Sourcegraph

Developer velocity vs. code security: How organizations can invest in both

"Security is everyone's priority" is received wisdom and presumed consensus that rarely occurs in practice. This tension is most palpable in the conflict between developer velocity and code security, wherein companies try to prioritize both speed and safety—and often achieve neither.

According to research from application security company **Veracode**, 85% of organizations "admit to releasing vulnerable code to production because of time restraints." In other words, more often than not, developer velocity and business objectives win out over security. This fact bears out in people's sentiments, too. A study from cybersecurity company **Snyk** found that even among organizations with "well-integrated" security operations, 33% still report that "security is a major constraint on the ability to deliver software quickly."

Revealed preference is clear: developer velocity is more important than security. **The Software Engineering Institute** reports that 90% of security incidents result from defects in software design or code, meaning the current state of things is not sustainable.

In this article, we'll explore the tension between developer velocity and security, show the shortcomings of DevSecOps, and offer a tooling-focused strategy for having your cake and eating it, too.

Software vulnerabilities: An exemplar of the developer velocity and code security tension

The Snyk research cited above shows, on an abstract level, that security constrains software delivery. With that fact in mind, let's sketch a prototypical workflow for a security engineering team and development team and see how that fact plays out.

For this example, we'll focus on a particular kind of workflow: vulnerability management. This workflow is an ideal example because it shows exactly when and how the handoff between security and development breaks down.

The prototypical vulnerability management workflow reveals fundamental problems

Before joining Sourcegraph, one of our security engineers worked on the security team at a Fortune 500 oil and gas company and experienced a fairly typical vulnerability management workflow. The security team at this company adopted a mature software composition security scanner and was regularly running security scans to find known vulnerabilities.

As a result of these composition scans, the security team would regularly block developers from installing packages that the scanner indicated contained vulnerabilities. Typically, this blocking prompted the developer to ask a security engineer to investigate the package and determine whether the developer could or could not actually install it.

A security engineer would then have to jump into parts of the codebase they likely weren't familiar with to determine the security of the desired package and how it would interact with the rest of the codebase. Generally, that security engineer would err on the side of caution and tell the developer "no."

Developer velocity would suffer as developers searched for different packages, and developer happiness would plummet as developers regularly ran into roadblocks that broke their flow.

This example is how a lot of companies run their vulnerability management programs, given a couple of variations:

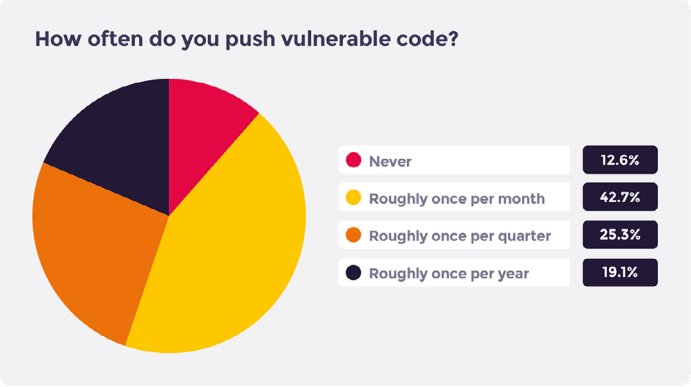
- On the negative side, some organizations skip the triaging step and hand off all the vulnerabilities to the development team, asking them to either fix all the vulnerabilities or justify the existence of those vulnerabilities.
- On the positive side, some organizations add Static Application Security Testing (SAST), which developers can use to proactively scan their code from inside their IDEs for potential vulnerabilities as they work.

Despite these variations, the general structure is common across companies: security engineers block and delay developers on behalf of security and at the cost of developer velocity; meanwhile, developers try to build with security in mind but struggle and let developer velocity win out more often than not.

False positives make security feel like an unproductive distraction

Security might delay development, but it's at least worthwhile, right? Unfortunately, especially in the case of vulnerability management, security tools produce a plethora of false positives that can make security delays feel like wastes of time and effort.

