engineering teams. Without alignment on priorities or a shared process for fixing issues (or even which issues to address), vulnerabilities go unfixed, which hurts the organization's security posture.

Using fuzz testing to enhance application security

Fuzz testing closes that gap by running the application itself rather than scanning code for patterns that might be a problem.

It discovers vulnerabilities by injecting a program with invalid, unexpected, or random data to trigger undesirable behaviors (such as crashes, infinite loops, or memory leaks), which can indicate underlying vulnerabilities. Guided fuzzing tools often **use dynamic analysis and symbolic execution** to systematically explore program paths and generate inputs that surface deep vulnerabilities.

As a result, fuzz testing is effective at uncovering novel vulnerabilities—or the “unknown unknowns”—that traditional tools tend to miss and you might not know to look for. Since these vulnerabilities are based on your actual code, they’re more likely to be a real issue. **This valuable information reduces alert fatigue** and gives the teams what they need to reproduce the issue and jump immediately into remediation.

Fuzzing is especially effective at uncovering vulnerabilities for the following:

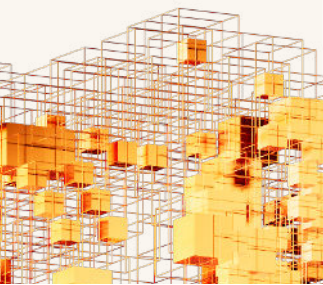
- ✓ Memory-unsafe code (C, C++, and Assembly) in the codebase
- ✓ Complex input parsing from untrusted or external sources
- ✓ Safety or life-critical consequences from software failure
- ✓ High regulatory scrutiny with demonstrable testing requirements
- ✓ Large, legacy codebases with limited original test coverage
- ✓ Network-exposed protocol handlers accepting structured data
- ✓ Long patch cycles, where finding bugs early is far cheaper than post-deployment

But none of this means you should get rid of your other AppSec tools. It's best to take a layered approach and build a toolkit where each tool can help fill gaps.

Fuzz testing complements other automated tools, such as SAST or Dynamic Application Security Testing (DAST), by feeding a running application unexpected input to surface the deep, novel vulnerabilities that pattern-matching and predefined test paths both miss.



Fuzz testing also helps meet certain regulatory requirements



- ✓ [ED-203A](#), which applies to aviation-related organizations, requires meeting a refutation standard. Specifically, organizations are required to show the presence or absence of a vulnerability in a test scenario, which fuzz testing can demonstrate.
- ✓ The EU’s [Digital Operational Resilience Act \(DORA\)](#) requires financial entities to conduct advanced threat-led penetration testing, including vulnerability testing of critical systems, where fuzz testing is an expected component.
- ✓ The [FDA’s Cybersecurity Guidance](#) explicitly lists fuzz testing as an expected security testing method for premarket submissions for medical devices.

	SAST	DAST	Fuzz testing
What it tests	Source code, before the app runs	A running application	A running application by feeding it invalid or unexpected input
When it runs	Early, often at commit or pull request	Later, once a build is deployed to a test environment	Continuously, in the pipeline or against in-progress builds
Finds	Known patterns: unsafe functions, injection risks, and insecure data flows	Exploitable issues visible externally, such as auth flaws or misconfigurations	Novel, deep vulnerabilities: crashes, memory leaks, and edge cases in actual code paths
Common limitation	High false-positive rate; can't see runtime behavior	Limited code path coverage; only tests what's exposed	Requires more compute and setup than a simple scan
Best for	Catching known issue patterns fast, early in development	Validating how the app behaves once it's running	Uncovering unknown vulnerabilities in complex, high-risk, or long-lived code

Introducing Savant Forge, Probe, and Runtime

Savant Forge, Savant Probe, and Savant Runtime—

Bugcrowd's fuzzing tools, formerly known as Mayhem Security—continuously test code, APIs, and software dependencies throughout development. Rather than waiting until a scheduled scan, they help you detect both known defects, such as buffer overflows, and more sophisticated, novel threats, like use-after-free vulnerabilities that only surface once a program is running the vulnerable code path.

These tools offer several advantages over fuzzing or AppSec testing alone. Savant runs your application in containers with standard configurations to show that a vulnerability is reachable under real-world execution conditions.

This keeps security and development teams focused on relevant issues and gives them the information they need to jump-start remediation and build alignment. It also combines fuzzing with symbolic execution, using mathematical reasoning to ensure it reaches all code branches.

