Sourcegraph

Less is more

How simplicity complements complexity in the pursuit of vulnerabilities

The ‘less is more’ approach to finding and fixing security vulnerabilities: How simplicity complements complexity in the pursuit of code security

When your company first got word of Log4j and the Log4Shell vulnerability, did you already have all the tools in place to immediately fix it across every single line of code in your organization? When the next vulnerability of that scale emerges—and it’s “when,” not “if”—will you be ready?

Code security tools are in a dramatically better state today than they were even a few years ago. Companies are shifting left, restructuring for DevSecOps, and investing more than ever in code security. Developers appreciate and collaborate with security teams, and vice versa, like never before.

Despite all this, many companies feel more afraid and more vulnerable than they used to. How can this be? What’s missing from the toolchain and the current security mindset?

Log4j exemplified this discontent. According to Kurt Seifried, Director of IT at the Cloud Security Alliance, Log4Shell—the vulnerability in the **ubiquitous logging library Log4j**—was “like a hurricane vs. a tornado.”

In other words, Log4Shell was both severe and widespread. Its path of destruction caused both immediate harm that organizations had to fix quickly and long-term damage that organizations will be repairing for years to come.

When we think of Log4j now, we should think of both the heroics required to mitigate it and how better prepared we could have been.

We should think of the developers and security engineers who dropped everything to remediate the vulnerability in the days and weeks after and the ongoing monitoring efforts it will continue to require.

We should think too of how we could have set ourselves up not to need heroics. Log4Shell was not the first vulnerability at that scale, and it won’t be the last. What Log4j revealed, however, is that we can’t simply finetune our existing processes. We need a new approach.

In this ebook, we’ll make a case for that new approach, including:

- What the current code security ecosystem is lacking
- Why vulnerability triage is the hidden driving factor in the vulnerability management lifecycle
- How developers and security teams can cross the chasm between 99% and 100% vulnerability coverage

In the end, Log4Shell was as straightforward as it could get: The fix called for a simple dependency upgrade. Given the renewed investment in code security, this situation should have been an easy success, a case study proving industry-wide security efforts worthwhile. And yet, it wasn’t. Why?

Log4j was like an industry-wide test that revealed—if we’re willing to look—all the ways our vulnerability systems and toolchains are lacking and the many ways we can improve.

Table of contents

Chapter 1:

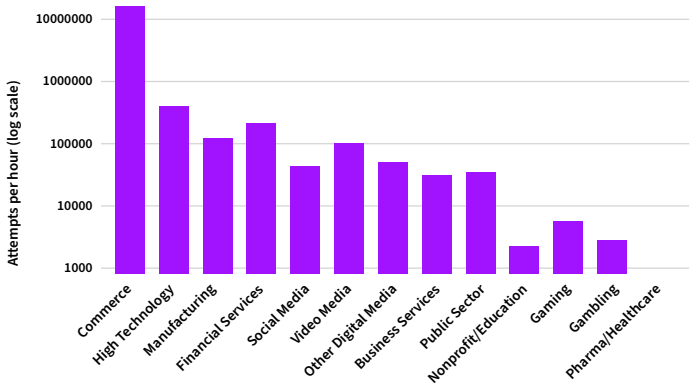
Log4j, a clarifying crisis

On Thursday, December 8, 2021, Apache—a foundation dedicated to supporting numerous essential open source projects—revealed that Log4j, a logging library, had a critical vulnerability.

Given the name Log4Shell, also known as CVE-2021-44228, the flaw soon received a 10/10 CVSS score and became headline news. As the **Federal Trade Commission (FTC) put it**, the threat was severe because Log4j was a "ubiquitous piece of software." The FTC went on to call it a "duty" for organizations to mitigate the risk and "critical" that companies and vendors "act now."

By December 14, hackers had launched over **840,000 attacks** exploiting Log4Shell. Akamai, a content delivery network provider, **broke the attacks out by industry**, finding that attackers hit the commerce, high technology, and financial services industries the hardest. Akamai also found that, beyond the initial exploit, hackers developed an "unprecedented evolution of exploit variants."

Log4j exploit attempts - by vertical



The threat grew as people realized how long Log4Shell would continue to haunt the world. "Log4Shell will be exploited for years because of its ubiquity," Forrester analyst Allie Mellen told **Infosecurity Magazine**. "There are over three billion devices running Java, many of which are using Log4j."

Developers, security professionals, and IT administrators worldwide scrambled to mitigate Log4j. Fixing Log4j required teams to find the library amidst massive codebases riddled with dependencies and deploy hasty patches. Log4j was so ubiquitous that teams couldn't even be sure if they used it or not.

All too many companies had to report to their customers that they were fairly confident—not fully confident—that they had either remediated all instances of Log4j or had verified that they didn't use Log4j.

The heroics of those who patched Log4j shouldn't go unsung. But as the world looks back on its response to the vulnerability, we should take a critical look at what went well, what didn't go well, and why confidence was so often lacking.

In conversations with customers and peers, our discussions kept returning to security tooling, particularly security scanners.

In theory, security scanners, including vulnerability scanners, software composition analysis scanners, and static code analysis scanners, provide a shield against vulnerabilities. In reality, as Log4j proved, even companies that adopted numerous types of scanners found themselves unprepared and ill-equipped.

When Log4j hit, scanner adopters had a lot of sophisticated technology at their disposal but still struggled to move fast enough to address this novel, ubiquitous vulnerability. When security teams looked for Log4Shell, they used these scanners to look for all binaries or artifacts that depended on Log4j. Whether they returned hits or not, the results did not inspire confidence.

Since Log4Shell was a novel zero-day vulnerability, companies who depended entirely on these scanners for vulnerability identification had to wait for their vendors to update their scanners to the novel pattern. And once vendors updated their scanners, users still had to run the scans. Some of our customers who use security scanner tools reported waiting 12 to 24 hours for results.

Ultimately, many users of security scanners couldn't be sure they had found all affected code or that these scanners had covered all the places in the codebase where they had found affected code. Companies that didn't have security scanners were even worse off, still effectively having to do the scanning work, but manually.

Companies familiar with code search, especially if previous employees worked for companies like Google, which has its own code search tool, found themselves in an odd situation. They had invested millions of dollars in complex security analysis tools that, in the case of Log4j, couldn't identify vulnerabilities as comprehensively nor as confidently as a simple text search for the string 'log4j'.

