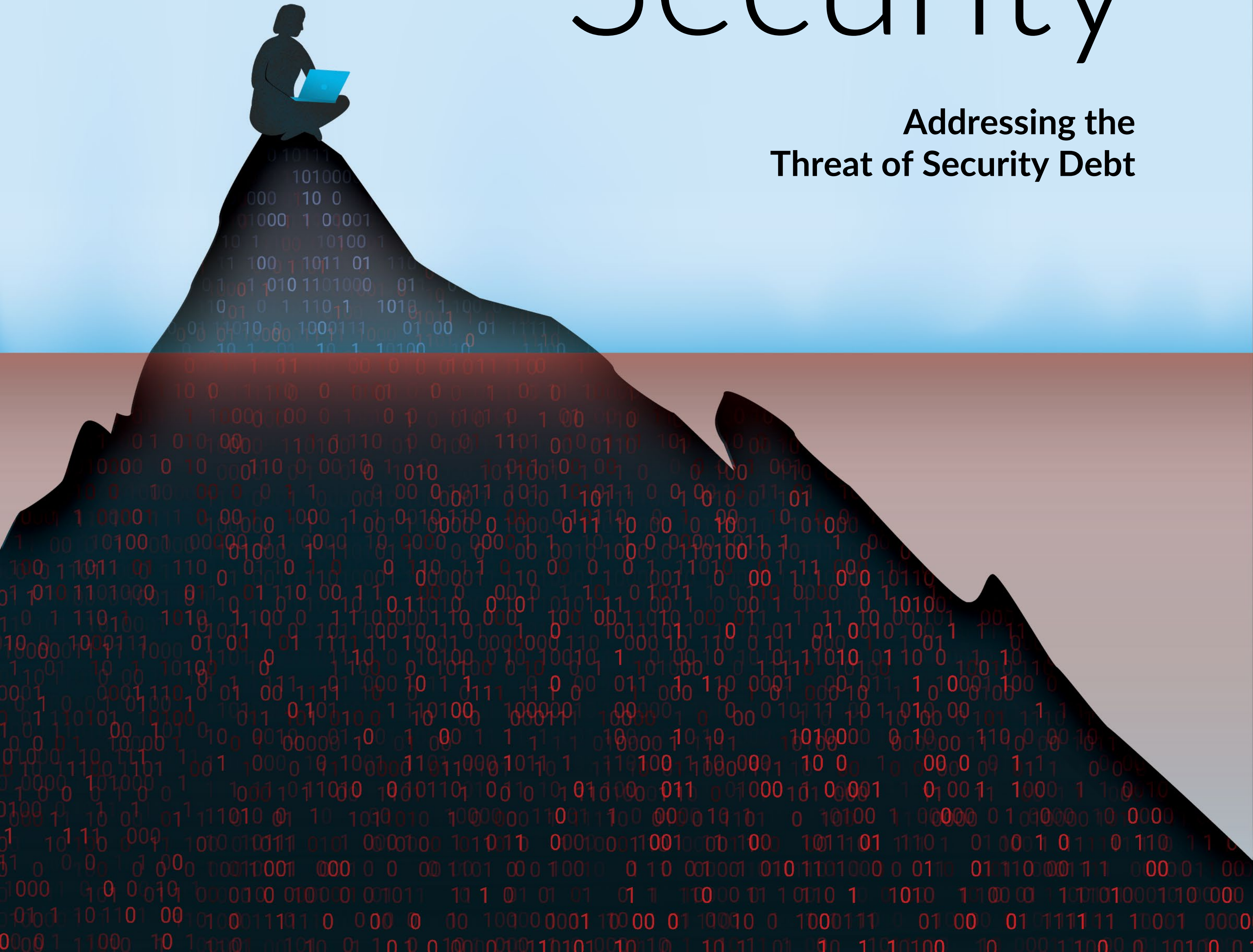
VERACODE

2024

State of Software Security

Addressing the Threat of Security Debt



The State of Software Security 2024

Addressing the Threat of Security Debt

Letter from the Editor

Artificial Intelligence (AI) wasn't born last year, but 2023 was its coming-of-age party. The proliferation of AI-generated code brings with it insecure code at scale and the likelihood of it becoming security debt.

Research indicates that code developed by AI contains about the same percentage of security flaws as that generated by humans. Other research suggests that programmers with a variety of experience levels fail to identify incorrect ChatGPT answers more than a third of the time.

While AI allows more code to be written more quickly, it does not deliver more secure code. The result is more risk introduced into your code base in the same amount of time.

The regulatory landscape has also evolved in the past year, with the US White House Executive Order on the Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence, the European Union's Cyber Resilience Act, and the US Security and Exchange Commission's Rules on Cybersecurity Risk Management, Strategy, Governance, and Incident Disclosure by Public Companies all coming into effect.

It's within this context that we explored Veracode's 18 years of data to answer questions about the accumulation of risk associated with insecure code. It's not news that applications contain security flaws, but we are excited to share insights on where, how, and why flaws persist over time.

In this year's report, our 14th, we do a deep dive into the distribution of security debt within applications, across industries and languages. We also continue the conversation that we began in last year's report regarding risks associated with how developers choose open-source libraries for their apps, with some surprising results.

Given the extent of security debt that we found, it is worth considering whether AI-assisted remediation tools may be helpful to pay down that debt, without the need to redirect your development teams or to increase their size.

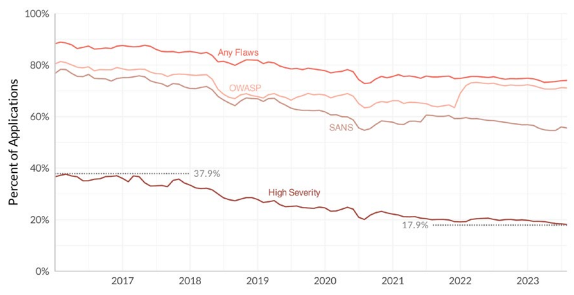
As always, we hope you enjoy the journey as much as we enjoy the discovery, analysis, and writing, and that you'll find in this year's report inspiration that improves the state of **your** software security and reduces risk to your organization.

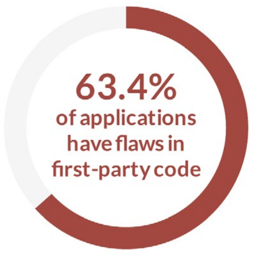
Chris Eng
Chief Research Officer

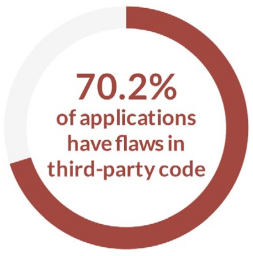
[illegible]

Contents

Key Findings

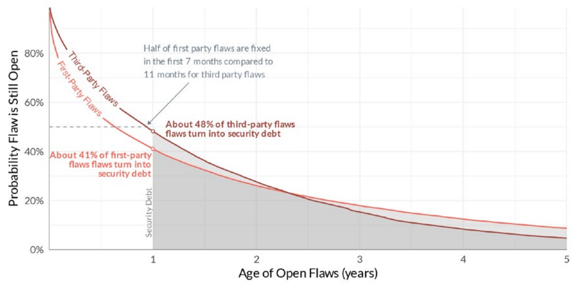


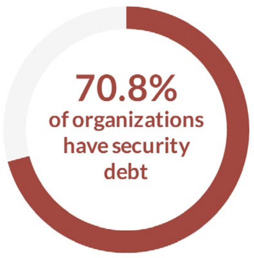


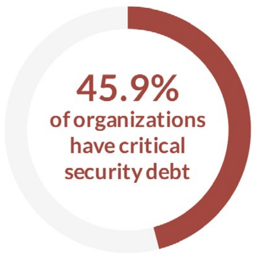


1. Good news up front: The prevalence of high-severity security flaws in applications has dropped to half of what it was back in 2016.

2. Roughly 63% of applications have flaws in first-party code and 70% contain flaws in third-party code. That's why testing both throughout the SDLC is so critical.

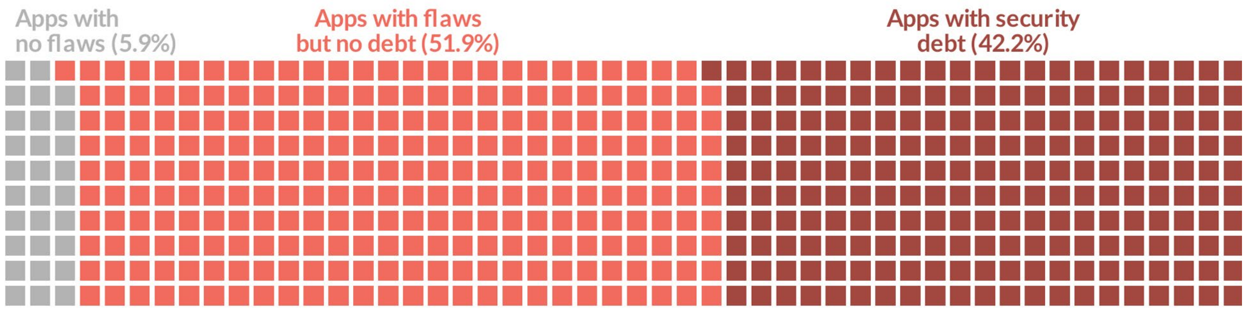






3. Third-party flaws take 50% longer to fix, with a half-life of 11 months vs. 7 months for flaws in first-party code.

4. More concerning still, almost half (46%) of organizations have persistent, high-severity flaws that constitute critical security debt.

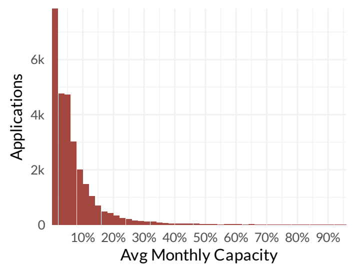


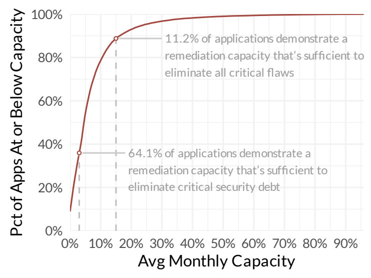
5. Flaws that stick around longer than a year become security debt. That occurs in 42% of applications and 71% of all organizations.

For the purposes of this report, we are defining security debt to mean all flaws that remain unremediated for over one year

	Manufacturing (12.0%)	Healthcare (14.8%)	Financial Services (15.4%)	Government (15.4%)	Retail & Hospitality (16.3%)	Technology (19.0%)
Larger (16.8%)	11.9%	15.3%	16.2%	6.6%	18.2%	20.4%
Midsize (16.5%)	13.2%	11.0%	8.6%	14.2%	9.2%	19.4%
Smaller (9.6%)	13.4%	3.7%	6.3%	19.8%	2.9%	10.4%

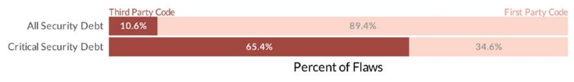
6 The prevalence of critical security debt across applications developed in large tech firms is 7x that of small firms in the retail and hospitality sector.



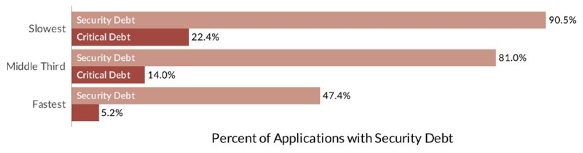


8 Only 64% of applications demonstrate a sustained capacity to eliminate all critical security debt. Prioritizing flaws for remediation is essential!

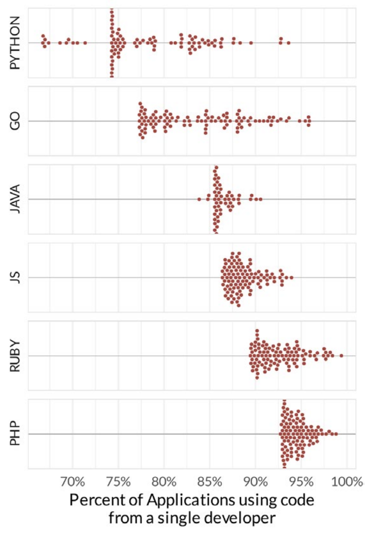
10 Developer education matters! Among organizations that use Security Labs, 37% have security debt. Compare that to 48% among application teams that do not. The time-to-fix difference is even more significant. Applications developed by teams that aren't using the Labs take seven months longer to reach that 37% mark.



7 Almost 90% of all security debt across all active applications exists in first-party code. But third-party code claims two-thirds of security debt rated as critical severity.



9 Continuous scanning must be accompanied by continuous remediation to be effective. Development teams that fix flaws the fastest are 4x less likely to let critical security debt materialize in their applications.



11 All languages have multiple open-source developers that contribute code that is included in 90% of applications. These represent potential weak links in the software supply chain.

State of Software Security at a Glance

We're thrilled to embark on another chapter in our ongoing quest to share data-driven insights on the current state of software security.

You'd think after nearly a decade and a half of doing this, we'd be scraping the bottom of the barrel in terms of learning new lessons and new ways to apply them. But the reality is there's so much to discover here that we must consciously restrain ourselves from filling 100+ pages with all the interesting and useful findings we uncover each year. And this year's no different.

Thank you for joining us on this journey of discovery. We'll start with a brief look at the lay of the land and then venture outward into the wilds of AppSec from there.

How common are security flaws?

Security flaws aren't ubiquitous, but they're far from rare. Based on the most recent SAST, DAST, and SCA scans using Veracode, unresolved flaws were detected in 80% of all active applications. If you're a longtime SOSS reader, the SAST-only version of the chart offers backward comparability and pegs flaw prevalence at 73% of all applications.

About 70% of applications have flaws included in the OWASP Top 10, an initiative to track the most critical risk to web applications. For the Common Weakness Enumeration (CWE) Top 25—another effort to track the most common and impactful security weakness—that statistic drops slightly to 41% of all applications. According to Veracode's rating, half of applications have flaws considered high (or very high) severity (though that drops to 19%, as seen through SAST scans alone).

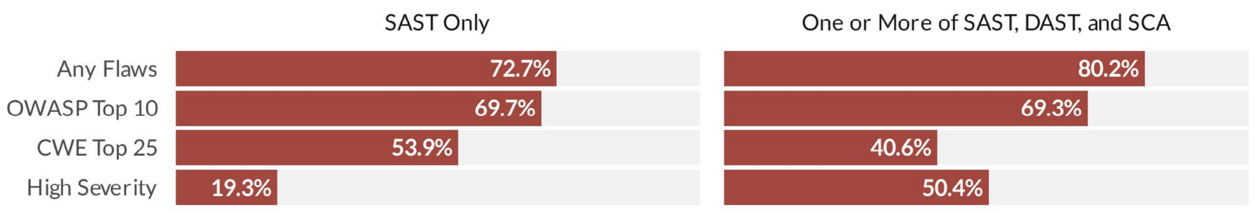


Figure 1: Percent of applications with security flaws detected in most recent scan

Why is the ratio of severe flaws lower in SAST?

The big difference here that we'll unpack throughout this report is that there are some major differences between first-party and third-party (largely open-source) code. SAST scans focus on the former, where developers are in full control and can prioritize severe flaws for faster remediation. Third-party libraries scanned via SCA, on the other hand, depend on other people to fix them before updates can be made.

When parsing these stats (and all those that follow), bear in mind that this report represents organizations that are proactively integrating tools like Veracode into their AppSec programs. Organizations without scanning integrated into their development processes will likely have a higher prevalence of security flaws than shown here.