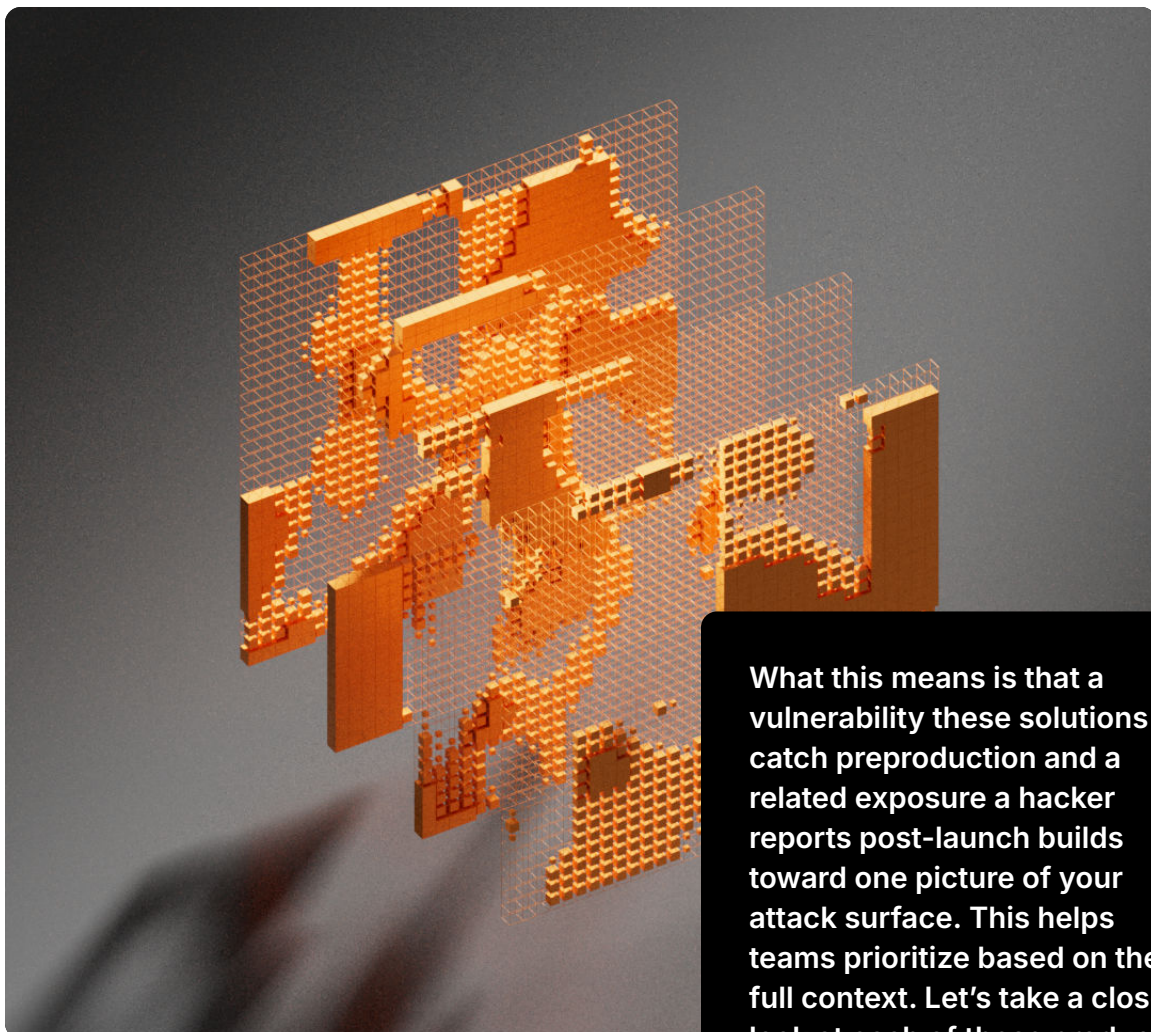


This combination finds twice as many bugs as fuzzing alone. Savant runs continuously in the background by integrating with your CI/CD pipeline. And there's no need to recompile since it works with dockerized images, Windows binaries, and more.

With this full suite, organizations can help detect and fix more threats faster. For example, one automotive manufacturer turned to Savant Forge to cut false positives and focus engineering time on actionable issues.

The team used continuous fuzzing to automate testing that had previously eaten into manual review time, while still meeting the requirements of a heavily regulated industry.

These three products operate across each layer of your attack surface. Savant Forge, Savant Probe, and Savant Runtime make up the Avoid stage of Bugcrowd's Avoid, Discover, Validate approach, catching risk before it ever ships. With these tools on the same platform as Bugcrowd's Discover and Validate capabilities (bug bounty, VDP, pen testing, and red teaming), findings connect across the whole platform.



What this means is that a vulnerability these solutions catch preproduction and a related exposure a hacker reports post-launch builds toward one picture of your attack surface. This helps teams prioritize based on the full context. Let's take a closer look at each of these products and how they can help.

Savant Forge

Formerly Mayhem Code Security, [Savant Forge](#) automates offensive testing to continuously uncover known and unknown vulnerabilities, so you can ship more secure code.

- ✓ **Proof of exploitability:** Get a test case demonstrating exploitability, a prioritization score, and automated regression testing to jumpstart remediation.
- ✓ **Built-in triage:** Automatically deduplicate bugs to keep the focus on unique issues.
- ✓ **Pipeline integration:** Connect SARIF-supported findings with other code and posture management solutions.

Savant Probe

Formerly Mayhem API Security, Savant Probe validates APIs in minutes by finding and prioritizing common vulnerabilities.

- ✓ **Comprehensive coverage:** Automatically catch 500 Internal Server Error responses, server crashes, and common security issues like SQL injections.
- ✓ **Performance insights:** Get a clear picture of which endpoints have inconsistent or degraded performance.
- ✓ **Quality:** Keep your API and spec in sync to shield consumers from unexpected behavior.

Savant Runtime

Savant Runtime ingests reports from static Software Bill of Materials (SBOM) and SCA tools and then builds a profile of your application as it runs, enriching those reports with an accurate picture of the components actually used at runtime.

- ✓ **Actionable insights:** Automatically prioritize issues found by static SBOM and SCA tools, such as unused code and dependencies.
- ✓ **Less noise:** Cut 80% of false positives from static analysis by pinpointing issues that emerge only at runtime.
- ✓ **Faster compliance:** Generate attestations and justifications, integrate with audit tools, and export VEX and SARIF reports that auditors require.

Preemptive security starts here

The best cure for any disease is to avoid getting it in the first place, and security is no different. Bugcrowd Savant Probe, Savant Forge, and Savant Runtime make prevention possible by delivering comprehensive, continuous coverage for software applications and APIs. With real proof of exploitability, teams can focus on legitimate issues and get the right context to jumpstart remediation.

Want to try out Bugcrowd?

Contact us to get started