Simplicity, it seems, was more effective than complexity. A complex security scanner tool that promised to scan, notice, and monitor everything wasn't as useful as a simple search. This realization drove many companies to discover their security tooling was still lacking.

Log4j, and the crisis it caused, was a clarifying moment for a still-maturing security industry. What matters most now is taking the opportunity that clarity provides to see how security tooling can evolve so that we're ready for the next Log4j.

Chapter 2:

‘And now what?’: The state of security analysis

Security scanners have been around for decades, but a new generation of security scanners, such as Snyk and GitHub’s Code Scanner, promise to exceed the promises of previous scanners and fix the problems those scanners had.

Before we analyze what the current code security ecosystem lacks, let’s look at the state of the industry today.

Security scanners are eating the world

The number of companies using security scanners, the frequency with which they’re testing dependencies and applications, and the number of scans they’re performing are all on the rise.

Snyk’s **2020 DevSecOps report** shows:

- 65% of companies currently use automated security testing tools for auditing their code.
- 57% of companies test for known vulnerabilities in their open source dependencies.
- 36% of companies perform static application security testing on their code.

Veracode’s **2021 State of Software Security report** shows that companies are performing more scans, scanning more applications, and using more and more scan styles:

- Organizations have tripled the number of apps they scan over the past ten years. As of 2021, organizations are scanning more than 17 new applications per quarter.
- Ten years ago, organizations scanned their applications two or three times a year. In 2021, 90% of applications received a scan more than once every week. A majority of those applications get scanned three times a week.
- Between 2018 and 2021, there was a 31% increase in the usage of multiple scan types.

Despite this rise in automation, manual security reviews appear to remain essential. The Snyk report found that 79% of companies still perform security code reviews.

The three types of security scanners

Security scanners come in many shapes and sizes with equally as many differences and purposes. Still, you can generally break them down into three types:

1. Vulnerability scanners
2. Composition analysis scanners
3. Static application security testing scanners

1. Vulnerability scanners

Vulnerability scanners assess computers, networks, and applications for known vulnerabilities. Many vulnerability scanners can inventory all IT assets and monitor them for vulnerable patterns. You can further break this category down into types of vulnerability scanners, including network-based scanners, host-based scanners, wireless scanners, application scanners, and database scanners.

Some of the top vulnerabilities scanner vendors, **as assessed by G2**, include:

- Burp Suite
- Nessus
- Intruder
- InsightVM
- Tenable.sc

2. Composition analysis scanners

Composition analysis scanners assess an organization’s dependencies for known vulnerabilities. Generally, they work by taking packaged JSON files and pom.xml files and mapping out the dependencies that exist in a codebase. Then, the scanner compares those dependencies against a database of known vulnerabilities and which versions of dependencies the vulnerabilities affect.

Some of the top composition analysis scanner vendors, **as assessed by G2**, include:

- Black Duck
- WhiteSource
- GitHub (specifically, the GitHub Dependabot feature)
- Snyk
- JFrog Xray

3. Static application security testing (SAST) scanners

Static application security testing (SAST) scanners assess an organization's source code for vulnerabilities. Generally, they work by reading source code, either in BIOS or plain text, and finding either insecure code patterns or known vulnerabilities. SAST scanners search for mistakes a developer might have made in the code itself that would eventually lead to a vulnerability.

Some of the top SAST scanner vendors, **as assessed by G2**, include:

- GitHub (specifically, the GitHub code scanning feature)
- GitLab (specifically, the GitLab code scanning feature)
- Coverity
- Checkmarx
- SonarQube

The main challenges of security scanners

The primary challenge for any security scanner vendor is that they have to promise and provide complete coverage, or else the scanner is effectively useless. Scanning half of your codebase or dependencies, even if scanned perfectly, means vulnerabilities might exist in the other half. You can't confidently communicate safety to your business leaders or your customers.

Coverage then isn't a nice-to-have. It's a necessity.

A secondary challenge for security scanner vendors, leading from the primary challenge, is that these scanners must necessarily be complex. You likely have an idea of the complexity, having seen the types and sub-types of scanners in the previous section. Still, we haven't even gotten into the multitude of things these scanners must cover, such as code hosts, build tools, and programming languages.

These two challenges produce a tension between security vendors and their customers:

- To compete with other vendors, security scanner vendors must promise complete coverage and provide as many results as possible to evidence that coverage.
- Security scanner customers, the users of which are typically security engineers and developers, must handle all those results, which is frequently overwhelming.

This tension results in an open secret among security professionals: Security scanners are helpful but insufficient, and using them is often burdensome.

The question is why, and we'll answer via three reasons:

1. Security scanners are rarely comprehensive
2. Security scanners are often noisy
3. Security scanners aren't typically actionable

1. Security scanners are rarely comprehensive

Security scanning is a necessarily complex process, and complex processes inevitably break down—often in numerous ways. Security scanners are no different. They are rarely, if ever, comprehensive, and they frequently suffer from the fact that no single person in a company, including security engineers, knows about all the software in operation. This problem is sometimes referred to as "shadow IT," but the problem exceeds IT.

The source of this deployment problem is that security engineers have to both know about all the software in use and configure the security scanner to ingest and cover that software. A perennial problem recurs here: It's hard to get everyone to adopt the same tool.

Partial adoption can be okay, though troublesome if some people are missing from tools like Slack or Asana. Still, it can be dangerous if some developers either aren't using a security scanner or aren't making their software available to a security scanner.

Consider code hosts. There's a good chance that code might be spread across multiple code hosts, especially at larger enterprises. As a developer tools company, our experience has proven this true.

For example, one of our customers, a rideshare company, uses four code hosts. Another customer, a cloud computing company, uses over 700 repositories. Many companies—correctly, we might add—empower developers to adopt software that increases their developer velocity. Overall, this permissiveness is a net good, but it does come at the cost of security scanning efforts.

The mobile team at a hypothetical organization, for instance, might have started using GitLab as part of a trial. But, because they're separate from the other development teams, that usage remained hidden. It'd be all too easy, then, for security engineers to run the security scanner on GitHub code and miss vulnerabilities in the GitLab code.