Are flaws becoming less common?

The prior chart showed a snapshot of applications as of their latest scan, but those stats don't tell us whether flaws are becoming more or less common over time. Obviously, we'd like to see that trending down, which would indicate the overall state of software security is improving.

In general, the results do show a steady downward trend over the last eight years. We're particularly encouraged to see that the prevalence of high-severity flaws has dropped to half of what it was back in 2016.

Some will wonder about the uptick among OWASP Top 10 flaws at the end of 2021. This traces back to some changes to the Top 10 that year, resulting in the added flaws being detected in a larger number of applications.

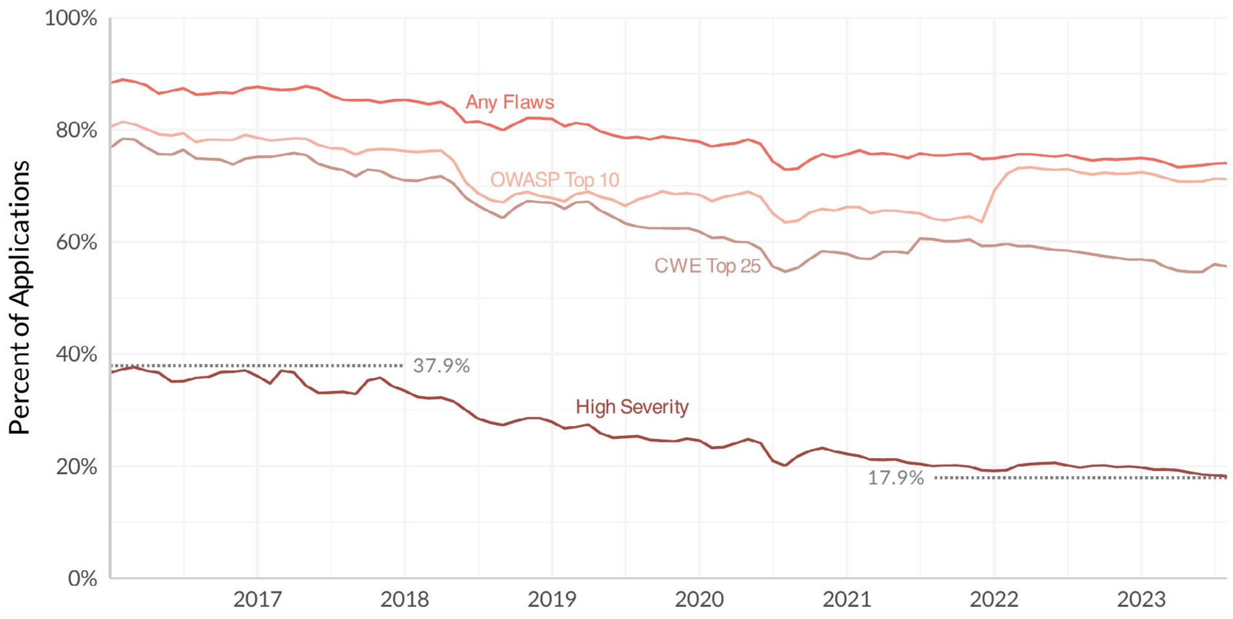


Figure 2: Trend of flaw detection rates over time

Keep in mind that this chart aggregates new and old, large and small applications across many different languages and types of organizations. The fact that there's a consensus trend in the direction we want to see is quite remarkable and points to the positive effects of sustained investments to improve AppSec.

How many flaws do applications have?

We know that the majority of applications have flaws, but are we talking more like ten or a ton? We use a metric called flaw density for this, because it normalizes for applications of different sizes. In short, flaw density tallies the number of flaws per MB of code identified in testing each application.

On average, a typical application has 42 flaws for every 1 MB of code. That seemed odd to us, so we asked Deep Thought to crunch the numbers. It took a while, but 42 was indeed verified to be the answer to life, the universe, and everything AppSec.[1]

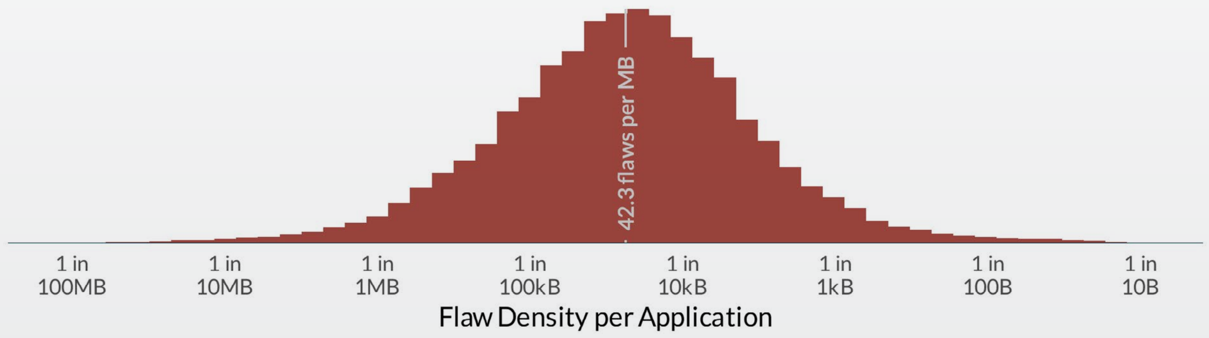


Figure 3: Density of flaws detected in applications

Of course, many applications don't conform to what's typical. We see some with densities as low as 1 flaw per 100 MB and others as high as 1 in 10 B. We'll explore some reasons why applications differ so much according to flaw density later, but we'll stop here for this "At a Glance" section.

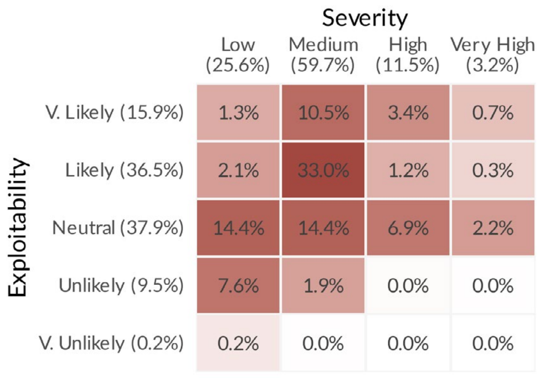
How critical are software security flaws?

There are two primary metrics for assessing whether flaws represent a critical security risk—severity and exploitability. Here's the gist:

1. **Severity** is the flaw's potential impact on confidentiality, integrity, and availability.
2. **Exploitability** is the likelihood or ease with which an attacker could exploit a flaw.

Overall, about 3% of all flaws are considered very high severity, and 16% are very likely to be exploited by attackers. Thankfully, less than 1% of all flaws earn the highest (worst) ratings for both criticality metrics.

If we broaden the threshold to include issues that are high severity and likely to be exploited (the upper-right 2x2 quadrant), we get 5.5% of all flaws. It goes without saying that these rare but risky security flaws warrant quick attention from developers, though it must be acknowledged that the glut of Medium-Likely flaws (43.5% of all flaws) represents a substantial attack surface for many organizations that must be managed as well.



Exploitability	Severity: Low (25.6%)	Medium (59.7%)	High (11.5%)	Very High (3.2%)
V. Likely (15.9%)	1.3%	10.5%	3.4%	0.7%
Likely (36.5%)	2.1%	33.0%	1.2%	0.3%
Neutral (37.9%)	14.4%	14.4%	6.9%	2.2%
Unlikely (9.5%)	7.6%	1.9%	0.0%	0.0%
V. Unlikely (0.2%)	0.2%	0.0%	0.0%	0.0%

Figure 4: Rating of flaw severity and exploitability (percent of all flaws)

[1] *There are also 42 charts in this report and the original draft had 42 pages. Alas, there were some changes during the Vogon review process.*

What types of flaws are most common?

In addition to ratings of severity and exploitability, identifying the types of flaws present in code provides useful insight into building threat models, assessing risk, and prioritizing remediation. There's no shortage of approaches for categorizing software flaws, but the CWE and OWASP Top 10 are two of the most popular.

The figures here plot CWE categories based on their frequency across and within applications. Those toward the right affect a lot of applications, and those near the top cluster in droves. Not many flaw types sit high on both the prevalence and intensity scales, but developers should take note of those that do. They're the most likely to affect your code.

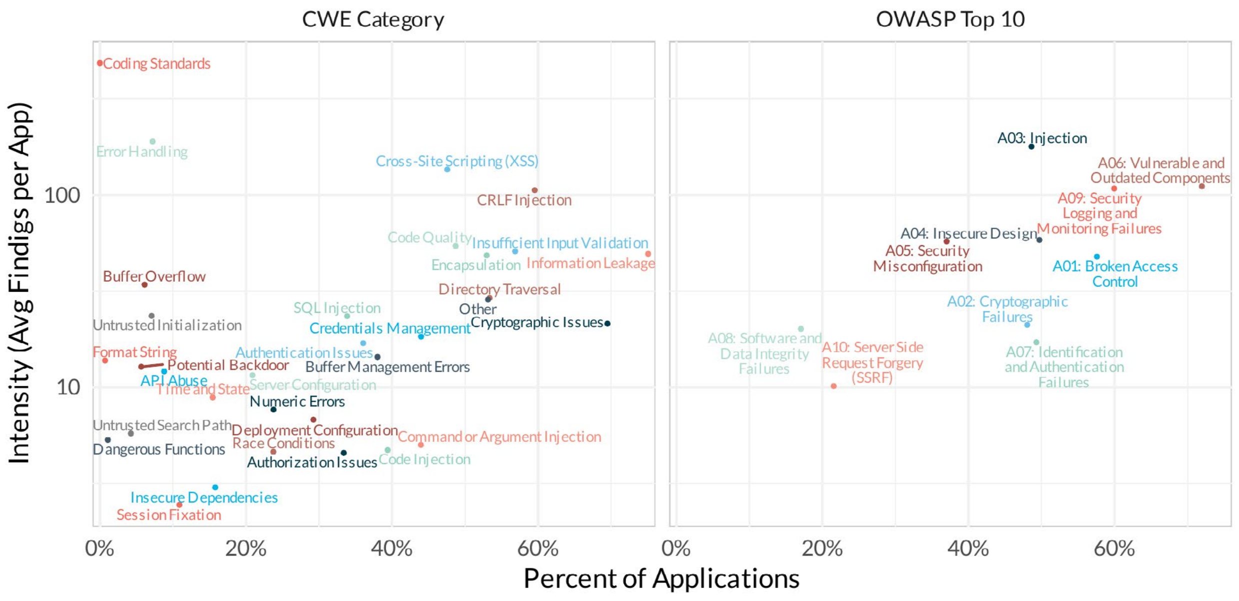


Figure 5: Prevalence and intensity of CWE and OWASP flaws in applications

We'll briefly point out Vulnerable and Outdated Components in the upper-right of the OWASP Top 10 plot. The inclusion of flaws found via SCA scans in these results is a big reason why this features more prominently than it has in the past. It's a reminder of a theme we'll keep coming back to in this report: **identifying and remediating flaws in third-party code requires a different approach.** The pervasiveness of vulnerable versions of the log4j library two years following the discovery of a zero-day exploit is a case in point.

Where are flaws most likely to be introduced?

The prevalence of flaws related to vulnerable third-party components necessitates a deeper look at the security of the software supply chain. We all know that flaws can be introduced in code that your team writes (first party) or via open-source software or other third-party libraries imported into applications. But which of these sources is more common?

Turns out it's actually pretty close. Roughly 63% of applications have flaws in first-party code, and 70% contain flaws in third-party code. That obviously means many apps have both, which is why including scans of both sources at various points in your secure development life cycle is so critical. We share more analysis specific to third-party libraries in the Securing the Software Supply Chain section.

First-party code: Code written directly by your development team for your applications.

Third-party code: Code imported into your applications via third-party libraries. Though not all third-party libraries are open source, we use that term interchangeably in this report.

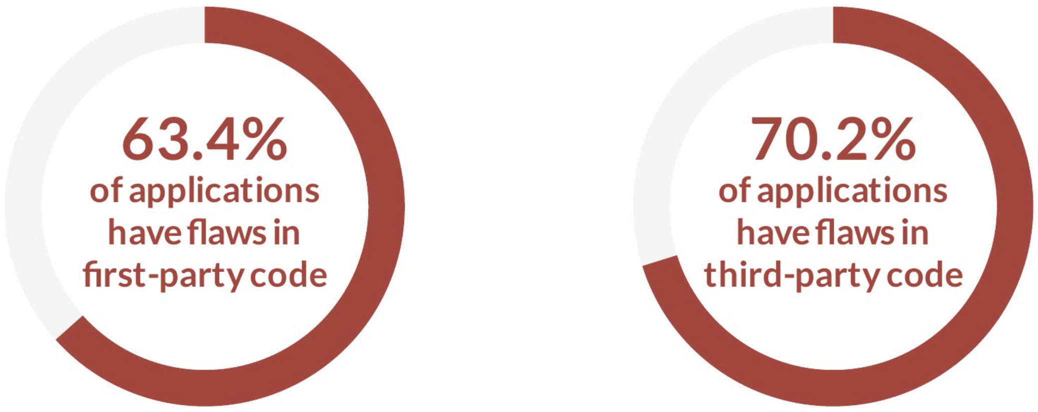


Figure 6: Prevalence of flaws in first-party (left) vs. third-party (right) code among applications

How quickly are flaws remediated?

Moving on from finding security flaws, let's talk about fixing them. How long do flaws stick around after being discovered? Overall, roughly one-third to one-fourth of all flaws are fixed in the first three months. The half-life for flaws across all applications (the time it takes to fix half of them) is about nine months.

Per the chart below, 41% of first-party and 48% of third-party flaws persist beyond the one year mark to become "security debt." The half-life of third-party flaws is also longer—11 months compared to 7 months for first-party flaws.

Security Debt: For the purposes of this report, we are defining all flaws that remain unremediated for over one year, regardless of severity, as security debt. In some cases of security debt, developers make a decision not to fix those flaws. In others, fixes are delayed for various reasons. We'll analyze remediation timelines more later in this report.

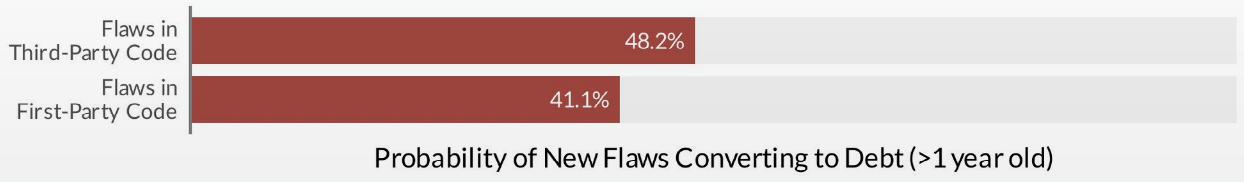


Figure 7: Probability of flaws converting to security debt in first-party vs. third-party code

Like its financial corollary, security debt tends to pile up over time and gets harder and harder for developers to pay off. But the good news is there are data-backed strategies for getting out of debt that we outline later in this report.

How common is software security debt?

In many ways, security debt reflects a culmination of the software security stats we've measured so far. It's a byproduct of how many flaws are present and how long it takes to fix them.