A study from cloud service provider Fastly shows, for example, that web application and API security tools produce an average of 53 alerts per day—45% of which are false positives. And research from developer-first application security management platform Tromzo shows that about a quarter of developers find that 50% to 75% of vulnerability findings are false positives. Additionally, close to a fifth of developers say that more than 75% of vulnerability findings are false positives.



(Source: Voice of the Modern Developer: Insights From 400+ Developers, Tromzo)

According to one study, vulnerabilities, in general, suffer from "over-inflation." Researchers argue that "about 20% of the dependencies affected by a known vulnerability are not deployed, and therefore, they do not represent a danger to the analyzed library because they cannot be exploited in practice." In other words, a fifth of dependencies that do contain a known vulnerability, meaning a security tool might detect it, are effectively false positives because they are not actually deployed and do not actually present a threat.

Josh Bressers, VP of Security at Anchore, a software supply chain management platform, refers to the work of filtering out false positives as "**backbreaking manual labor**." Every vulnerability result, he says, must be filtered through five questions:

1. Do you actually include the vulnerable dependency?
2. Is the version you're using affected by the issue?
3. Are you using the feature in your application?
4. Can attackers exploit the vulnerability?
5. Can attackers use the vulnerability to cause actual harm?

Whether security engineers or developers do this filtering work is organization-dependent, but either way, false positives threaten to significantly worsen the already likely development delays that security work incurs.

Beyond their financial cost, delays also tend to be costly for morale and collaboration. If development teams resent the security requirements, and perhaps even the security engineers themselves, then organizations will have a recipe for a slow, unhappy developer experience and a slow software delivery process.

Security isn't the first field to raise the problem of cross-team resentment. On behalf of operations teams working with development teams, DevOps has tread this ground already, which is why we should turn to DevSecOps to see if that philosophy has a solution for security and development conflict.

The limitations of DevSecOps and the shift left

DevSecOps and shift-left security are, depending on your perspective: a) jargon, or b) visionary mission statements. Either way, organizations are adopting both as core values. Adoption offers unassailable benefits, but as the industry integrates these philosophies, it will also benefit us to examine where they fall short.

DevSecOps and shift left: Jargon or meaningful?

"DevSecOps" is a controversial term both because of its implication that DevOps doesn't already include security and because it's another term to add to a seemingly endless list.

In its **2021 State of DevOps Report**, infrastructure automation company Puppet splits the disagreement into two camps: Those who believe the "sec" in DevSecOps is a necessary call to action that drives people to include security in the software development lifecycle, much like "ops" before it, and those who believe security is already a part of the DevOps way of work.

A similar idea is the term "shift left security," which **Google describes** as the process of "better integrating information security (InfoSec) objectives into daily work, [so that] teams can achieve higher levels of software delivery performance and build more secure systems."

The term originates from the software development lifecycle, which typically has four stages: design, develop, test, and release. Security typically happens in the testing stage, after development is complete, so to "shift left" is to consider security concerns earlier in the process. The primary goal of shifting left is to prioritize security and avoid finding security problems so late in the process that issues require expensive re-architecting.

Of course, shift left has its detractors as well. Rickard Carlsson, co-founder and CEO of Detectify, a SaaS security company powered by ethical hackers, argues that in a world of continuous delivery, **you can't shift left**: "There's really no more 'earlier' and 'later.' There's just 'now.'"

Whether jargon or inspiration, the terms are likely to persist because of the benefits they point to. One of the driving factors behind the adoption of DevSecOps and shift-left security is the dynamic this data point from the **2016 State of DevOps Report** reveals that high-performing companies spend 50% less time remediating security issues than low-performing companies.

> "By better integrating information security objectives into daily work, teams achieve higher levels of IT performance and build more secure systems."

Continuous delivery software company **CloudBees** visualizes the transformed process via two graphs in its "DevSecOps: Speed and Security Together, at Last" whitepaper. The first graph shows the traditional software development flow, wherein security comes in toward the end of the process.