This problem isn't limited to code hosts. Manulife, a multinational insurance company and financial services provider, **found that** "developers continuously introduced languages, frameworks, and platforms." The word worth emphasizing here is "continuously." Coverage isn't a one-time problem solved by an audit. Developers will always be introducing new software, and security scanners will always be behind.

Manulife isn't alone. Twilio, a company that provides programmable communication tools, found the **continuous introduction of programming languages** to be both a "boon and a bane." Laxman Eppalagudem, Senior Product Security Engineer at Twilio, admits:

"When we started our investigation we really did not have an idea about the scope [of the programming languages in use]."

There's always the possibility, too, of an insider threat. Through no fault of their own, security engineers might be unaware of software because a malicious developer has installed it to create a backdoor. Security engineers won't know about this software because the malicious developer won't tell them, meaning that if an organization's security relies on a security scanner, and a security scanner relies on the knowledge of the security engineers, then a malicious developer can go undetected.

The scope of what security scanners have to scan is continuously growing. Security engineers will likely always be searching in the margins for undiscovered software.

Solving the above problem—if it can be solved at all—presumes that security scanners support all the discovered software. They do not.

Parallel to the fact that developers are continuously adopting software, developers, inside and outside the organization, are building new software that will eventually be adopted and require support from the security scanner vendor. Keeping up with the notoriously fast-paced software industry is a tall, if not impossible, order.

Let's narrow it down. If you look at JVM build tools alone, you have Maven, Gradle, Ant, sbt, Pants, Buck, Bazel, etc. Each of these build tools splinter into different versions. Each tool and version requires first-class support from a security scanner, or the scanner won't be able to find a vulnerability in the source code.

Twilio **noted this problem**, too, writing: "While we did have a central build system, we did enable our engineers to craft their builds how they see fit without any restrictions." The result? "This caused blockers for us in our investigation since we weren't entirely sure how some language builds were happening and required us to spend additional time to identify how builds are set up in Twilio."

One of our customers, an e-commerce startup, took particular issue with SAST scanners, saying, "Our problem with static code analysis is that no one's using the software that we do…we're not in Java World or C Land, where the packages have been pretty consistent for like, a decade. We're trying to use new technologies, modern technologies. We use packages as they come out."

The problem surpasses the sheer novelty of new software. Many security scanners just don't support everything that needs to be supported.

For instance, let's say you're scanning an artifact repository manager, such as Artifactory, and trying to find all artifacts that contain a vulnerable .class file. A typical security scan wouldn't give you confidence that you found all instances of Log4Shell because there could be projects that publish JARs to other Artifactory instances. There could be numerous ways developers deploy or release JARs that a security scanner won't catch.

Let's say, too, that you're looking for vulnerabilities triggered only when you call certain APIs in a given library (of which there are many). For those vulnerabilities, it's actually most helpful to browse the source code instead of scan it so that you can find and triage affected instances.

2. Security scanner results are noisy

Security scanners are producing even more information.

According to Veracode, in 2010, the median application was scanned less than once a month. By 2021, there was a 20x increase, and 90% of apps received scans more than once a week.

Veracode shows, too, that while the amount of scans has increased, so has the number of scan types. As we covered earlier, between 2018 and 2021, there was a 31% increase in the use of multiple scan types.

The problem with this density is the noise. The number of results can be overwhelming, and the rate of false positives can be misleading. According to Josh Bressers, VP of Security at Anchore, "the vast majority [of security scanners] will produce a 13,000 page report." It's then, he says, that you run into a **human-scale problem**.

Even if they team up with developers, security engineers will struggle to parse, prioritize, triage, and remediate vulnerabilities in a 13,000-page report. It's simply too much to handle. Bressers writes that the number of false positives can make it difficult for users to **figure out the next steps**.

The rate of false positives is not insignificant. A **Fastly study** shows, for example, that web application and API security tools (a broader category that includes security scanners) produced 53 alerts per day on average, and 45% of these alerts were false positives. And **Tromzo research** shows that about 25% of developers find 50-75% of vulnerability findings are false positives and almost one-fifth of developers find that more than 75% of vulnerability findings are false positives.

The term "false positive" actually covers a range of findings, including vulnerabilities that a scanner claims exist in your codebase that don't, vulnerabilities that a scanner finds in a feature you don't use, and vulnerabilities that a scanner finds but that you haven't deployed.

The latter, in particular, suffers from **what researchers have called** "over-inflation." In one study, researchers found 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 scanner might detect it, are effectively false positives because they are not actually deployed and do not actually present a threat.

One of our customers, an online retail company, candidly referred to this problem as a “pain in the ass.” According to them, their composition analysis scanner will “check your package manifest and say ‘You have a vulnerable package in your package list,’ but it doesn’t tell you if you’re using it in a vulnerable way. So it’s a huge time sink.”

The result is a lot of noise for very little signal.

3. Security scanners rarely produce actionable results

Security scanners, as the name suggests, scan for security flaws. However, not all security scanner vendors are in the business of offering fixes. As a result, security engineers and developers are often left with a bundle of problems and few solutions.

Composition analysis tools, such as Snyk, Black Duck, and WhiteSource, often mention which version you should upgrade a vulnerable dependency to. Vulnerability scanners, however, often suffer from context collapse because they will report a vulnerability without offering context on where in the codebase you can fix it.

In general, when security scanners suggest fixes, they often aren’t customized to the codebase they’re scanning. This lack of customization and specificity means that developers either have to edit or rewrite the fix if it can be used at all. SAST scanner results, particularly, tend not to be actionable in this manner because the recommendations are generic.

These fixes also tend to lack scalability. As covered in the previous section, security scanners can produce a lot of information. They rarely, however, match the scale of problem-identification with solution-suggestion. Aside from Snyk, which does offer the ability to **upgrade dependencies with automatic PRs**, most security scanners leave security engineers and developers to draft a multitude of PRs to fix the problems they’ve discovered.

This security scanner problem points to a larger, systemic problem, which we’ll cover in the next chapter: triaging.