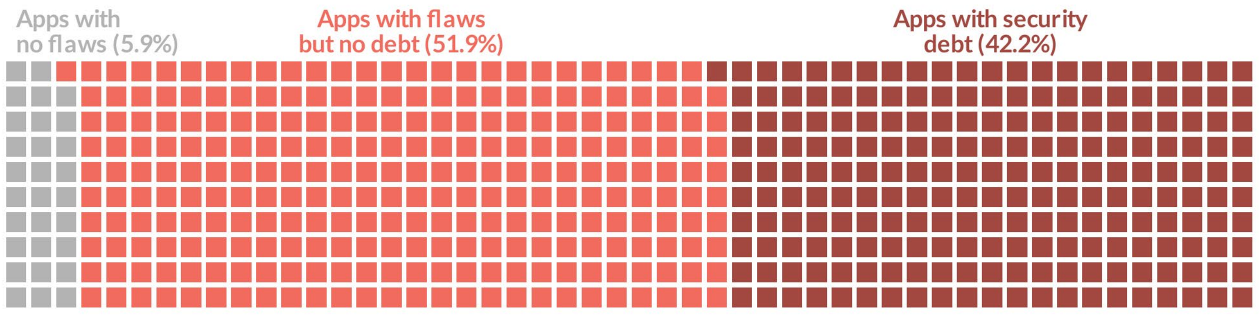


Figure 8: Prevalence of security debt across all applications greater than one year old

As shown here, 42%[2] of active applications (those that have been scanned for at least one year) have flaws that constitute security debt (and some of them carry a lot of debt).[3] That's an important finding that deserves much more than a passing glance. Let's dig deeper into exactly how much debt, where it comes from, and how to begin eliminating it in the next section.

[2] *We also verified this with Deep Thought to be sure.*
[3] *The proportion of apps with debt changes substantially depending on how we slice the data. Showing those different slices goes beyond the scope of this "at a Glance" section, so we added a special insert on the next page for those interested.*

How common is software security debt? (Revisited)

Before moving on to explore security debt more deeply, let's revisit the final question posed in the prior section. The answer greatly depends upon how we slice the data. We'll give some examples here.

If we don't filter out applications less than one year old, the waffle plot shown back in Figure 8 looks like this. There's a larger proportion of apps with no debt (because they're too young to have started accruing it), and the debt ratio drops from 42% to 23%.

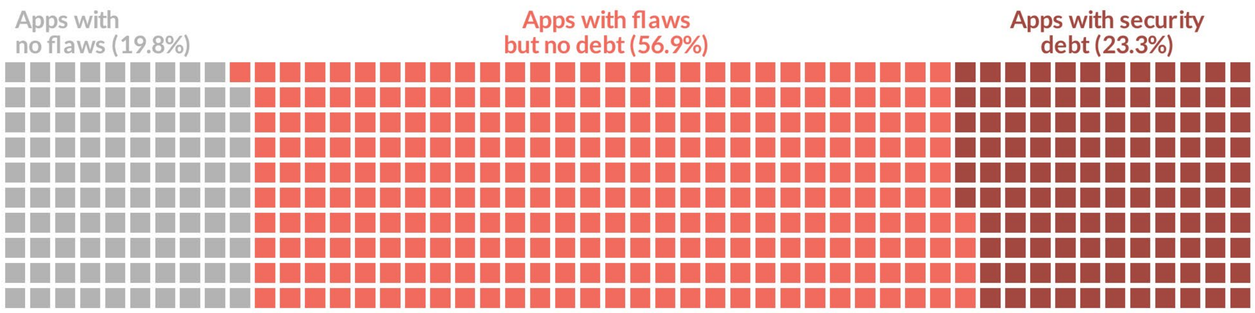


Figure 9: Prevalence of security debt across all applications (NOT filtered to one year old)

We also see substantial differences based on the types of scans conducted. We'll get into this later, but SCA scans of open-source libraries tend to show a lower proportion of security debt than SAST or DAST. The net result is that the combined view in the original Figure 8 has a lower overall debt rate.

To illustrate that effect, here's a version of this same chart based on SAST and DAST scans (SCA removed). This may be more in line with the expectations of many with a much larger percentage of applications saddled with security debt (69% rather than 42%).

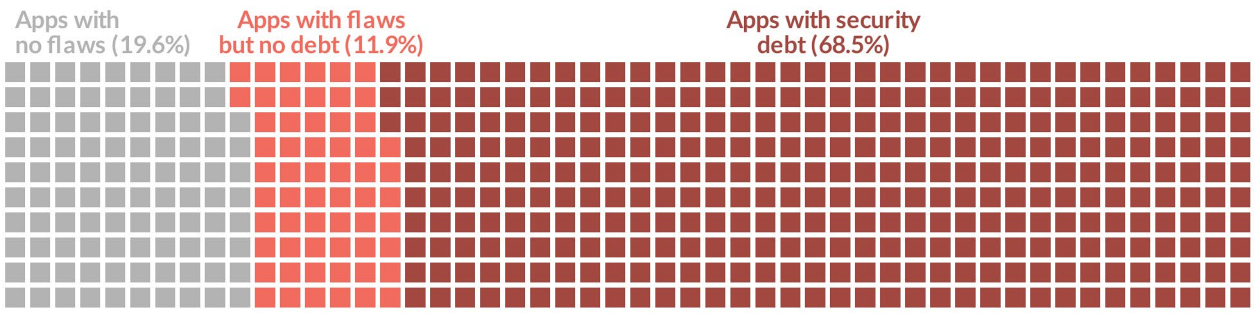
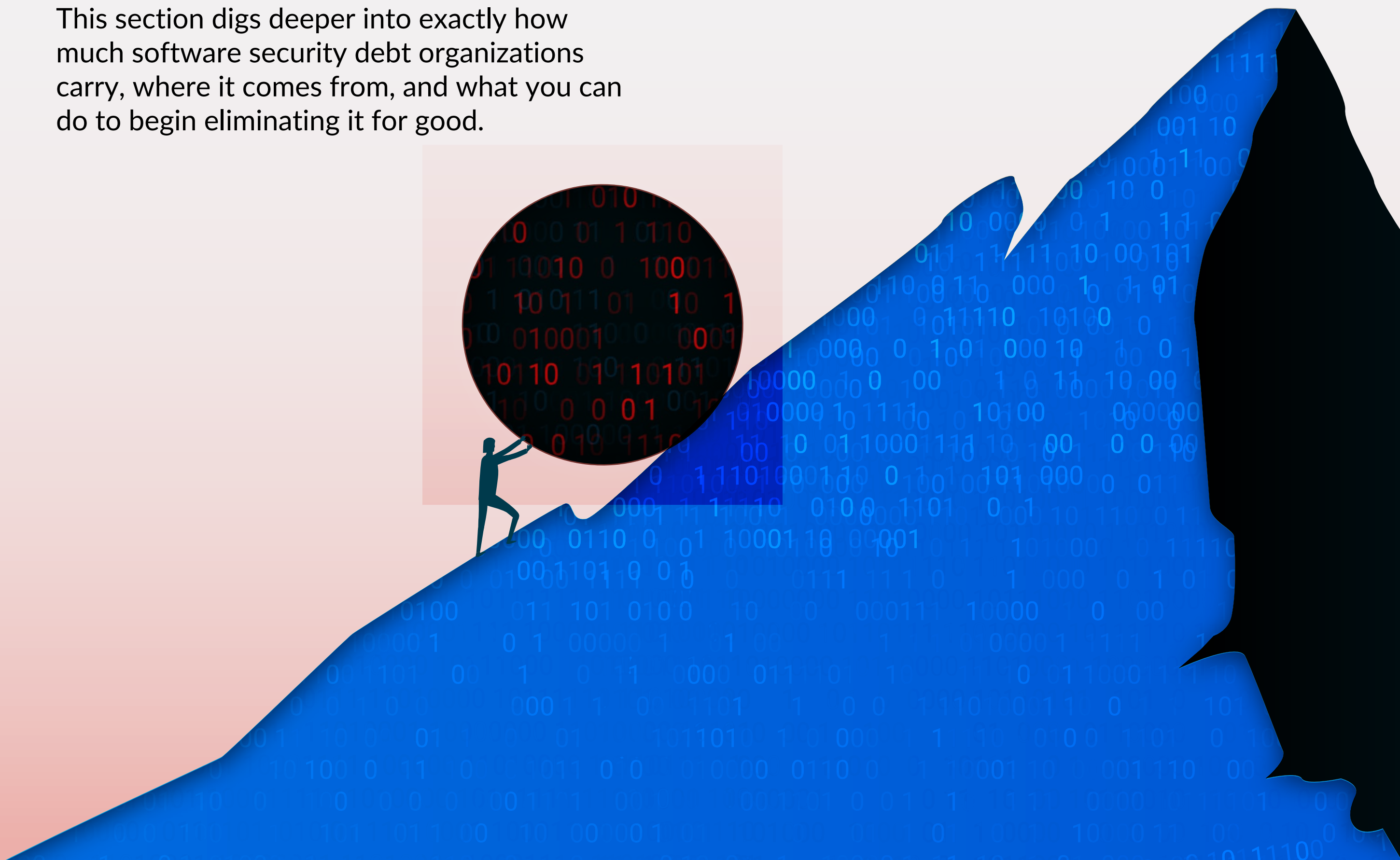


Figure 10: Prevalence of security debt across all applications scanned with SAST and DAST

We're not trying to undermine the findings or delve into the pedantic here. We simply want to arm readers with the insight that things will shift depending on what subset of the data (or type of scans) we're examining, which is why we'll try to make that clear in the analysis that follows.

Measuring and Managing Security Debt

We define security debt as any flaw that persists unremediated for longer than one year. We saw that 42% of all applications have flaws that exceed that threshold, which makes security debt far from a rare phenomenon. This section digs deeper into exactly how much software security debt organizations carry, where it comes from, and what you can do to begin eliminating it for good.



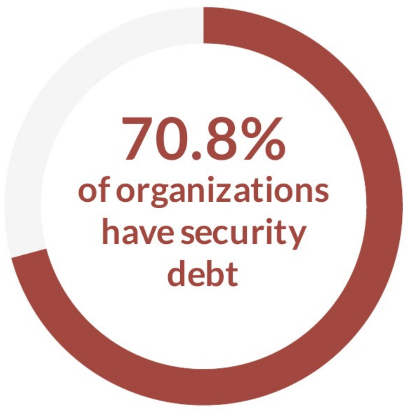
The prevalence of security debt

Security debt occurs within individual applications, but those applications are developed and managed by organizations. Thus, we begin by measuring the prevalence of security debt across all active applications for each organization in our dataset.

A large majority of organizations (71%) have security debt at some level. And close to half of all firms (46%) have high-severity persistent flaws that we'll classify as critical security debt. Based on this, we can conclude that software security debt is a major organization-level challenge.

Security Debt: All flaws that remain unremediated for over one year, regardless of severity.

Critical Security Debt: High-severity flaws that remain unremediated for over one year.



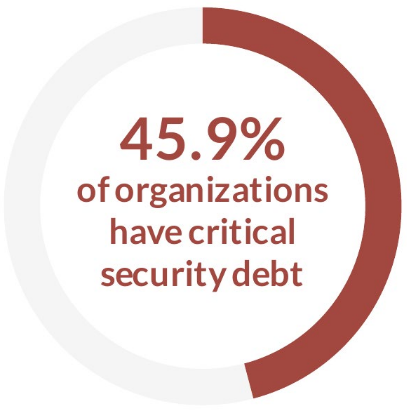


Figure 11: Prevalence of security debt (left) and critical security debt (right) among organizations

The prior charts do not address the extent of debt within these organizations, so let's go there next. To do this, we have removed organizations with very few applications to avoid misleading conclusions based on tiny scopes (e.g., debt in 1/1 apps would be 100% debt ratio).

Within this sample, 10% of organizations show no security debt.[4] Among those that do, as shown in Figure 12, the typical (median) firm has security debt in a third of its active applications. Security debt affects at least two of every three applications in a quarter of organizations. And just north of 1 in 10 firms has security debt in at least 90% of its applications.

[4] *This differs from Figure 11, the preceding donut chart (29% with no debt), because we've removed firms with less than seven applications.*

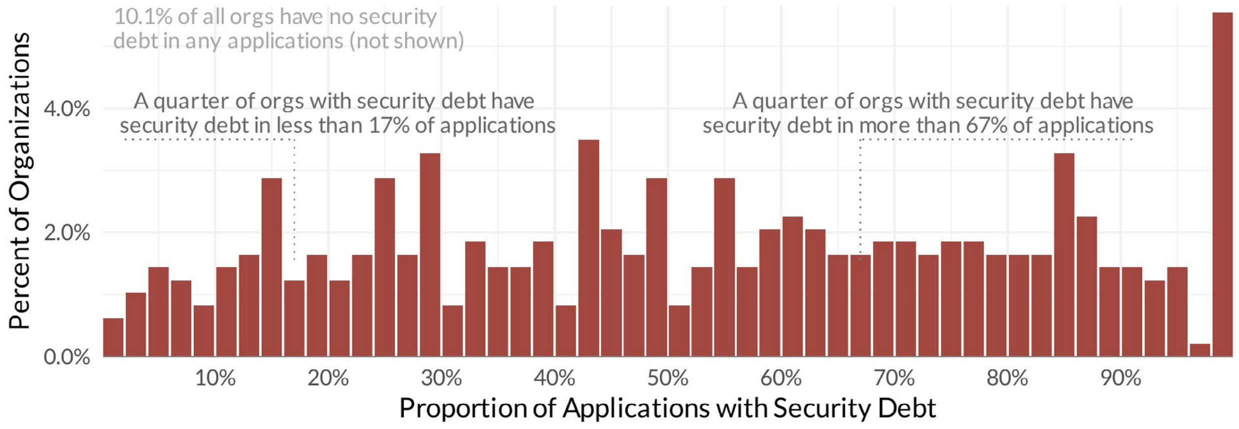


Figure 12: Prevalence of security debt across applications >1yr old within organizations

We create a version of this chart that isolates critical security debt in Figure 13. Over a quarter (27%) of organizations in this sample show no signs of persistent high-severity flaws as of their latest scans. Roughly half of them have critical debt in less than 16% of their applications. About one in ten firms exceed a critical debt ratio of 49%.

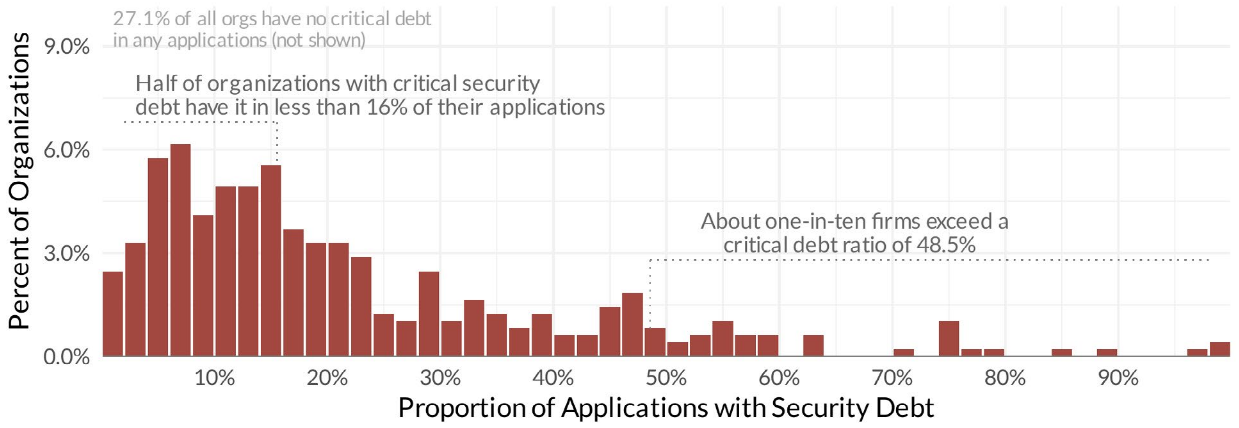


Figure 13: Prevalence of critical security debt across applications >1yr old within organizations

Continuing our task of measuring the prevalence of security debt across all organizations, we thought it might be interesting to look at debt ratio as a ratio of all flaws. Figure 14 takes the form of a beeswarm chart, with each "bee" (dot) representing an organization. Those firms are grouped according to the percentage of flaws across all their applications that constitute security debt (top) and critical security debt (bottom).