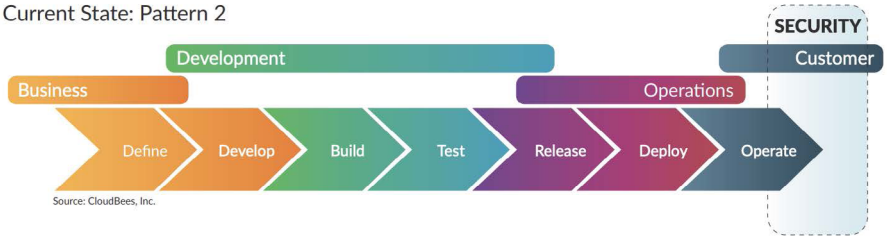
Research from cloud security company Threat Stack validates this visualization, finding that 38% of organizations use a security team that is completely separate and only bring them into the process as necessary.

The second graph from CloudBees shows the transformed version, wherein security is not only shifted left but dispersed throughout the entire process.

One benefit of the former model is the predictability of its sequential steps, which points to one defect in the latter model: dispersing security work threatens to distract from and delay development work in progress. This flaw doesn't make DevSecOps any less worth pursuing, but accounting for it is necessary for organizations that want to maintain a healthy, efficient development experience.

Figure 2

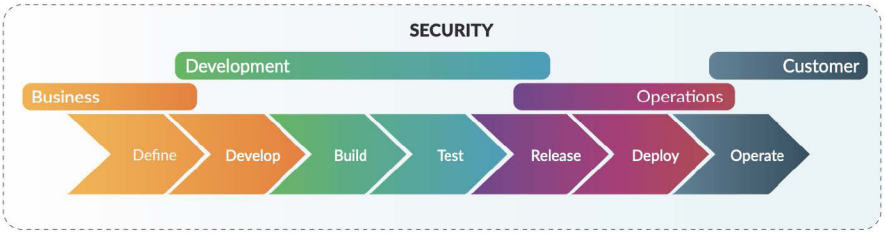
Current State: Pattern 2



Source: CloudBees, Inc.

Figure 3

Ideal State: Pattern 3



Source: CloudBees, Inc.

Shifting security left threatens to exacerbate development delays

DevSecOps and left-shifted organizations test security concerns daily and involve everyone in the security process, rather than the traditional way, which calls almost exclusively on security professionals and typically occurs near the end of the software delivery process.

In real terms, shifting left requires teams to integrate security testing and controls into the everyday work of software development, operations, and quality assurance—most of which will be an automated part of the deployment pipeline. However, only so much of this work can be automated away.

The 2016 State of DevOps report, for instance, mentions testing and controls, both of which can be automated, but also mentions security reviews for major features, noting that teams should "[ensure] that the security review process does not slow down development."

Building on this report in a separate post, Google emphasizes this requirement, writing that the team should "integrate security review into every phase" and that security should be part of the "daily work of the entire software delivery lifecycle." Google also writes that teams should ensure the security review process doesn't slow down development.

"Ensure," here, is doing some heavy lifting: how can teams ensure security doesn't slow down development? As we saw in the previous section, when we look at real, on-the-ground workflows, security does tend to slow down development. And as we saw in the intro, according to Snyk research, security is indeed a major constraint on development velocity. To make matters worse, Threat Stack research shows that 44% of developers have not been taught to code securely.

Shifting security left threatens to worsen development delays: more security reviews, happening more often, means more development delays. When you account for the nature of flow state, wherein developers achieve peak productivity from time spent in deep focus, more interruptions from security can actually result in exponentially larger delays.

There's no pretending that there's "one weird trick" or one solution that will make security and development move as one, that will ensure security never slows down development and development never runs off without security. What we can do, however, is double or triple down on a strategy we know has the greatest payoff: tooling.

The simple solution is the right one: You need better tooling

The Pareto principle, otherwise known as the 80/20 rule, dictates that roughly 80% of outcomes tend to come from 20% of causes. Applying this to technology strategy means optimizing for greater results by identifying and doubling down on that 20%, the so-called "vital few." Our case is that better tooling is a 20% cause, and investment in it will net more significant benefits than any other investment for both developer velocity and security.

The case for and against more tooling

At first glance, developer velocity and security provide contradictory cases for more tooling.