Chapter 3:

Triage is the vulnerability management bottleneck

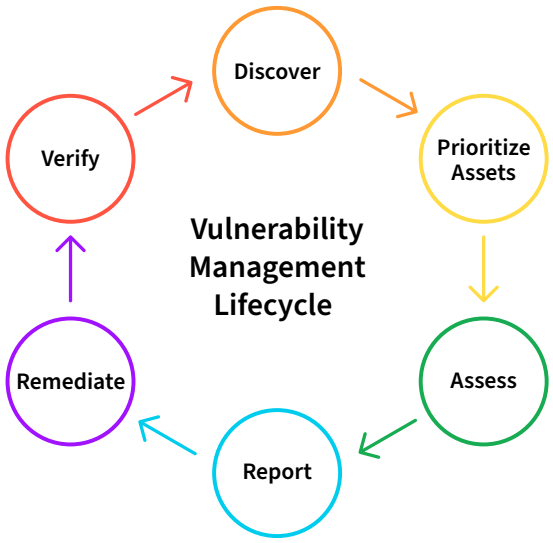
When you analyze a process, one of the most important components to identify is the process's bottleneck. Every process has one, and the effects of a bottleneck can vary from slowing work down to delaying or stopping it.

The security industry is at a point in maturity where the bottleneck is no longer identifying vulnerabilities but triaging them. Triage—the bottleneck in the vulnerability management lifecycle—defines the entire system, and it's there that we must focus on improvements.

The misleading vulnerability management lifecycle

The vulnerability management lifecycle describes how security teams discover vulnerabilities, prioritize assets, assess threat levels, report vulnerabilities, remediate vulnerabilities, and verify remediation.

The lifecycle, typically visualized as a circular flowchart, is continuous, meaning that security teams are supposed to be running the process repeatedly.



Here, we run into a "**the map is not the territory**" class of problem: Visualizations like the one above incidentally and erroneously depict each of the stages in the vulnerability management lifecycle as more or less equal. The Report and Verify stages, for instance, likely take much less time and effort than the Discover and Remediate stages.

But when security teams actually run the vulnerability management lifecycle, they often spend inordinate amounts of time in the seemingly unassuming Assess stage.

Triage: new technology creates new problems

Security teams spend a lot of time in the Assess stage of the vulnerability management lifecycle because the Assess stage subsumes a process that's likely deserving of its own stage (and does receive one, in **our own lifecycle**): Triage.

Triage is a term generalized from the medical profession, wherein it refers to the process of assigning degrees of urgency to patients such that medical staff can **establish a treatment order**. In other words, if a dozen patients all come to the same hospital at the same time due to a large pileup of cars on the highway, triage will determine who gets care first, and good triage will ensure patients who need help the most urgently will get that help first.

Triage was not always a part of the vulnerability management lifecycle, which is likely why it is rarely depicted. As we covered in Chapter 2, according to Veracode, organizations have tripled the number of apps they scan in the past ten years. In the same period, organizations have gone from two or three scans a year to **more than one scan every week**. And the results of these scans are often voluminous.

Though significantly less deadly, the vulnerability management lifecycle does have a problem analogous to hospitals: many "patients" are arriving at the same time and there's a need to prioritize care. Now, we have to triage.

The (currently) broken vulnerability workflow

If we step back from the abstract vulnerability management lifecycle diagram and look at a prototypical vulnerability management workflow, we can see the triage problem clearly.

One of our security engineers worked at a Fortune 500 oil and gas company and experienced a fairly typical workflow. The security team adopted a mature software composition security scanner and regularly ran security scans to find known vulnerabilities.

As a result of these composition scans, the security team (of which our engineer was a part) 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 sentiment bears out in the research. According to a Snyk study, "Security is perceived as an activity that slows down the business and overall software delivery." Snyk breaks out how well organizations have integrated their security operations. Even among the organizations with the highest levels of security integration, **33% of them still report** that "security is a major constraint on the ability to deliver software quickly."

This workflow isn't even the worst version. At this company, there was a security engineering team, albeit small, to make some effort toward triaging vulnerabilities. Many other organizations skip the assessing and triaging stages entirely and hand that 13,000-page vulnerability report directly to the development team, asking them to either fix all the vulnerabilities in it or justify the existence of those vulnerabilities.

We've run into this workflow numerous times with customers. One customer, a Fortune Global 500 company in the fire, HVAC, and security equipment industry, said:

> **"We have to go through and identify whether they're real issues or they're false positives and then we need to resubmit after we've resolved it. So it's on us to resolve it and then rerun the scan and make sure that we pass and clear all critical and highs before launch. If we do have any kind of medium or minor [vulnerability], we have to build a plan out and show that there will be a future release where we can get those resolved or they have to be resolved before initial launch."**

Triage becomes inevitable whether you have a security team handling vulnerabilities or not. Anchore VP of Security **Josh Bressers explains**: "If you ran the scan or you were handed a scan, one of the biggest jobs will be figuring out which results are false positives. I don't know of a way to do this that isn't 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?

Research indicates this process is likely valid. **One study** found that "the vast majority (81%) of vulnerable dependencies may be fixed by simply updating to a new version, while 1% of the vulnerable dependencies in our sample are halted, and therefore, potentially require a costly mitigation strategy." In other words, if security teams aren't pausing development for false positives, they're pausing development for fixes that require simple updates.

Those familiar with security scanners likely have an objection: Most scanners assign severity scores to their results. **Bressers has a response**: "Don't trust the severity the scanner gives you…They have no idea how you're using a particular piece of code or dependency. Their severity ratings should be treated with extreme suspicion. They could be an easy way for a first pass ranking, but those ratings shouldn't be used for anything after the first pass."

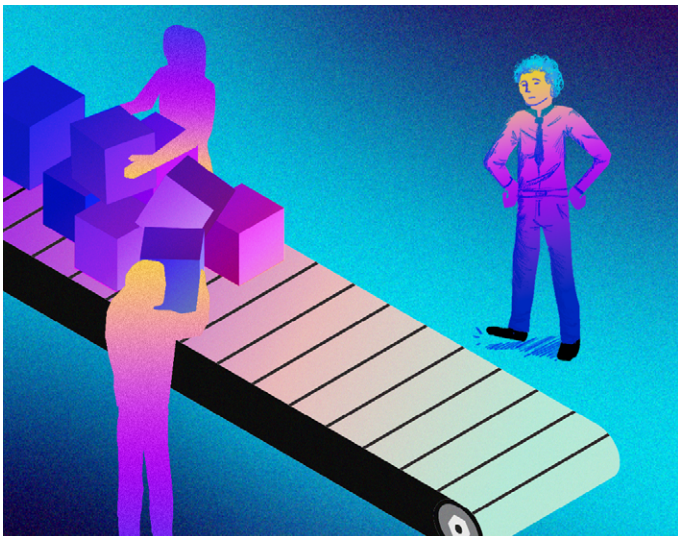
However, the argument here isn't that this workflow is a result of incompetence or malice—it's that we haven't yet given triage the respect it deserves in the vulnerability management lifecycle.