This security debt ratio is spread fairly evenly across organizations but "swarms" a bit more toward the higher end of the scale. From this, we can discern that just under half of all flaws (47%) for the majority of organizations can be considered security debt. In our experience, many dev teams let old issues slide in order to better focus on fixing new issues that arise as code is written.

At the same time, many organizations manage to achieve a much lower security debt ratio—especially for critical debt. The ratio of critical debt generally ranges in the low single-digit percentages among all flaws for most organizations. That, at least, is something we can all "bee" happy about.

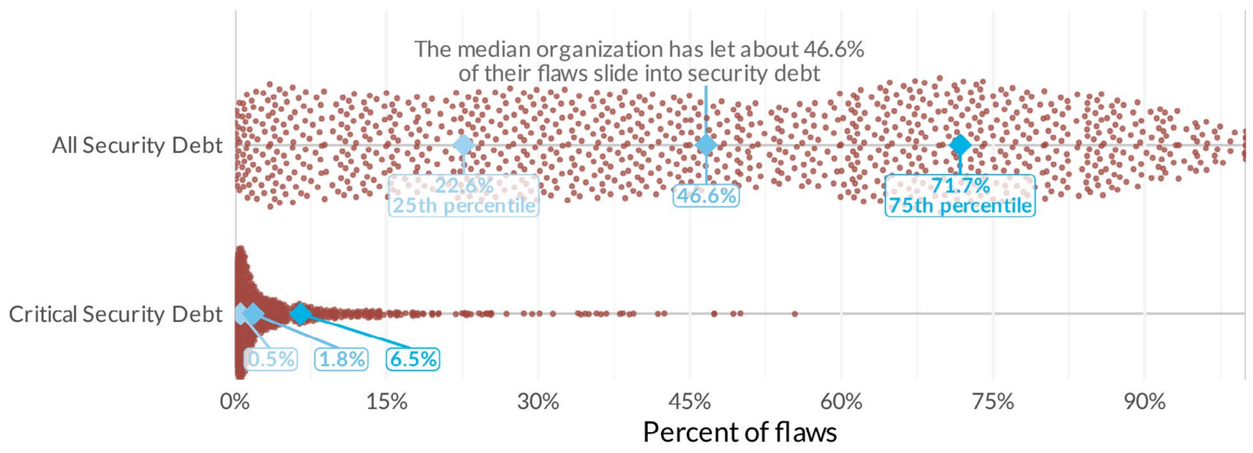


Figure 14: Percent of flaws per organization that qualify as security debt

To close out this section and help visualize the prevalence of security debt in organizations, we offer Figure 15. Each rectangle represents an organization sampled from our dataset, which is subdivided into sections corresponding to the size of its active applications. The color applied to those applications measures their density of security debt. You can think of this as a depiction of the application attack surface of each organization.

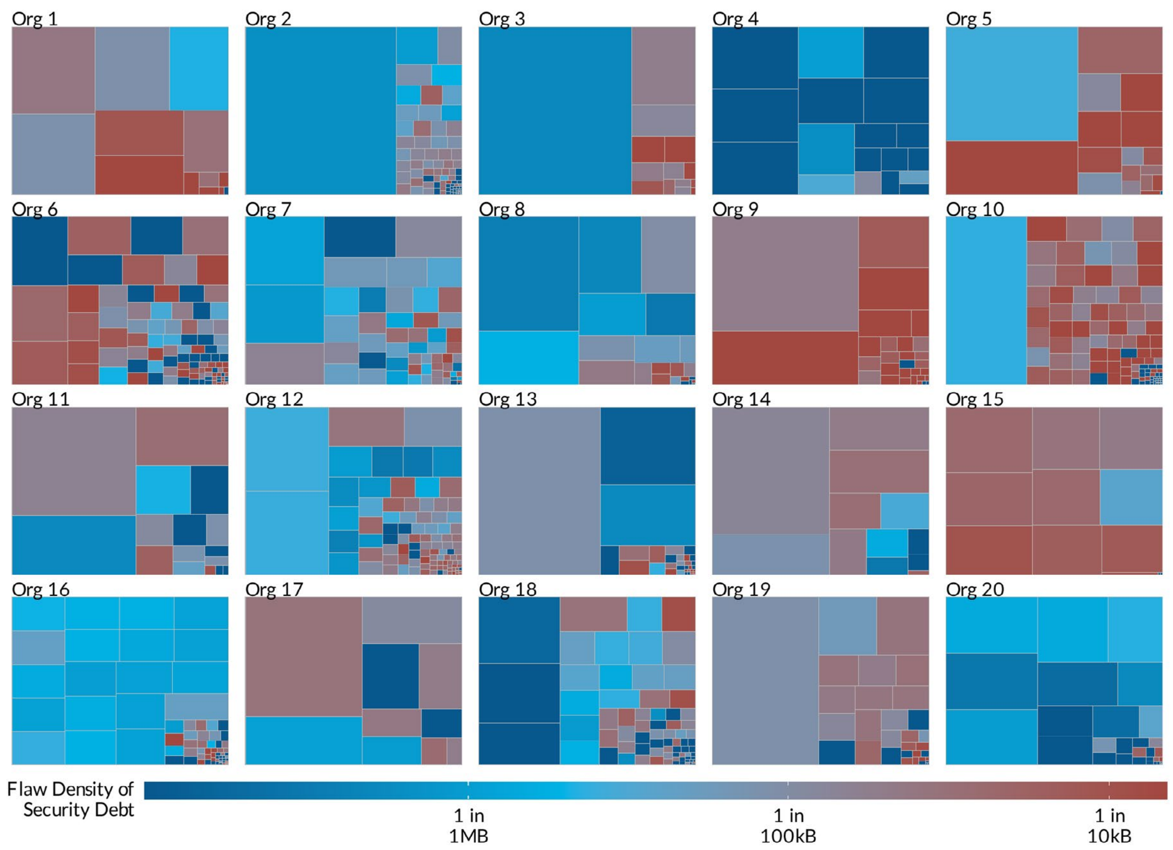


Figure 15: Distribution of security debt across applications >1yr old in 20 organizations

All data used in this report is anonymized. We're able to attribute flaws and applications to a randomized organization ID to enable analysis like what you see in Figure 15, but the data does not allow for identification of "Org #4."

Organization #4 is relatively clear of debt across most of its applications, with a few exceptions in the lower right corner. Organization #9 below it shows pretty much the opposite trend—one tiny debt-free application floats alone in a sea of debt. Other organizations in this sample exhibit a relatively even distribution of debt across applications (e.g., Org #14) to an application attack surface that runs the gamut of debt density (e.g., Org #6).

So, what makes one organization look so different from another when it comes to the distribution of security debt across their applications? Is it possible for teams to reverse indebtedness for the applications they develop and manage? Evidence presented in the next couple of sections points to the affirmative—keep reading!

Factors that contribute to security debt

How do applications and organizations fall into security debt? What dynamics are at play to drive it up or down over time? These are the types of questions we aim to answer in this section as we explore some of the contributing factors to security debt.

Application age

Since security debt has a time component, it makes sense that the age of the application may play a role in its accumulation. We've also observed in prior SOSS editions that the pace of flaw remediation tends to wane as an application ages.

The chart below suggests there is indeed an age factor driving security debt. Recall that unremediated flaws become debt after one year. At that point in an application's life cycle, ~42% of all flaws roll over into security debt. Indebtedness grows to ~62% at the three-year mark and rises further still to 75% after five years.

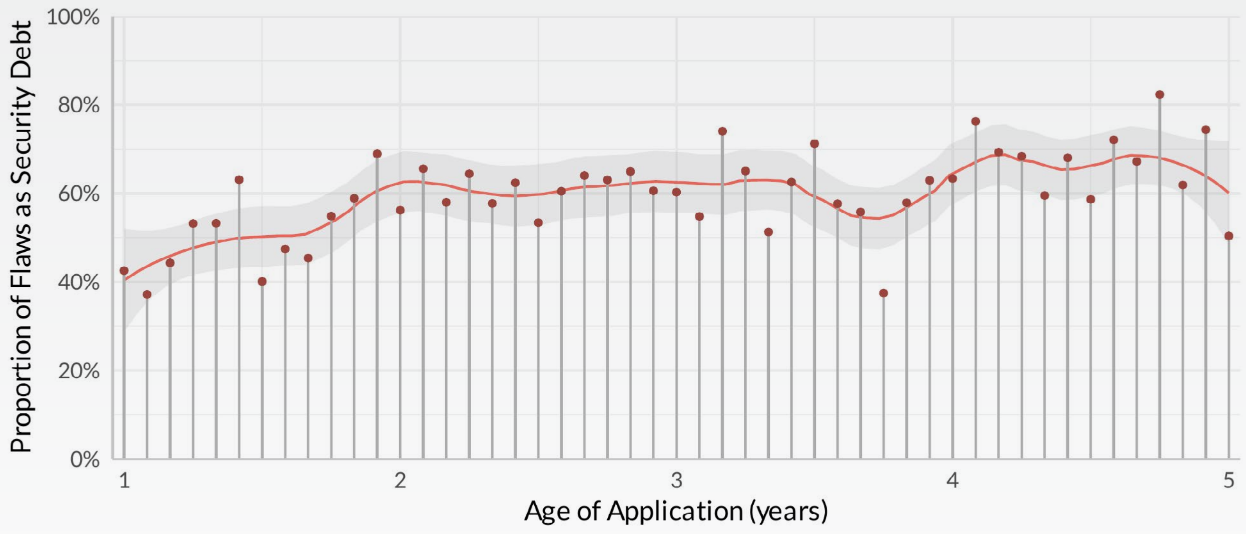


Figure 16: Proportion of flaws that qualify as security debt by age of application

Application size

The codebase of most applications grows over time, so it's logical that a correlation might exist between age and the accumulation of older, unremediated flaws. Let's test that theory.

The charts below show the distribution of security debt among applications grouped into similar age and size ranges. On the age scale, applications are considered younger if they're between 1 and 2.1 years old, middle between 2.1 and 3.4 years), and older after 3.4 years. Those are admittedly weird breakpoints, but they roughly divide all applications into three equal bins. We took a similar approach for grouping small (<250kB), medium (250kB-1.55MB), and large (1.55MB+) applications based on the size of their codebase.

Ignoring the values in the shaded squares for a moment, let's note the overall prevalence of security debt listed below the size labels on the vertical axis. Large applications have the highest proportion of security debt (40% of applications) and critical debt (47%). Medium-sized applications fall a little below that, and the debt ratio among small applications is lower still in both charts.

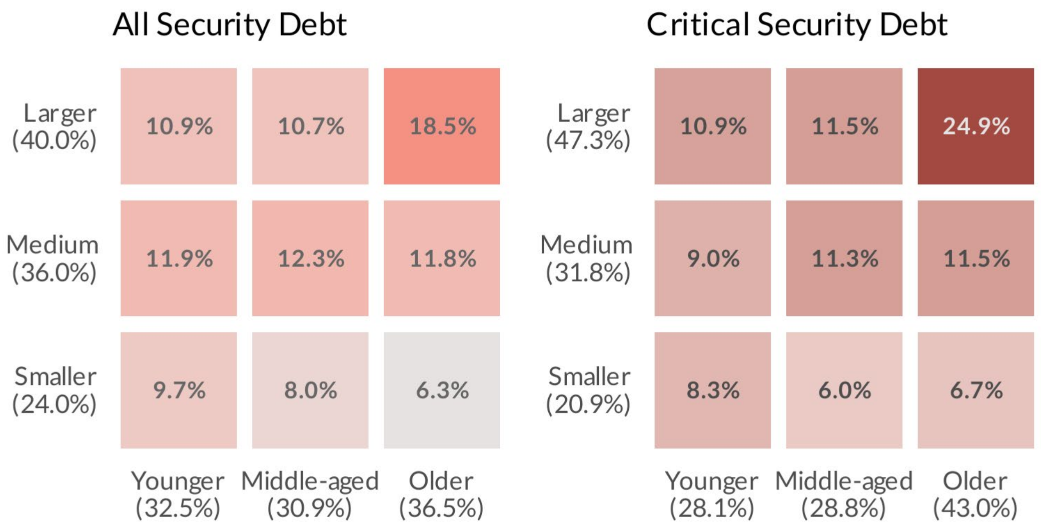


Figure 17: Distribution of security debt across application age and size

Now, on to the shaded age-size groupings. It's obvious that both security debt and critical debt concentrate most intensely in the upper-right section of large legacy applications. That tracks with our assumptions and observations.

That fact may lead one to assume that the youngest, smallest apps would be the least riddled with debt, but that does not appear to be the case. That designation goes to older small applications in both charts. But overall, we definitely see a size factor contributing to security debt here.

These results won't settle the perennial debates on the vices and virtues of monoliths vs. microservices in AppSec. But on the topic of managing security debt, monoliths clearly present a greater challenge.

Application language

In prior SOSS versions, we've shown that the language used to develop applications has a strong bearing on security outcomes across the board (flaw types, prevalence, fix times, etc.). It seems like a foregone conclusion, then, that languages would have a lot to do with an application's debt destiny too. But there's no need to rely on assumptions here; we have the data to know for sure.

Figure 18 compares the security debt profile for different development languages based on the most recent scans. It does this by plotting the proportion of applications (x-axis) and flaws (y-axis) that exhibit debt (left) and critical debt (right). The size of the dot corresponds to the number of applications for each language.

It's admittedly not a chart lending to quick and easy takeaways. But if you really want to understand the debt profile of the languages you use, the juice here is worth the squeeze. We'll start with a couple squeezes and sips of our own to get the juices flowing.