If you're prioritizing developer velocity, more and better tooling is an obvious good. McKinsey research shows that four factors correlate with companies that have the highest developer velocities: tools, culture, product management, and talent management. McKinsey notes, however, that developer tooling is an outlier: "Our research shows that best-in-class tools are the top contributor to business success… Yet only 5 percent of executives recognized this link and ranked tools among their top-three software enablers."

The case is clear: If you want greater developer velocity, invest in developer tools.

However, if you're prioritizing security, more tooling is a less obvious strategy. As it stands now, companies likely already have too many security tools. According to Tromzo research, 65% of developers use 11 or more security tools. IBM research shows that, on average, enterprises deploy 45 cyber-security-related tools. Interestingly and counterintuitively, enterprises with over 50 tools consider themselves to have an 8% lower ability to detect threats than companies with fewer tools. Tool sprawl for security engineers and developers is real.

So then the case isn't clear, is it? If you want security, more tools might make things worse.

The cases for and against tooling come into direct conflict because tooling problems tend to compound. According to Veracode, most developers are releasing code twice as fast as they used to. Increasing code velocity has convinced many companies to adopt more security tools and use them more often.

However, this creates another problem. ESG survey data shows that 27% of cybersecurity professionals report that their security products "generate high volumes of security alerts," which makes prioritization and investigation difficult. Security alerts, and security in general, become noise that developers ignore in the pursuit of a flow state and greater developer velocity.

Developers adopt tools that increase velocity, but at least so far, when security teams adopt tools to layer in security, velocity tends to suffer, and cross-team resentment tends to rise. Tooling, then, is at the center of the tension between development velocity and code security.

How can we resolve this contradiction? Can companies get both greater developer velocity and security?

The bottleneck has moved, and so must tooling investment

The solvent for this contradiction is deceptively simple. Dotan Nahum, Co-founder & CEO of developer-first cloud security company Spectral, captures it in a pithy line: "We need developer tools for security and not security tools for developers." He lists 16 things a developer tool suited for security must have. Still, the spirit of the message is that we can enable greater security by building security tools with developer velocity as a first principle.

Let's return to the prototypical vulnerability management workflow explained at the top of this article. In this workflow, a security engineer blocked developers from using a package because their security scanner warned it contained something vulnerable. Developer velocity plummeted, and the organization, due to the aforementioned high rate of false positives, was likely no more secure.

In the spirit of DevOps, let's apply the Theory of Constraints (owing to "The Phoenix Project," one of the most popular introductions to the world of DevOps, which borrows from "The Goal'' by Eliyahu M. Goldratt). The Theory of Constraints calls on us to visualize a workflow like a conveyor belt to see where in the process work builds up. When we find built-up work, we've discovered a bottleneck or constraint and can improve the system drastically by fixing it.

At one point, the constraint for vulnerability management was finding vulnerabilities. Security scanners solved that, meaning the constraint has moved: now, the constraint is triaging vulnerabilities. The previously mentioned Fastly study shows that developers and security engineers receive, on average, 53 alerts per day. According to Anchore VP of Security Josh Bressers, one vulnerability scan can result in a 13,000-page report.

If we're visualizing a conveyor belt, we can imagine a pileup of vulnerabilities at the triage workstation: developers and security engineers alike struggling to prioritize vulnerabilities and thus struggling to remediate them.

Triaging vulnerabilities, of course, isn't the only problem in the world of code security but it is representative of the fact that developers are struggling to use security tools, and their developer velocity is suffering, rather than using developer tools suited for code security, and increasing both developer velocity and code security.

This is how organizations can have their cake and eat it, too: invest in tooling but focus on developer-first tooling that prioritizes developer experience and developer velocity in the pursuit of code security. This balance isn't a compromise because, ultimately, increasing developer velocity and security together ensures there's no resentment between teams and that developers never feel the need to sacrifice security for speed.

No case better exemplifies this dynamic than the critical, ubiquitous vulnerability Log4j. Download our ebook, The 'less is more' approach to finding and fixing security vulnerabilities: How simplicity complements complexity in the pursuit of healthy code, to learn how simple tooling can work with complex tooling to improve your security posture.

about.sourcegraph.com

Sourcegraph is a code intelligence platform that unlocks developer velocity by helping engineering teams understand, fix, and automate across their entire codebase.