Triage: the hidden bottleneck

Abstractions are only as useful as the information they provide. As we covered at the start of this chapter, the typical visualization for the vulnerability management lifecycle proposes a circular process with more or less equal stages.

To learn more, let's shift the visualization to a conveyor belt. The conveyor belt metaphor is hardly a novel image, but it is useful. The book "The Phoenix Project," one of the most popular introductions to the world of DevOps, uses it handily. The book owes much, and the authors make this debt explicit, to the work of Eliyahu M. Goldratt, particularly his book "The Goal," and even more particularly to what he called the "theory of constraints." We, too, will owe Goldratt for his analysis.

Visualizing workflows as conveyor belt processes is useful primarily because you can discover those processes' constraints, or bottlenecks, more easily. You can easily imagine factory workers assembling a car, piece by piece, and see where the conveyor belt has to pause. Then, you can see how work that's done before the bottleneck piles up at the bottleneck and how workers after the bottleneck are stuck waiting for work. This is the theory of constraints in action.



As the authors of “The Phoenix Project” write, “Any improvement made after the bottleneck is useless, because it will always remain starved, waiting for work from the bottleneck. And any improvements made before the bottleneck merely results in more inventory piling up at the bottleneck.”

Of course, most processes don’t look like conveyor belts—at least at first glance. In “The Phoenix Project,” the authors argue you can apply the theory of constraints to software development and IT operations and analogize the conveyor belt image to diagnose those bottlenecks. We can do the same for vulnerability management.

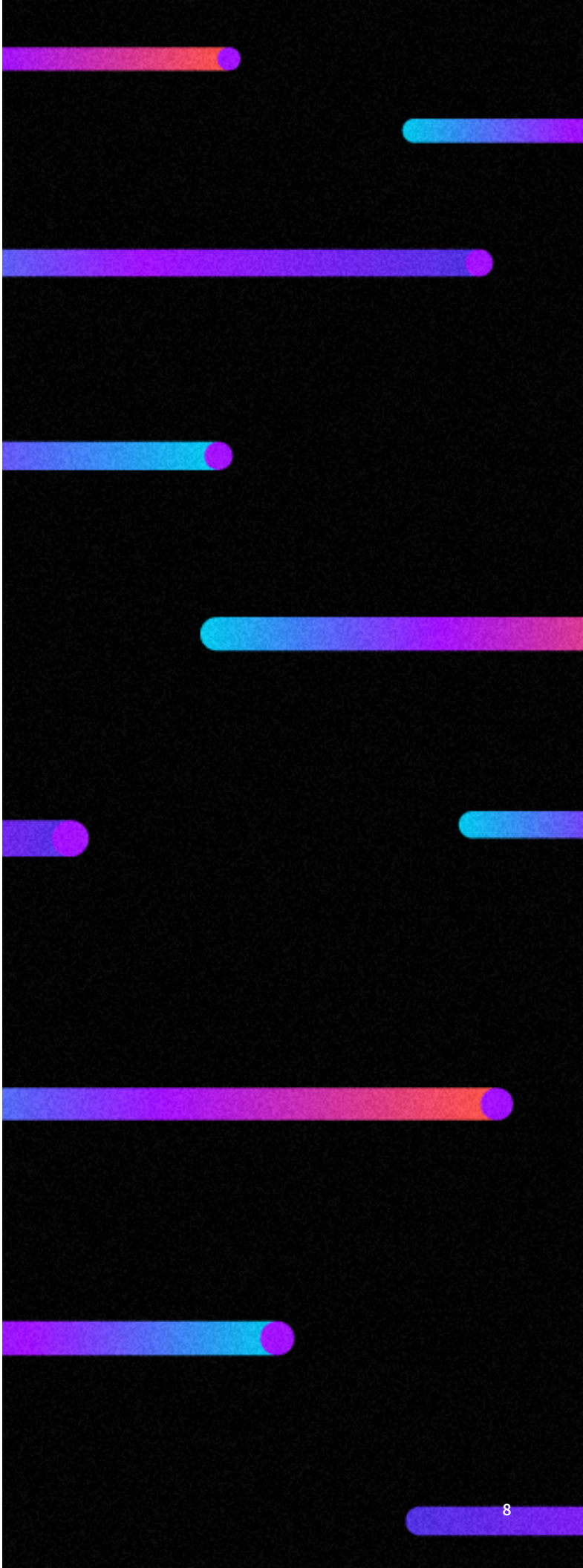
Security scanners, as Bressers claims, often produce reports of vulnerabilities that number in the tens of thousands of pages. These results, then, are like a pileup of automobile parts sliding along the conveyor belt all at once. The question is where the parts have to stop along that conveyor belt and where workers have to wait after that stopping point.

The answer, as you’ve likely guessed, is triage.

The previous section covered how the prototypical workflow involves a security team handing a development team a long list of scanner results. The business then has to wait on the next commit as the development team sorts through these results and either remediates or justifies them. Triage is the bottleneck.

Identifying the bottleneck is useful from a problem-solving standpoint, but it’s also valuable from a systems standpoint. “The Phoenix Project” authors write that the bottleneck “dictates the output of the entire system.”

Right now, the security industry spends a lot of effort on “shifting left” and evolving from DevOps to DevSecOps. These efforts aren’t wasteful, but they largely take place after the bottleneck when it comes to vulnerability management. If the bottleneck isn’t removed, the system’s output will remain more or less the same as scanners produce overwhelming amounts of results and increasingly agile, DevSecOps-informed development teams look for actionable, valuable work.



Chapter 4:

True comprehensiveness and the 99% to 100% chasm

The real work begins once security engineers take action on the security scanner results. Security scanners run periodically, creating a stream of results with unpredictable highs and lows. As new results come in, security engineers and developers work to triage those results and remediate the most urgent to the least urgent.

The goal is not to find and fix every vulnerability. As Andrew Magnusson, author of "Practical Vulnerability Management: A Strategic Approach to Managing Cyber Risk," writes, vulnerability management "takes a more pragmatic approach."