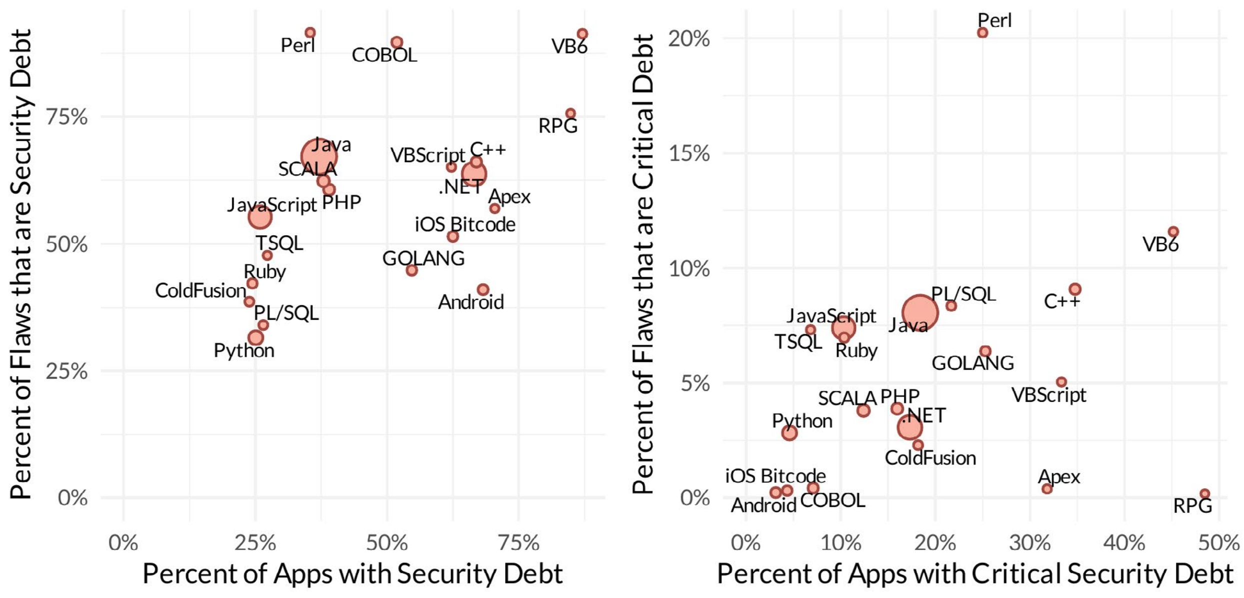


Figure 18: Prevalence of security debt by application development language

Remember, different languages have inherently different security postures, environments, and controls. Developers can compare how their languages perform and get a view of areas for future focus.

The top languages in terms of total applications—Java, .NET, JavaScript—land mostly in the middle of the pack for security debt and critical debt. Holdovers of legacy codebases, such as VB6 and Perl, exhibit the higher rate of indebtedness that likely squares with the expectations of most readers. We also note the low levels of critical debt for iOS and Android apps.

First vs. third-party code

We saw earlier that flaws in third-party (open-source) code tended to become security debt at a slightly higher rate (48%) than those in first-party code (41%). Let's pull on that thread a bit more here to see just how big a role this plays in the piling up of security debt in your applications.

The aforementioned remediation statistics represent the probability that first or third-party code in applications contains unremediated flaws older than one year. Another way to look at this is to isolate all security debt that exists across all applications and then determine what proportion of that debt is in first-party or third-party code. We've done exactly that in Figure 19.

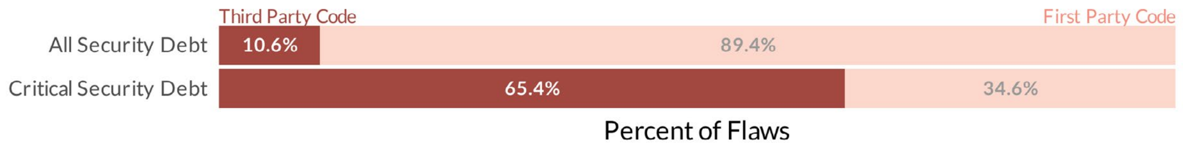


Figure 19: Proportion of security debt and critical debt in first-party vs. third-party code

The top part of the chart makes it clear that the vast majority (89%) of all security debt across all active applications exists in first-party code. So, statistically speaking, your own code is 90% of your debt problem. That strongly suggests that organizations wanting to drive down debt most effectively should focus first on code created by their own developers.

BUT there's another side of the story (and chart) here. When we narrow in on critical security debt, third-party code comes to the forefront with a two-thirds majority. This suggests that organizations wanting to drive down debt representing the highest risk should focus first on third-party libraries that often constitute a large proportion of their codebase (see Figure 34).

Flaw types

It's tempting to think of all security debt as resulting from oversight, poor decisions, failure to execute, etc. But that's not always the case. Sometimes, developers make conscious decisions to fix certain flaws while letting others lie. The data can't reveal to us the logic behind such decisions, but it does clearly show that certain types of flaws are more likely to become security debt.

Figure 20 lists the CWE categories we analyzed for prevalence back in the "At a Glance" section. This time, however, we're focused on the likelihood of first-party flaws in each category hanging around long enough to become security debt (based on SAST and DAST scans). Ostensibly, those toward the top of the list tend to be addressed faster because they're perceived to represent more risk or are easier to fix.

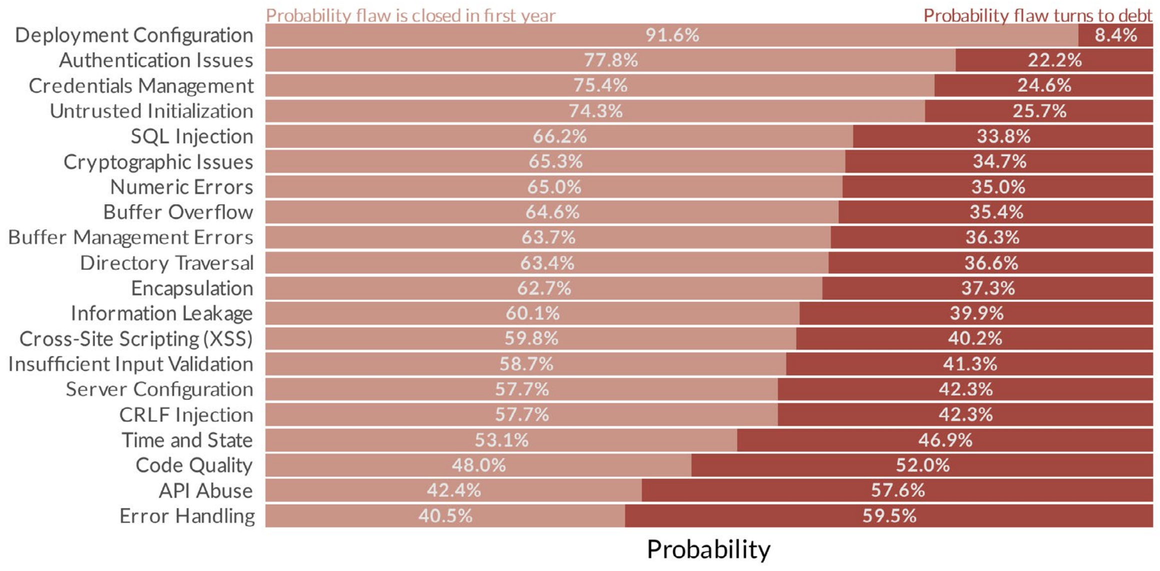


Figure 20: Probability of first-party flaws converting to security debt by CWE category

Because developers don't tend to set priorities for remediating third-party code, one may expect to see different results for flaws identified through SCA scans. Indeed, Figure 21 reveals many new CWE categories as well as substantial shifts in the ordering among existing categories. For example, Credentials Management and Authentication Issues are among the least likely to become debt in first-party code, but the opposite is true for third-party code.

Many of these differences can be traced back to how these flaws are fixed. The developer sets the priority and fixes flaws in first-party code. Third-party code is largely fixed by updating the libraries, so the type of flaw isn't really a consideration. The correction sometimes given to children applies here: "You get what you get and you don't get upset."

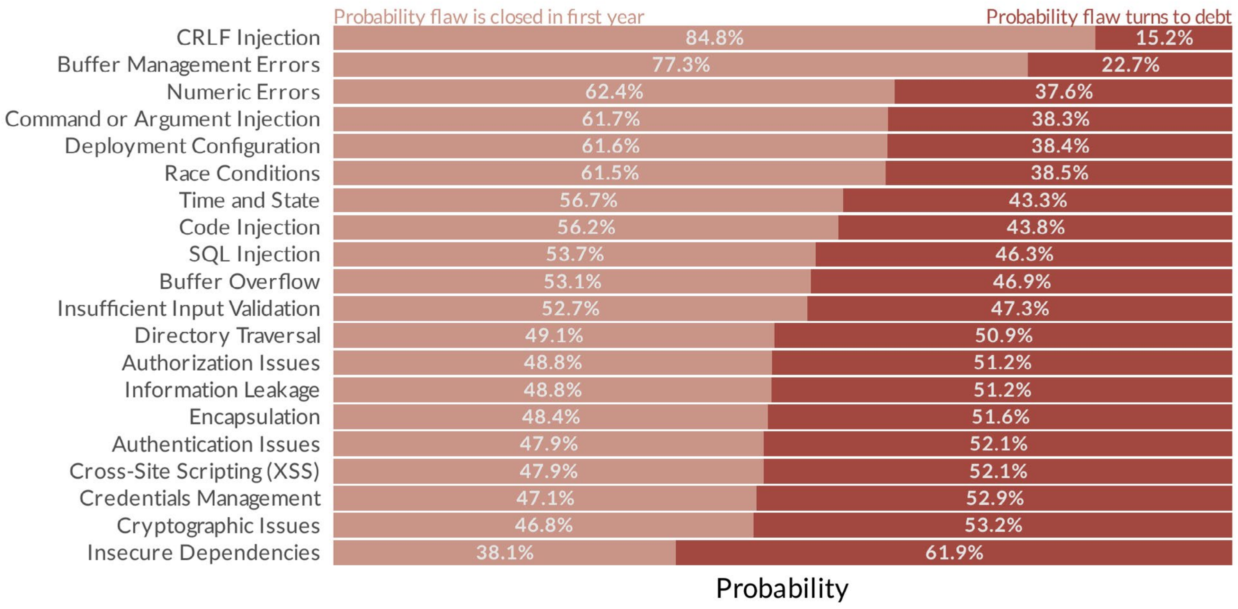


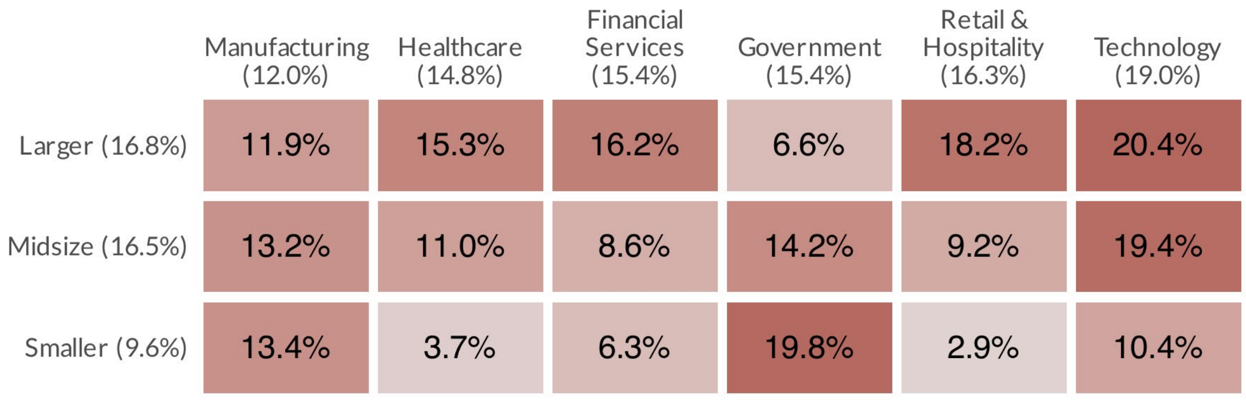
Figure 21: Probability of third-party flaws converting to security debt by CWE category

We don't go deep into it in this SOSS, but we want to stress that the most common types of flaws differ for each language. It's important for developers to be aware of flaws most relevant to their language and how they are introduced to remove the chance of them adding to security debt down the road. Here's a quick summary from SOSS 11 of the top flaws per language to help foster awareness.

Firmographic factors

We've shown several views of security debt at the organizational level in this report. That's because, regardless of the many technocentric drivers, managing security debt is ultimately up to the organization. And some organizations (and types of organizations) will do that better than others. We'll make some brief comparisons of the prevalence of critical security debt across organizations of different sectors and sizes here.

Overall, there's a 7x disparity between the segment with the highest ratio of debt (large tech) and the lowest (small retail/hospitality). The differences aren't as dramatic when comparing industries as a whole (high of 19% vs. low of 12%). In general, it appears that midsize and larger organizations carry more security debt than smaller firms, though there are notable exceptions. That's probably a reflection of the difficulty inherent to maintaining an increasingly large codebase as the organization grows.



	Manufacturing (12.0%)	Healthcare (14.8%)	Financial Services (15.4%)	Government (15.4%)	Retail & Hospitality (16.3%)	Technology (19.0%)
Larger (16.8%)	11.9%	15.3%	16.2%	6.6%	18.2%	20.4%
Midsize (16.5%)	13.2%	11.0%	8.6%	14.2%	9.2%	19.4%
Smaller (9.6%)	13.4%	3.7%	6.3%	19.8%	2.9%	10.4%

Figure 22: Percent of applications >1yr old in each sector with critical security debt

Government (which includes education) and, to a lesser extent, manufacturing sectors buck the "bigger = more debt" trend. We can't help but suspect that a dependence on legacy technology and strained IT and security resources have a lot to do with this reversal of fortunes among small government institutions.

If this leaves you wanting more, that's exactly what we intended. We wanted more, too. And that's exactly why we'll be releasing several industry-specific mini-reports in the near future to make this analysis even more relevant and useful. Stay tuned!

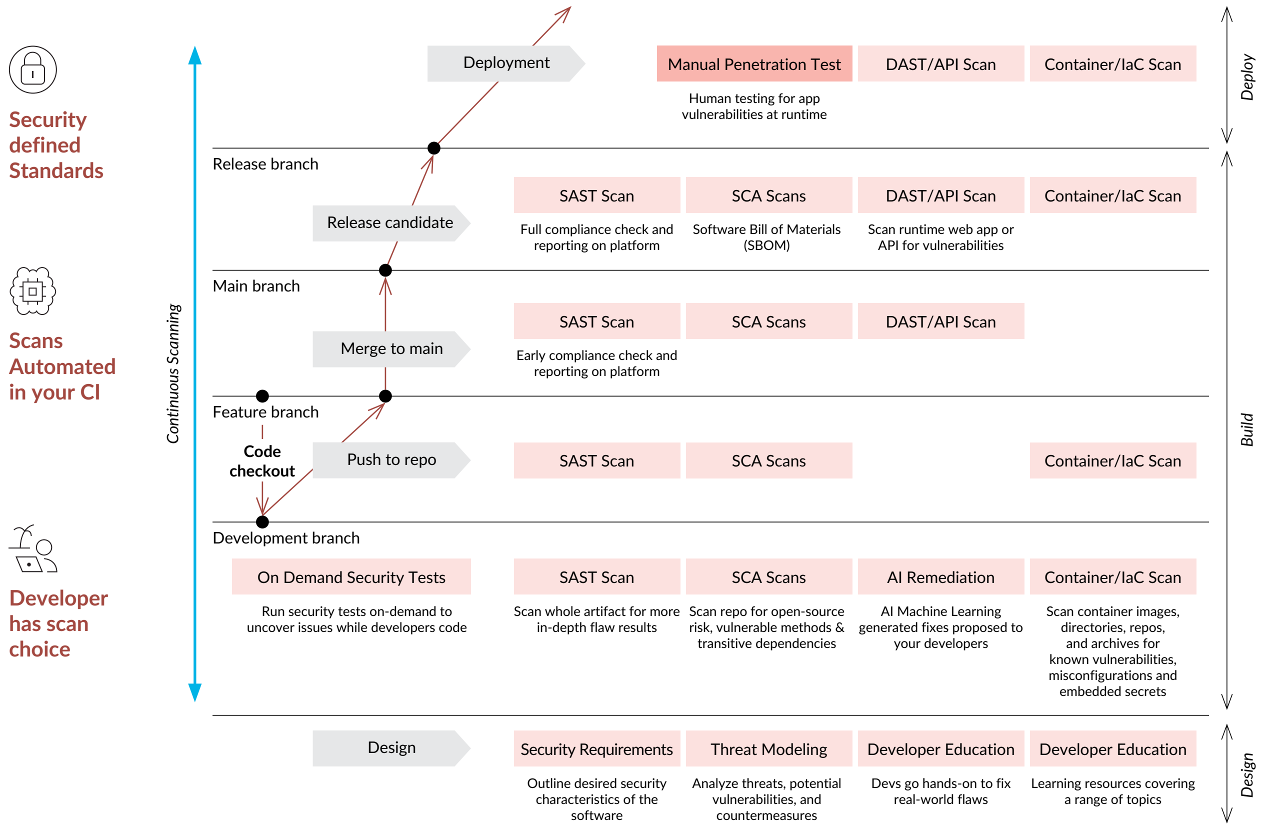
Recommendations to minimize security debt

We now know what security debt is, why it forms, and where it tends to pile up within applications and organizations. What we haven't yet covered is how organizations can begin getting rid of it. This section offers five evidence-backed strategies for minimizing security debt. It's our hope that these observational recommendations will empower your development team to create and maintain more secure software.