Continuing, he writes, "Instead of asking, 'How can we apply all of these patches?' vulnerability management asks, 'Given our limited resources, how can we best improve our security posture by addressing the most important vulnerabilities?'"

In other words, vulnerability management takes a risk-management approach, not a perfectionist approach. This approach, however, applies to identifying and remediating the totality of vulnerabilities. This approach does not apply to finding and fixing individual vulnerabilities. When you actually identify a vulnerability, and your triaging identifies it as critical, you need to find every instance of that vulnerability. You need 100% coverage.

At first glance, 0% to 50% to 99% to 100% vulnerability coverage looks like a linear progression, as though coverage were a dial you could turn up and up. However, the size of the gaps between each level of coverage varies. Ubiquitous vulnerabilities like Log4j can emerge in obvious and obscure places, meaning it's easy to find some instances, hard to find most, and without the right tools, impossible to find all with confidence.

The gap between 99% coverage and 100% coverage is more like a chasm than a crack, and crossing it requires a new approach.

Vulnerability management and the nature of failure

Software development is a notoriously failure-prone industry. Unlike, say, bridge-building, where the failure state (collapse) is obvious, the many failure states latent in an application are **hard to define ahead of time**.

The result, which isn't necessarily bad, is software teams building as best as possible and fixing problems as they emerge. Bridge-builders can't add beams post-collapse, but software developers can patch a release.

If software developers and bridge builders lie on two ends of a spectrum, security engineers lie somewhere in between. Ideally, security teams know exactly what potential vulnerabilities lie on the horizon and can prepare for them. In reality, they don't.

But unlike software developers, the failure state isn't a broken feature and a temporarily unhappy customer. The consequences can involve:

- A data breach
- A loss of money or personally identifiable information
- A business-wide hit to the organization's reputation

No organization has collapsed because of a bug, but many organizations have struggled because of unforeseen vulnerabilities and resulting exploitation.

The pressure is high. And when it comes to vulnerabilities, it means that security teams can't just find some or most of a vulnerability; they have to find every single instance and be able to confidently assert so.

Take Log4j and Log4Shell, for example. Leaving even one instance of Log4Shell in your codebase opens your organization up to attack. If you're attacked, finding 99% of a vulnerability isn't substantially different from finding 0%.

Parallel to the challenge of finding every instance is the challenge of confidently finding no instances. You have to be sure that when your search turns up zero results, there's no instance hiding in some nook or cranny of your codebase. If you don't, you could communicate safety to your customers, suffer a breach, and have to apologize to your customers, as well as regulators, for allowing the breach and for wrongly assuring them they needn't worry.

The 99% to 100% chasm, then, not only needs an approach that closes the gap but one that closes the gap and ensures the gap will remain closed.

Security scanners are only part of the solution

Security scanners approach the possibility of vulnerabilities from a largely automation-based perspective. After some tuning, security scanners run automatically and mostly passively in the background. Some vendors even allow you to automatically create and deploy PRs to address the problems their scanners identify.

The fundamental problem is that you can't automate 100% of vulnerability discovery, identification, and remediation.

As we covered in Chapter 2, it's unlikely that even the most sophisticated security scanner will account for every code host, build tool, and programming language. Meaning that it's doubtful a security scanner will be able to scan every codebase comprehensively. Even assuming a scanner was comprehensive, novel vulnerabilities will continue to crop up.

Software development is also inherently error-prone, and bugs, some of which will become vulnerabilities, are inevitable. These vulnerabilities can lie in wait for years and get deployed ubiquitously, as in the case of Log4j, before being discovered.

Log4j isn't unique. For example, another major vulnerability, Spring4Shell, was **discovered** just three months after Log4j. Spring4Shell was leaked before a CVE came out, officially making it a zero-day, so, as with Log4j, security scanners—and their users—were stuck waiting for a CVE.

Don't think either that this problem is localized to the Java ecosystem. Lodash, an NPM package with over 40 million weekly downloads, had a **series of high-risk vulnerabilities** in 2020. And once again, these didn't get a CVE, which made it harder for security engineers using tools dependent on CVEs to catch them.

The goal of pointing out these vulnerabilities is not to criticize security scanner vendors but instead to make the case that, if you want comprehensiveness, security scanners are only part of the solution.

If security scanners approach code security from an automation perspective and handle vulnerability discovery passively, then a full suite of security tools requires a complement, a tool that approaches code security from a human-actor perspective and enables users to handle vulnerability proactively. Our argument isn't that security scanners are bad but that security tooling needs a fuller approach.

In defense of reactivity: The three components of comprehensiveness

"Being reactive" has a bad connotation, but some amount of reactivity is optimal, even in a security process.

Just like development teams have to make compromises for the sake of feature development and take on technical debt, security teams need to know they can't catch every vulnerability before it enters the codebase. It's ultimately better to catch as many vulnerabilities as possible by using security scanners, keep shipping, and react quickly and comprehensively when a novel vulnerability is discovered, using, as we'll argue, Sourcegraph.

A truly comprehensive solution breaks down into three tactics: the systemic, preparing; the complex, scanning; and the simple, searching. All three tactics are necessary, and all are better together.

1. The systemic - preparing:

Before you start scanning or searching, you should prepare. Vulnerabilities shouldn't surprise you, and you should have procedures and vulnerability response plans in place before anything happens. A software bill of materials, or SBOM, is an excellent example of preparation in action. For instance, email delivery service Mailgun was able to determine within two hours that Log4Shell didn't affect any of its public assets because **it had an SBOM**.

2. The complex - scanning:

As we covered in Chapter 2, security scanners are necessarily complex. This complexity is both a benefit and a drawback. It's a benefit because that complexity is crucial for covering everything that needs to be covered, and it's a drawback because that complexity creates the conditions for breakdown. Nonetheless, good scanning is an essential part of an effective approach to vulnerability management.

3. The simple - searching:

Many of our customers, even those who weren't using Sourcegraph primarily for vulnerability management, found that code search was the best tool in their arsenal for finding and remediating Log4j. In the wake of Log4j, many customers found that a simple search for the text string 'log4j' was more useful than the delayed results of complex security scanners. From this simple but incisive starting point, our customers were able to add narrower and subtler searches that could pick out every instance of Log4j—some of which you can see in a **blog post** we published soon after Log4j was discovered.

The systemic keeps you prepared, the complex keeps you aware, and the simple keeps you proactive. All three are necessary. You need volume vulnerability management, which security scanners can provide, and precise vulnerability management, which Sourcegraph can provide, layered on top of a prepared security pose.

While the security scanner runs in the background, Sourcegraph can enable you to triage vulnerabilities, scope fixes, and focus your remediation efforts.

Log4j highlights the chasm between 99% coverage and 100% coverage. The success of our customers in dealing with Log4j proves code search is the best way to cross the chasm.

Chapter 5:

Start and end with search

Code search, one part of the Sourcegraph code intelligence platform, is the best way to start finding and fixing vulnerabilities and the best way to close the loop and prove those vulnerabilities are fixed.

Security scanners are also essential, but code search is the missing piece. Code search enables security engineers and developers alike to triage vulnerabilities, find vulnerabilities wherever they may exist in even the most complicated codebases, and prove to business leaders and customers that fixes are deployed and vulnerabilities are remediated.

Sourcegraph customers can immediately triage vulnerabilities

As we identified in Chapter 3, triage is the bottleneck that defines the vulnerability management system. As vulnerabilities pile up, developers and security engineers without code search struggle to triage vulnerabilities, scope the impact of particular vulnerabilities, and assign levels of urgency for remediation.

Let's start with Log4j. While other organizations had to wait for their security scanners to update, Sourcegraph customers leaped to action. They were able to:

1. Break down the vulnerability into parts.
2. Triage vulnerable versions, defining which were least and most severe and which were easiest and hardest to fix.
3. Find vulnerable versions wherever they exist in the codebase and deploy fixes.

The severity and range of impact for Log4j varied across versions. For example, with the Log4j 1.x vulnerability, codebases were only insecure if they used JMSAppender. Jon Kohler, Technical Director of Solution Engineering at Nutanix, used Sourcegraph Code Search to see where JMSAppender existed, fixed it, and sent out a release.

According to Kohler, "That took almost less than five minutes." With Sourcegraph, he could triage by identifying the scope of impact and predicting the effort it would take to fix it.

Log4j exposed another problem with traditional triaging: It's hard to know which versions of a potentially vulnerable dependency are actually vulnerable. Sourcegraph customers used code search to do both basic and advanced searches, including broad searches to find instances of Log4j and regular expressions to narrow results down to only vulnerable versions of Log4j. Customers were then able to figure out how vulnerable they were on a codebase-scale and how hard it would be to fix each vulnerable version.

The ability to narrow and filter results is useful beyond vulnerabilities like Log4j. For example, if you have a compromised vendor, you can use Sourcegraph Code Search to find the author, date, and the degree of usage of any vendor-specific features. You could easily determine whether you could do a simple migration off that vendor's product rather than wait for the vendor's fix.

Being able to narrow and filter is relevant for understanding dependencies, too. With code search, you can see whether you're using a dependency directly or indirectly, enabling you to decide whether you should change how you're calling it (if you're using it directly) or upgrade the dependency that pulls it in (if you're using it indirectly). You can also get more context for the dependency and see which of your developers you can go to for help either understanding or fixing it.

With structural search, you can even dial in fine-grained search criteria to search specifically for files that use a vulnerable package and are actually vulnerable. If only a specific method is vulnerable, you can use a structural search with regex to search for files that contain that package but do not contain that vulnerable method.

Sourcegraph Code Search becomes an essential tool in the vulnerability management lifecycle. Search makes a new level of triage possible, enabling companies to find vulnerabilities, scope impact, and determine what it will take to deploy fixes. With code search, the vulnerability management lifecycle can start spinning efficiently again, the triage bottleneck relived, and developers and security engineers can fix vulnerabilities effectively.

Code is the source of truth, and code search makes code accessible

Code is the ultimate source of truth in any organization. Documentation is essential but is always a step behind the codebase itself. Even the most well-documented codebase contains information that isn't present in the docs.

At Sourcegraph, for instance, we turned to the source of truth—our code—before we did anything else for Log4j. We determined immediately, via a search for 'log4j', that Log4j wasn't present in our codebase. To be more confident, we searched for the presence of any Java code since Log4j is a Java-based logging utility and found,

similarly, no Java code. A security scanner might have been able to do the former but wouldn't have been able to do the latter.

The ability to shift from search to search is useful when looking for the presence or absence of vulnerabilities. In Chapter 3, we described a workflow at an oil and gas company—though not unique to this company—where security engineers had to parachute into different parts of a codebase to find and fix vulnerabilities. With code search, security engineers can easily analyze parts of the codebase they aren't familiar with without spending unnecessary time trying to understand every bit of the codebase.

Security engineers can, for instance, use **Go to definition** to immediately find the definition of a symbol or use **Find references** to find all the references for your search results. Code search makes it easy to follow what's calling a function without having to read the whole file. Security engineers, and developers who might not be familiar with a given part of the codebase, can focus on only the parts of the codebase relevant to a given vulnerability.

As discussed in Chapter 4, the unfamiliarity problem hinders automatic security scanners. If security engineers don't know about all the code hosts, programming languages, and build tools in use, they can't ensure a code scanner will cover all code.

Sourcegraph Code Search has an advantage over many security scanners because it works across code hosts and repositories. As a developer tool, Sourcegraph is more useful the more developers use and integrate it. There's a natural incentive to connect code search to the entire codebase, making coverage easier to ensure.

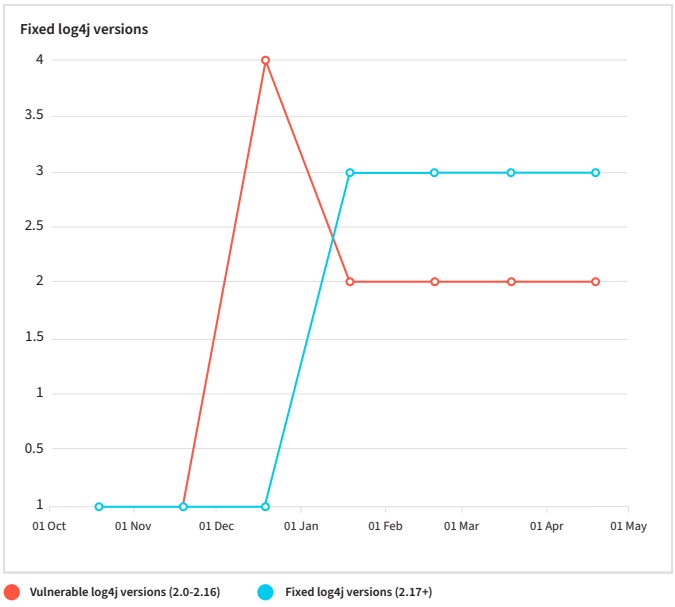
Close the loop and prove successful remediation

As we covered in Chapter 4, confidence is key to successful vulnerability management. You need to not only find and fix vulnerabilities but be able to confidently report to business leaders and customers that you have actually done so.