Integrate security into the entire SDLC

In prior SOSS, we presented findings specific to SAST, DAST, and SCA separately and supplemented that analysis by comparing and contrasting them as appropriate. This is the first volume in which we show a combined view across them. There are some good reasons for analyzing them separately, but combining the results is more in line with how we seek to integrate security into the entire SDLC.

Rather than testing in silos—whether those be in code, processes, or teams—continuous scanning via SAST, DAST, and SCA at various points along the Design, Build, and Deploy phases is a much more effective way to combat mounting security debt. Findings from each of those would obviously be reviewed and remediated, but within the broader context of the SDLC to maximize efficacy. This is exemplified in the diagram below.



Security defined Standards
Scans Automated in your CI
Developer has scan choice
Continuous Scanning
Deployment
Manual Penetration Test
Human testing for app vulnerabilities at runtime
DAST/API Scan
Container/IaC Scan
Deploy
Release branch
Release candidate
SAST Scan
Full compliance check and reporting on platform
SCA Scans
Software Bill of Materials (SBOM)
DAST/API Scan
Scan runtime web app or API for vulnerabilities
Container/IaC Scan
Main branch
Merge to main
SAST Scan
Early compliance check and reporting on platform
SCA Scans
DAST/API Scan
Feature branch
Code checkout
Push to repo
SAST Scan
SCA Scans
Container/IaC Scan
Build
Development branch
On Demand Security Tests
Run security tests on-demand to uncover issues while developers code
SAST Scan
Scan whole artifact for more in-depth flaw results
SCA Scans
Scan repo for open-source risk, vulnerable methods & transitive dependencies
AI Remediation
AI Machine Learning generated fixes proposed to your developers
Container/IaC Scan
Scan container images, directories, repos, and archives for known vulnerabilities, misconfigurations and embedded secrets
Design
Security Requirements
Outline desired security characteristics of the software
Threat Modeling
Analyze threats, potential vulnerabilities, and countermeasures
Developer Education
Devs go hands-on to fix real-world flaws
Developer Education
Learning resources covering a range of topics
Design

Figure 23: Process for integrating security testing into the SDLC

If that concept seems a tad “pie in the sky” to you, the next chart will help make it more real. The fact of the matter is that without integrating multiple scan activities, you'll never have a holistic view of security debt across your applications.

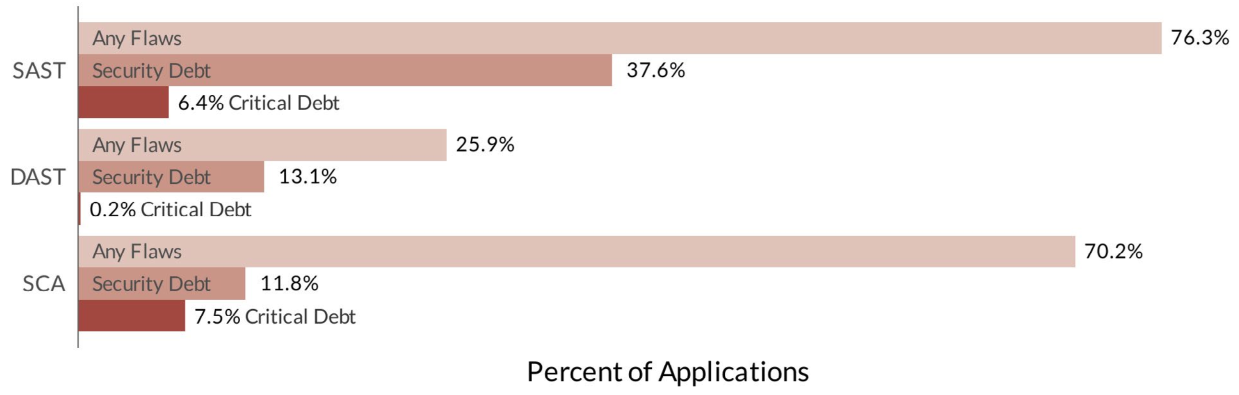


Figure 24: Comparison of security flaw and debt detection rates among scan types

Figure 24 shows the percentage of applications that have security flaws as detected through SAST, DAST, and SCA testing. The proportion with security debt is significant for each type of scan. It's tempting to conclude that DAST scans don't find critical debt based on the low ratio indicated in the chart. But we know that's not the case because DAST findings tend to have a higher severity rating overall. We attribute this more to developers prioritizing critical flaws detected by DAST for remediation so they're less likely to become debt.

Relying on a single testing method (e.g., SAST)—especially if done only once at a single point in the SDLC—is not a strategy for debt elimination.

Move toward continuous remediation

News flash: Organizations that fix flaws faster have less security debt. Ok, maybe that's not a headline that shocks you. But you'd still like to verify that it's true, right? So do we.

Before we do that, though, we'll share something that did shock us—more frequent scans do not correlate with less debt (at least according to the definition of debt we're using here). Yep; that was counterintuitive to us as well, especially since we've shown in the past that increasing testing cadence leads to faster remediation (that still holds true, by the way). Our best explanation is that knowing is only half the battle.

The other half of that battle is actually doing something about it. Applying that to AppSec, more frequent scans will undoubtedly give dev teams up-to-date knowledge of the flaws that exist in their code. But if that knowledge doesn't translate to expedient remediation, debt will continue to pile up in the applications they maintain. Continuous scanning must be accompanied by continuous remediation to be effective.

Figure 25 proves this claim. We measured the speed of remediation for all active applications and split them into three equally sized buckets: slow, medium, and fast. Less than 50% of applications in the fast bracket have security debt and only 5% have critical debt. Compare that to an indebtedness rate of over 90% for slow applications, with critical debt plaguing just under one-quarter of them.

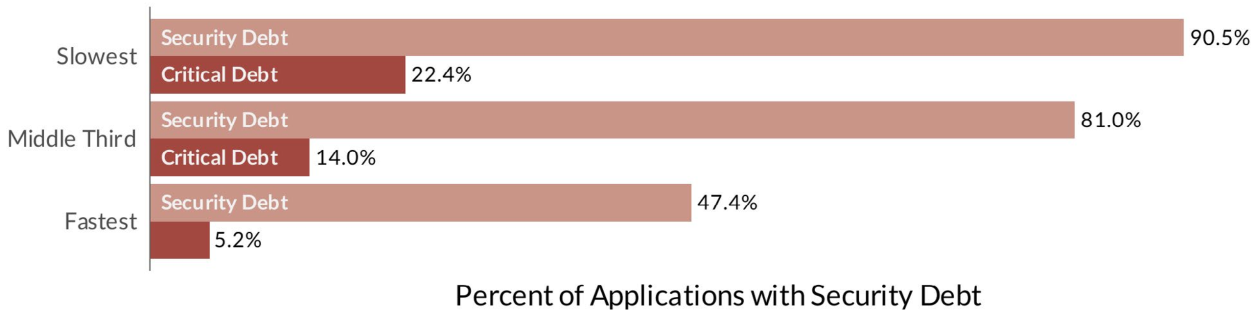


Figure 25: Effect of flaw remediation speed on prevalence of security debt

That means the slowest third of applications have four times more critical security debt, on average, than those with the fastest fix rates! It's also noteworthy that security debt doesn't substantially depreciate when moving from the slow to medium tier. Rather than justification for the status quo, development teams should see that as a strong incentive to strive for continual improvements in remediation speed to unlock maximum benefits.

Of course, speed isn't everything. Mahatma Gandhi once said, "Speed is irrelevant if you're going in the wrong direction." We discuss pointing your remediation efforts in the right direction in the next recommendation.

Prioritize remediation of critical security debt

We've shown that working faster is critical for combating security debt in software. We'll now show that working smarter must be a core pillar in your debt elimination strategy, too.

Circling back to the Gandhi quote that closed out the last section, remediation needs a direction or object of focus. Flaws are an obvious candidate for that focus, but many organizations simply cannot remediate them all consistently. The high levels of debt witnessed in this report attest to that.

Risk offers a better and more achievable object of focus for remediation. And what's more risky than critical security debt? Unfortunately, the overall track record for organizations isn't stellar in terms of prioritizing critical security risk. This is evidenced by Figure 26, where the survival curves (see callout) for critical and non-critical flaws follow a similar path.

Survival Analysis for Flaw Remediation

For several years now, the SOSS has used a technique called survival analysis to produce flaw remediation curves like those shown in Figure 25. Survival analysis has been applied to many natural and man-made phenomena where measuring the time to key events is needed. In the case of the SOSS, the curves trace how quickly flaws die out (through remediation) and how many are still alive (unresolved) at any given time. For example, 44% of critical flaws are still unresolved one year after their discovery.

Frankly, it's a minor miracle that we've gotten this far into a SOSS without yet showing survival curves. This refresher should assist with the ones you'll encounter over the next few sections.

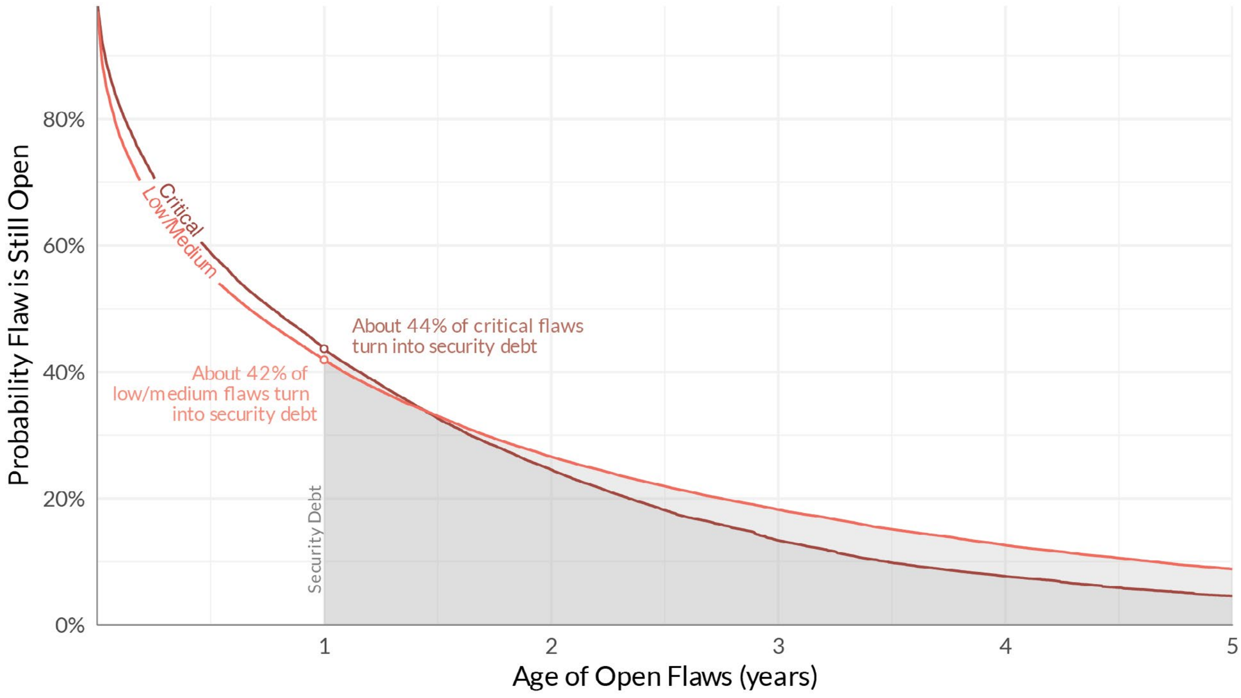


Figure 26: Comparison of remediation timelines for lower-severity vs. critical flaws

Ideally, we'd like to see the dark red line for critical flaws dive sharply down with a very low percentage that become security debt at the one-year mark. The reality is that there's virtually no difference in the ratio of debt conversion between critical and non-critical flaws (44% and 42%, respectively).

Setting this disappointing track record aside for the moment, let's establish a baseline target for how many flaws organizations would need to remediate to keep critical security debt from piling up across applications. Figure 27 carves up all flaws identified across the entire codebase of the typical organization into four quadrants. The horizontal separation is between critical and non-critical flaws. The vertical axis divides flaws that are currently debt vs. those that aren't (yet).

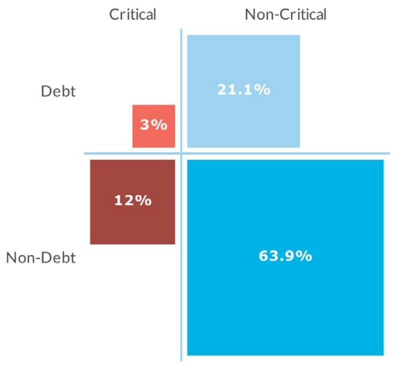


Figure 27: Distribution of all flaws based on severity rating and security debt status

Close to two-thirds (64%) of all flaws are neither debt nor critical severity. We're not saying ignore those (or they'll eventually become debt), but remediation of those flaws can be deferred in favor of those that represent greater risk. The same can be said for non-critical debt, which accounts for another 21% of all flaws. Thus, 85% of flaws across your applications don't deserve top priority from your development team.

Instead, focus their finite remediation capacity on the 3% of flaws that constitute critical debt. Once they've tackled those, focus them on fixing the remaining 12% of critical flaws so they never become debt in the first place.

We've discussed remediation capacity in previous SOSS reports, but let's define it again here as a refresher. To measure capacity, we look at all the flaws facing a development team in a given month and then see how what percentage of those are fixed during that month. This reflects how much time the team has chosen to allocate to fixing security flaws.

Looking again at Figure 27, we now know that if a team's remediation capacity is 3%, they could theoretically eliminate their critical security debt within that month. A team with a 15% remediation capacity could fix all of their critical issues -- both debt and non-debt. But is that realistic? Can organizations actually hit that mark? Figure 28 gives the very thought-provoking answer.

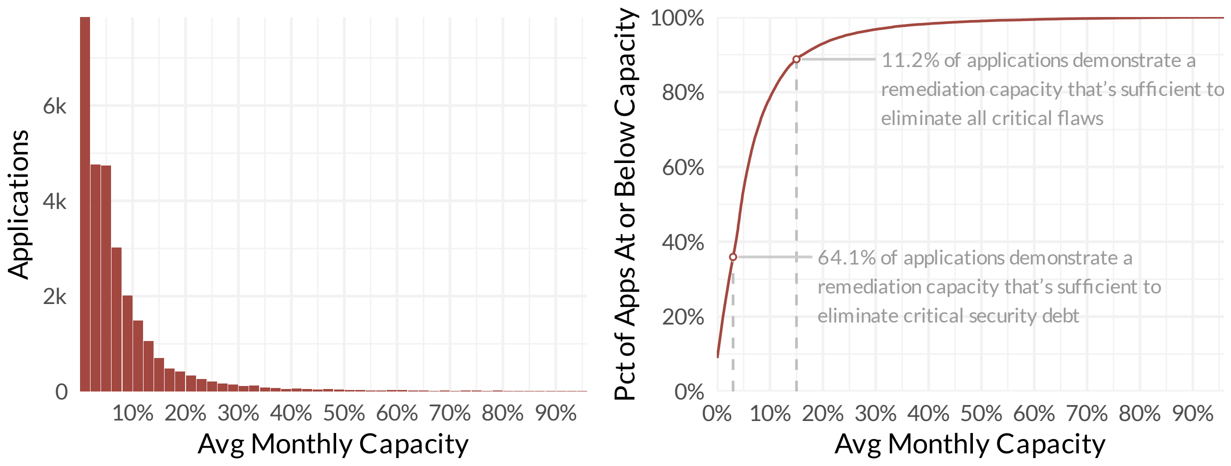


Figure 28: Average monthly flaw remediation capacity across applications

Per the chart on the left, the average monthly remediation capacity for most applications is less than 10% of all flaws. There's a long tail of applications that boast rates above 20%, but it drops off quickly. So a decent number of applications are hitting our baseline mark of 3%, but not many reach the ideal goal of 15%.

Remediation capacity is more of a choice than a ceiling. Though the word “capacity” connotes an inherent limitation, it’s choices that truly dictate how many flaws can be remediated in a given timeframe. As just one example, devoting time during sprints specifically to the elimination of security debt could be an effective tactic to increase capacity.

The chart on the right adds some concrete statistics to that general observation. About 64% of all applications in our sample demonstrate a remediation capacity that’s sufficient to eliminate critical security debt, but only 11% regularly demonstrate rates required to address all critical flaws.

We’d like to paint a rosier picture of realized flaw remediation capacity and reassure you with a confident “You can do it!” But the reality revealed by the data is that you probably can’t do it—at least not without making some changes. The good news is that the findings in this report highlight numerous opportunities for meaningful changes that will give your team the best chances of success.

Build developer security competency

For developers to execute strategies such as prioritizing critical security debt, they need to understand what it is, where it exists, and how to accomplish it. We hope this report helps in that regard, but it falls far short of a sustained program to build developer security competency.

A recent survey from ESG found that less than 50% of organizations require their developers to engage in formal training more than once each year. Another Veracode study revealed that nearly 70% of developers say their employers don’t offer adequate security training. That suggests many organizations are going through the motions rather than building a real “shift left” culture.

This lack of regular security training creates friction between development and security teams, making it difficult to secure applications efficiently. What do the results of that friction look like in practice? Figure 29 paints us a picture using the example of Veracode’s Security Labs.

Security Labs are designed to instill secure coding practices across your organization. They employ real, containerized applications and APIs in action-based labs that allow developers to find, fix, and explore security flaws to see the impact on an application. Figure 29 demonstrates that organizations using these labs exhibit faster flaw remediation and have less tendency to pile up debt.

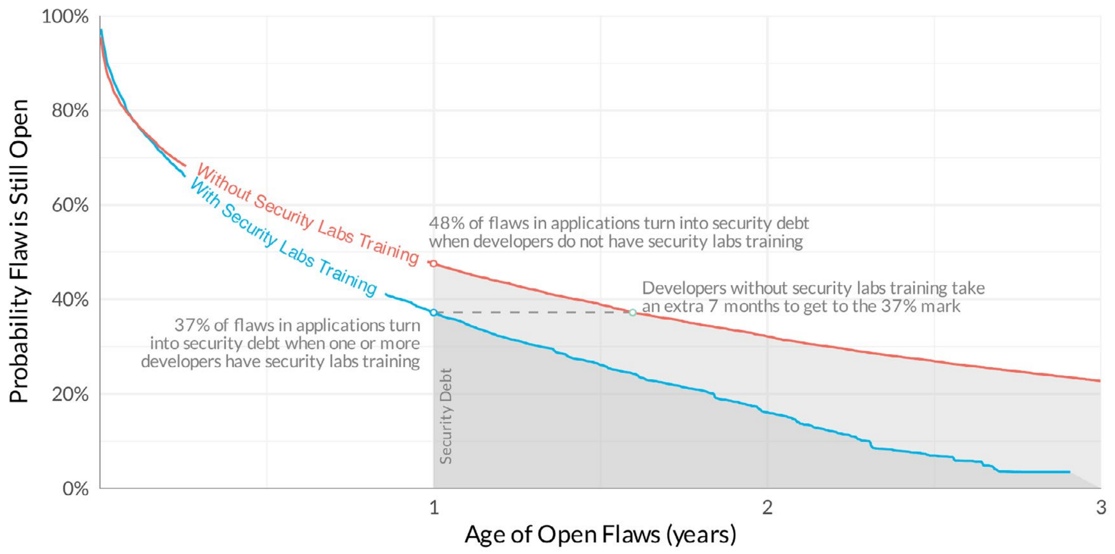


Figure 29: Comparison of remediation timelines for teams that use vs. don’t use Security Labs

Among organizations that use Security Labs, 37% have security debt. Compare that to 48% among application teams that do not. The time-to-fix difference is even more significant. Applications developed by teams that aren’t using the labs take seven months longer to reach that 37% mark.

Of course, labs aren’t the only way to build developer competency. It starts with aligning priorities to secure development objectives, includes various forms of regular learning activities, and carries through to measurable performance outcomes. You’ll find resources for getting started with all of these here.

Know your language's debt profile

Regardless of what aspect of AppSec is being measured, development language is a major factor. From the prevalence of flaws to a proclivity for security debt, much of an application's security posture ties back to its underlying code. The language-based survival analysis in Figure 30 proves that the metric of remediation speed is no exception to that rule.

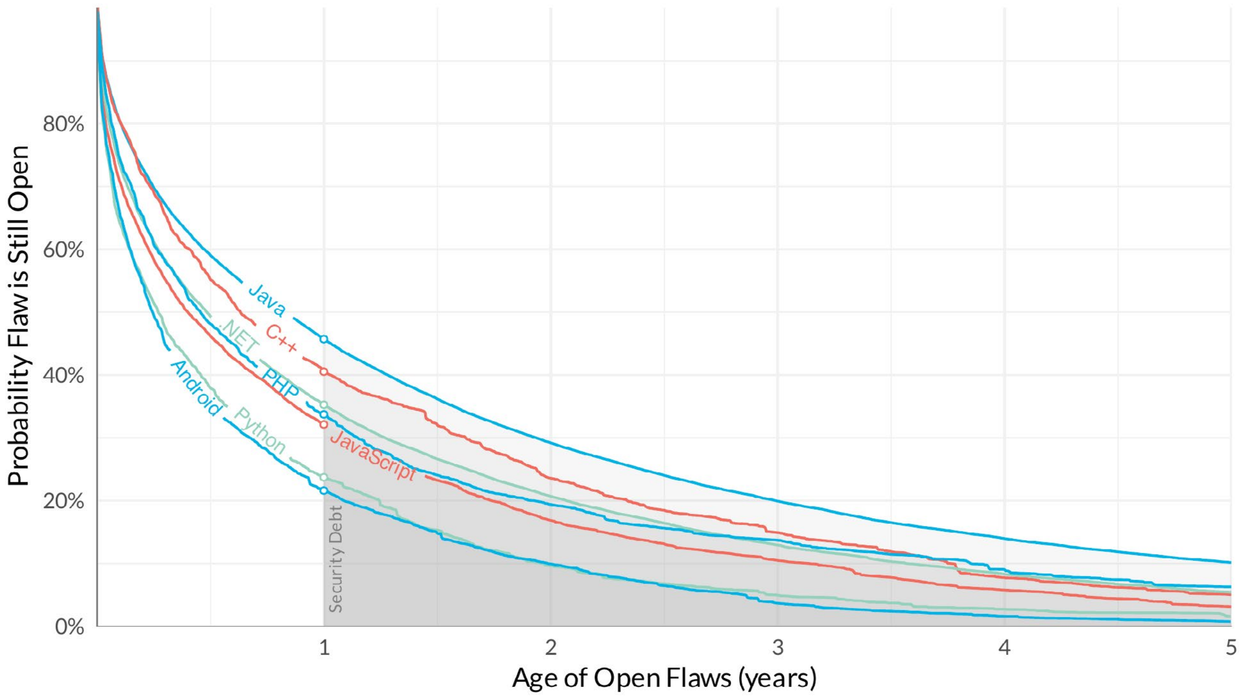


Figure 30: Comparison of remediation timelines for application languages

Figure 30 expands on what we learned in Figure 18. Namely, flaws in some languages (e.g., Java and C++) are far more likely to become debt than others (e.g., Python, Android). The comparative probability of indebtedness is indicated for each language at the one year mark. For example, any given flaw in a Java application has a 46% chance of turning into security debt, while it's half that rate among Python applications. This likely stems from Java's role in large, complex enterprise applications and Python being popular for lighter apps.

The main point we want to get across here is that applications developed in different languages will require different strategies for reducing security debt. What works in Python probably won't work in Java, and the amount of resources needed will likely differ as well. Understanding the strengths and weaknesses of the languages used by your organization is fundamental to that strategy.

Given that, one may conclude that choosing one or two primary development languages to specialize in would make the job of creating and maintaining debt-free code easier. The data, however, doesn't support this conclusion.

The analysis in Figure 31 compares the level of security debt and language diversity among organizations. Those on the left are more of a language monoculture across their applications, while those on the right are far more language-diverse. As you can see, there's no significant difference in security debt between those groups. That said, organizations with less language diversity do tend to show a lower prevalence of critical debt.

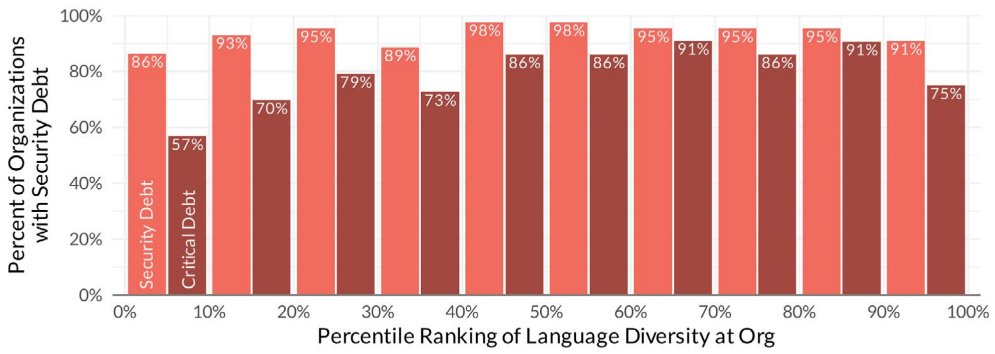


Figure 31: Effect of language diversity on security debt and critical debt

The takeaway in all this is that it's very important to create a strategy that accounts for the debt-related proclivities of the languages used by your development teams. But there's no need to force either consolidation or diversification on them as part of that strategy. Focus instead on finding a trusted partner that offers testing tools and educational resources in the languages your developers use. This will improve the efficacy of flaw remediation and reduce debt across your applications.

Develop a strategy to secure the software supply chain

We've raised several red flags thus far, pointing to a less than stellar track record for security debt in open-source code. Third-party flaws are prevalent (affecting 70% of apps), are more likely to become security debt, and house the majority (~2/3rds) of critical debt. At the risk of beating a dead horse, we'll add one more chart before moving into proactive strategies for protecting the software supply chain in the next section.

Figure 32 expands on a statistic we shared in the "At a Glance" section, pointing to slower remediation of flaws in third-party libraries. The survival analysis depicted here reveals that the half-life of flaws in third-party code is 11 months compared to 7 months in first-party code. The momentum appears to shift around the 2.5-year mark, but that doesn't negate the premise that managing flaws in third-party code requires a separate strategy.

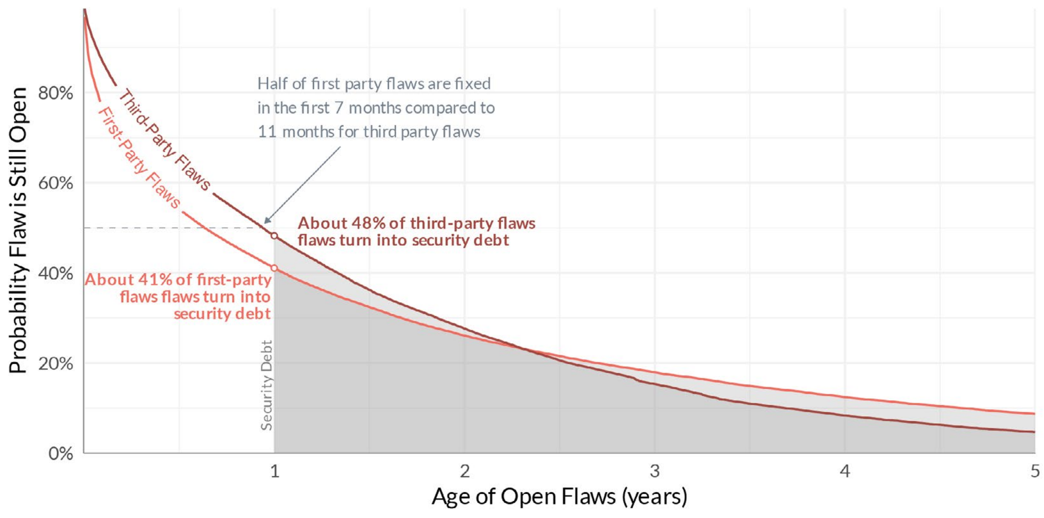
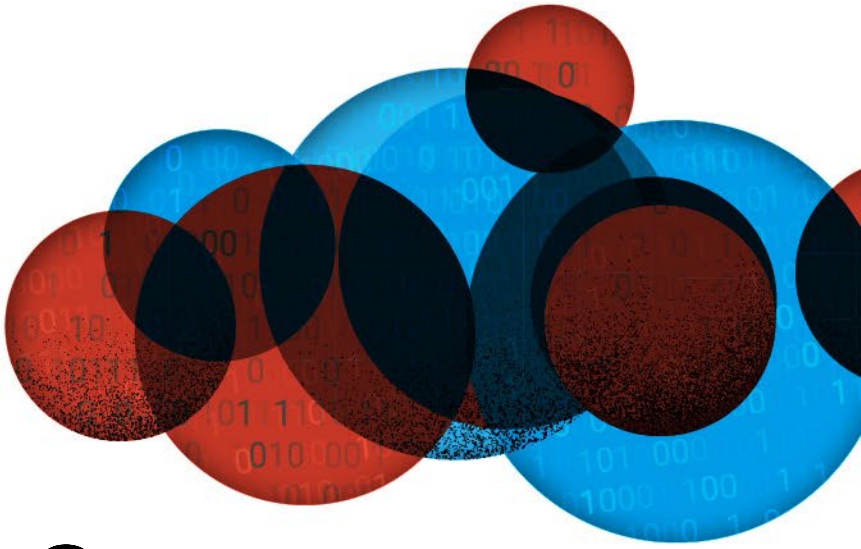


Figure 32: Comparison of remediation timelines for first-party vs. third-party code

What does that strategy entail, and how can your organization begin implementing it? We're glad you asked. Read on to find out.