Confidence is the difference between being one of many vendors who said, "We believe, at this time, that we have found every instance of Log4j but will report back as progress develops," and being one of the few vendors who said, "We have found every instance of Log4j. This issue is resolved." It's in closing the loop that Sourcegraph proves itself to be an essential part of the security toolbelt.

With Sourcegraph's **Code Insights**, you can track the deployment of fixes and show business leaders intuitive graphs that visualize the progress of these fixes. As Kohler put it, "It's nice when you can just run a report and say, 'Here it is,' or 'Here it isn't.' It's much better than having to say, 'Well, boss, I think we got it all.'"

The below Code Insights graph shows the progress of Log4j fixes in an example codebase.



With **Code Search**, you can definitively prove that, after remediation, no further instances of a given vulnerability exist. "Confidence is everything," Codecov CEO Jeff Holland told us. "It's extremely important, the 100% confidence that you can go out in good faith to your customers and report the absence of a vulnerability."

With **code monitoring**, you can add a string, say, 'log4j', and get alerts whenever that string is added back into the codebase. Code monitoring ensures your short-term efforts have long-term effects.

Sourcegraph closes the vulnerability management loop, making it the best tool to start and end with. Sourcegraph doesn't replace security scanners, but it completes the solution of which security scanners are a part.

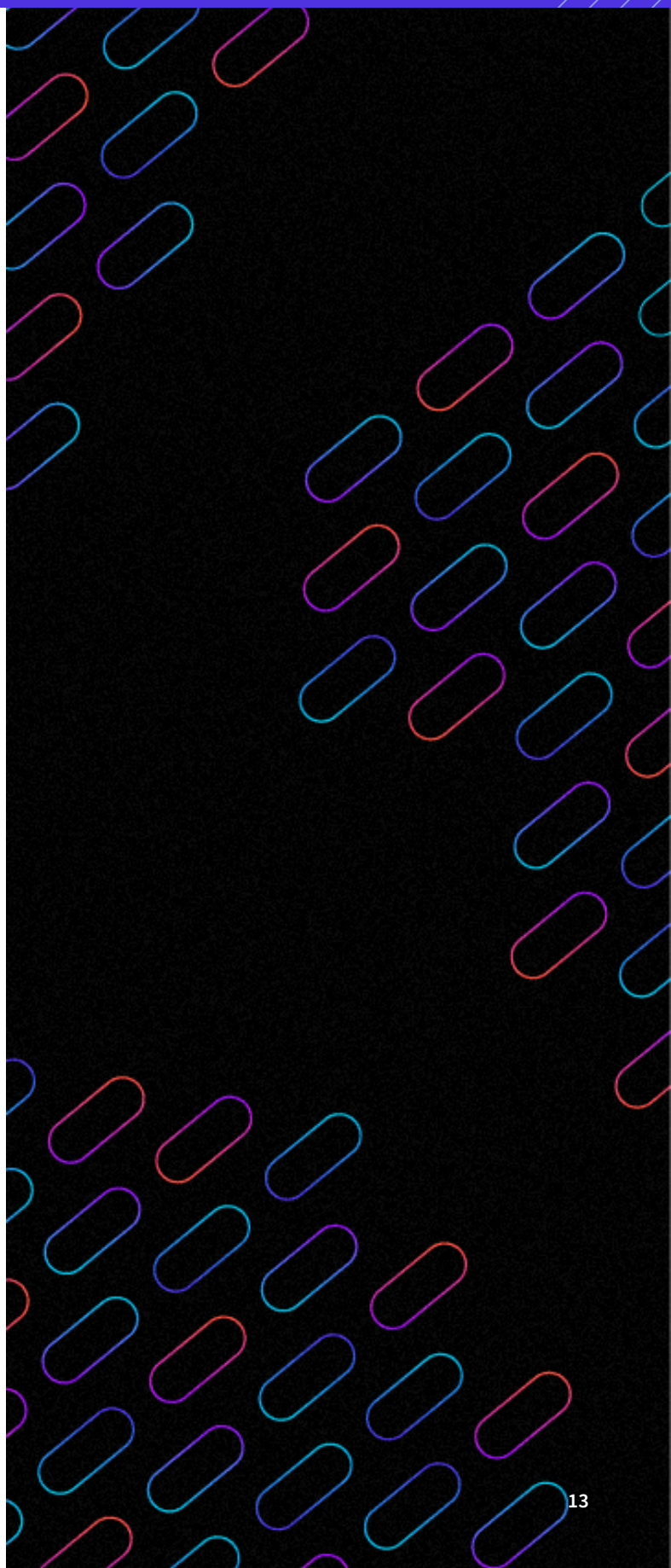
Log4j shows the necessity of simplicity

Complexity is attractive, and if we're not careful, we can pursue overly complex strategies that would benefit us less than simpler ones.

Complexity bias describes how we tend to "give undue credence to complex concepts." Studies show that we're more likely to choose the more complex option when given two competing hypotheses. As **Farnam Street puts it**, complexity bias leads us to "ignore simple solutions—thinking 'that will never work'—and instead favor complex ones."

Log4j revealed the limits of relying on complexity and the benefit of using a "less is more" approach. Ultimately, a simple search for 'log4j' proved more useful for many of our customers than the results of a complex security scanner. This is not a critique of security scanners but evidence that an effective vulnerability management lifecycle and resilient security posture require both simple and complex strategies, both precise and volume-based tactics, and both proactive and passive processes.

When a vulnerability emerges, Sourcegraph is the best place to start and end. Start with search to identify impact; continue with search to guide your fixes; and end with search to prove remediation and provide closure. Sourcegraph is the missing tool in your security toolbelt.



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.