Securing the Software Supply Chain

Up until now, we've spoken about the challenges and landscape of AppSec in third-party and first-party code in nearly the same breath. But we haven't really addressed the elephant in the room: Managing third-party (open-source) code security is an entirely different animal (elephant even) than managing first-party code.

Organizations are royalty within their own castle with first-party code. Everything is malleable (within reasonable limits) to meet the needed functionality and security goals. But once third-party code enters the picture, things become less controlled. Suddenly, the codebase relies on developers, infrastructure, and a development pipeline that are all outside your organization's purview, not to mention the dependency tree that can branch pretty quickly.

In this section, we'll dive into some of the specifics of third-party code use and how it can affect an application's security. In particular, we leverage some open standards to try to identify signs that point to some code repositories being more secure than others.

Understanding open-source dependencies

This fundamental question is one we explored in the original Open Source Edition of the SOSS. Back then, we examined a point-in-time snapshot of the number of libraries per application. But applications are dynamic, and we have updated that with a view over time. The result in Figure 33 is more interesting than we expected.

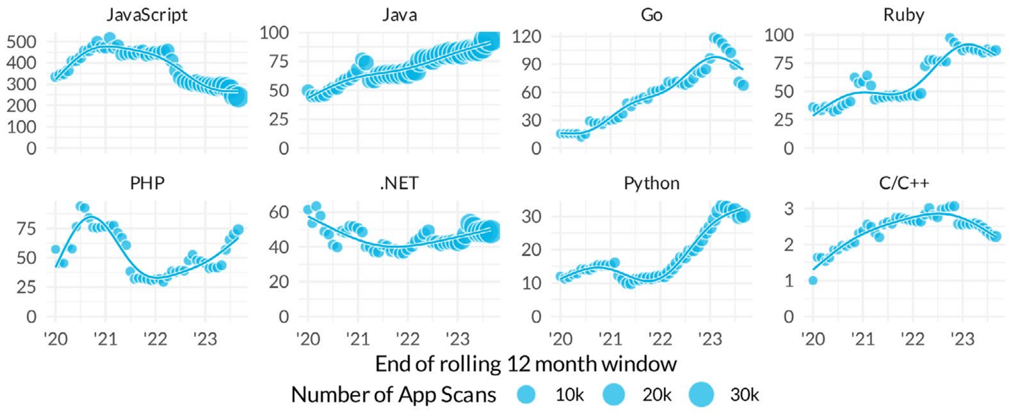


Figure 33: Number of libraries per application over time (rolling geometric mean)

The big takeaway here is that "homegrown" applications likely include dozens—or even hundreds—of libraries developed outside of the house, and the degree of those external dependencies appears to shift over time (note the different scales on the y-axis for each language). JavaScript applications seem to have peaked with nearly 500 libraries per application (on average), which has steadily declined in the last few years. Meanwhile, Java, Ruby, and Python are seeing expanding dependencies on third-party code.

The absolute number of libraries is one thing, but how much actual code volume this entails is another. In Figure 34, we examine how much of an application's actual codebase (measured in bytes) comprises first vs. third-party code. As with so many things in this space, complexity abounds.

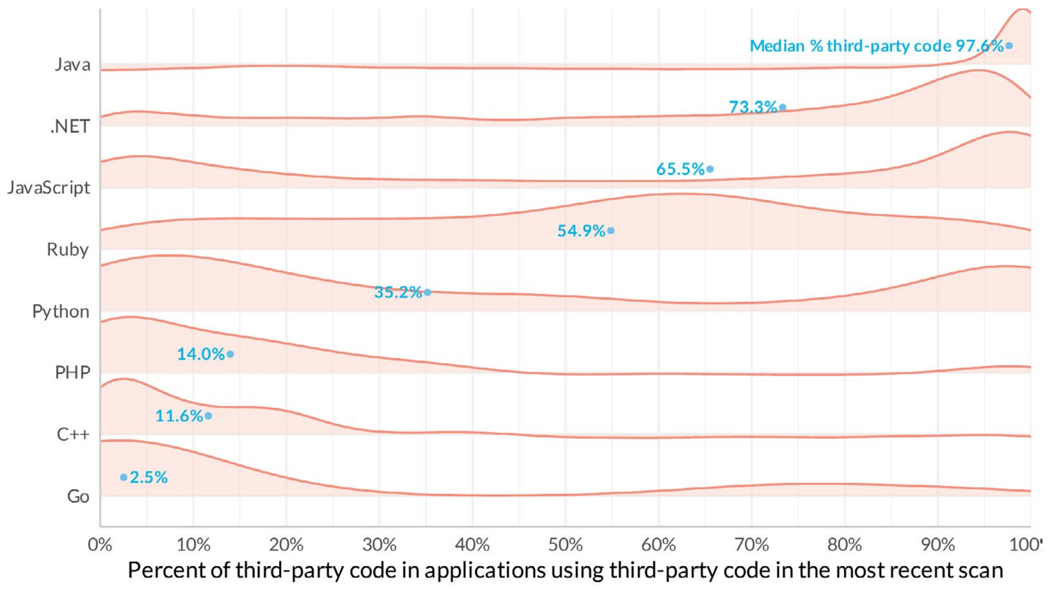


Figure 34: Percentage of an application that is third-party code. The light coral represents the distribution of apps in that language, while the blue points are the median for that language.

Here, we see that some languages tend to mostly consist of third-party code (e.g., Java at 98% and .NET at 73%), while others show the opposite trend (Go at 3% and C++ at 12%). Others (JavaScript and Python) appear to have "either/or" tendencies, with clusters on both ends of the spectrum (a "multimodal" distribution in statisticians' parlance). And then there are Ruby apps, which vary widely from mostly first-party to mostly third-party and everything in between.

It's no secret that many open-source repositories have only a handful of developers behind them. Furthermore, we've observed that some developers contribute to projects that become phenomenally prolific across the ecosystem of applications. So it's worthwhile to ask not only "What libraries are used most often for various languages?" but also "What percentage of applications use code from a single open-source developer?" The answer is pretty wild, so buckle up.

For this analysis, we focused on open-source libraries that could be mapped to specific code repositories and pulled the commit history for those repositories. Then we tallied the total number of applications using those libraries in a particular language that have commits by a single developer. The top 100 open-source developers for each language and the fraction of all applications using their code can be seen in Figure 35.

Though findings differ across languages, all of them have multiple developers whose code is included in 90% of applications. For PHP and Ruby apps, nearly all the top 100 open-source developers touch 90% of applications.

Given that these developers represent a single point of failure, it highlights the wide reach of potential weak links in the software supply chain. A single set of compromised credentials and some malicious code pushes could have extremely wide-ranging impacts. For that reason, we won't identify those developers/projects here. But this should sufficiently illustrate that a small subset of open-source developers can be a critical component of your software supply chain.

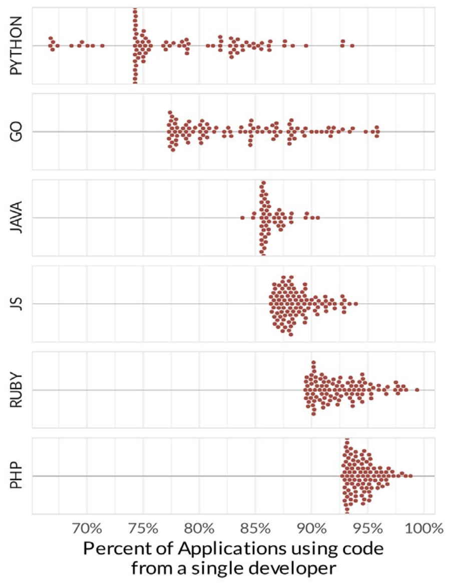


Figure 35: Percentage of applications using code from the top 100 open-source developers.

Without diving into other fancy visualizations, we'll also reiterate some of the results we've seen in previous SOSS reports are still indeed true even with another year and another big stack of scans:

- There is a small number of libraries that are found in most applications. In fact, broken down by language, the most popular libraries are found in at least 75% of applications.
- Most libraries are "set it and forget it," and not in the Ron Popeil rotisserie kind of way; more in the "import it and ignore it" kind of way.
- The number of direct vs. transitive dependencies varies by language, but the "dependencies of your dependencies" at least doubles your supply chain in most cases, and dependency-happy languages like Java and JavaScript can increase the size of your application's dependencies by 6x or 5x respectively.
- Nearly identical to last year's report, more than half of applications use libraries with less than 10 contributors or libraries that haven't been updated in over a year.

This little review should give us a notion of what the software supply chain looks like and sets up the all-important question: just how secure is it?

Assessing the security of third-party libraries

We'd love to see security as a key factor in selecting third-party libraries, but functionality usually drives those decisions. Even if two options provide similar functionality and security will cast the deciding vote, it's difficult to discern how secure one library is compared to another without testing them directly. That dilemma was one of the main drivers behind the Open Source Security Foundations (OpenSSF) "Scorecards" project.

The scorecard was developed to scan open-source repositories and identify potential and realized security issues. They include checks for things like binary artifacts in the codebase, use of branch protection, signs of dangerous coding patterns, known vulnerabilities, the use of code reviews and SAST scans, etc.

These checks can be useful to devs when examining their own repositories or evaluating others. They also provide us with a rich data set that we can combine with Veracode SCA data to get deeper insights into software supply chain security. This overlap between those sources can be seen in Figure 36.

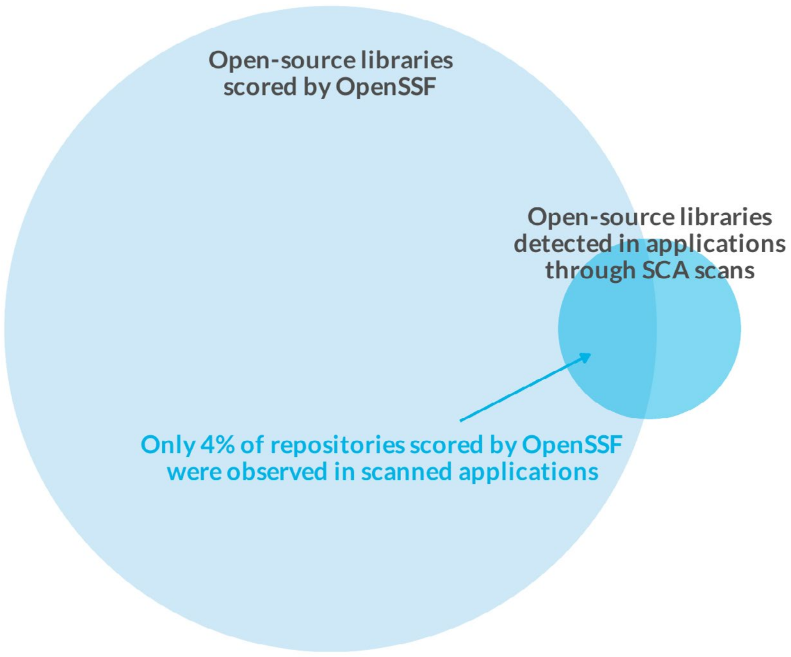

Open-source libraries scored by OpenSSF
Open-source libraries detected in applications through SCA scans
Only 4% of repositories scored by OpenSSF were observed in scanned applications

Figure 36: Overlap between libraries with OpenSSF scores (1.24M) and libraries used in applications scanned by SCA (97,887).

The larger cyan-shaded circle in Figure 36 represents the 1.24 million repositories (libraries) scored by OpenSSF, and the smaller blue circle represents the nearly 98 thousand libraries detected during our SCA scans of enterprise code. There are a few interesting discussions Figure 36 should open up. For example, the sheer quantity of open-source libraries and the management challenge that they present to organizations and developers. Now that challenge is made more manageable because only about 4% of those libraries are actually observed in use through SCA scans of enterprise software. But Figure 36 brings up another interesting discussion when dealing with third-party libraries: while naming things is hard, naming things consistently across multiple locations would give Sisyphus a run for his money when it comes to the futility. To outline the challenge in library names, we discover over 1,700 unique Javascript "react" libraries being used, with nearly 400 libraries just within the "react-native" world, and don't get us started on names that include their version string. But the outcome is a challenge in attribution and tracking libraries back to their source repository, making any automated attempt to monitor for updates, patches, or security advisories just that much more difficult.

The difference between a library existing versus actually being used does open the door for a comparison of the relative level of security between the two. Figure 37 compares the overall OpenSSF scores (higher is better) between the libraries we discovered in use, versus those we did not find evidence of use. In general, those used in customer applications scanned with SCA tend to have a higher score, perhaps suggesting that the more "successful" a library is, the more that security is a priority for library maintainers.

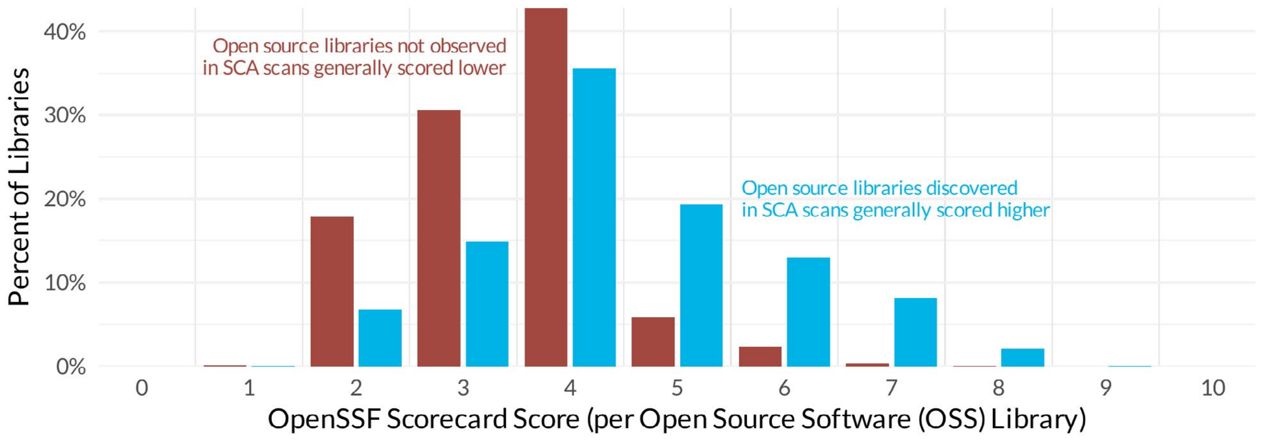


Figure 37: OpenSSF score for used (blue) vs. unused (red) libraries.

This is great information when a library's repository and code are available, but as we noted above, only about half the libraries observed in applications have a repo that we can successfully match to an OpenSSF scorecard. So what about those other ~40k libraries? Are there other reasonably discernible indicators that serve as a proxy for how secure they are?

A few candidates stand out that we'll test: the number of contributors, the time between active contribution days, the time since the last commit, and committer diversity. The first three are fairly self-explanatory, but the last one deserves an explanation.

We're referring to diversity in the ecological sense here. We want to measure not only the number of committers to a repository but how well the commits are spread across them. For example, if a repository has 100 contributors and one developer is responsible for 98% of the code, the codebase arguably has a single developer. A codebase with higher committer diversity would have a more equitable share of contributions.

So, how do these potential indicators of open-source security correlate with the OpenSSF scorecard? Pretty well, actually, according to Figure 38.

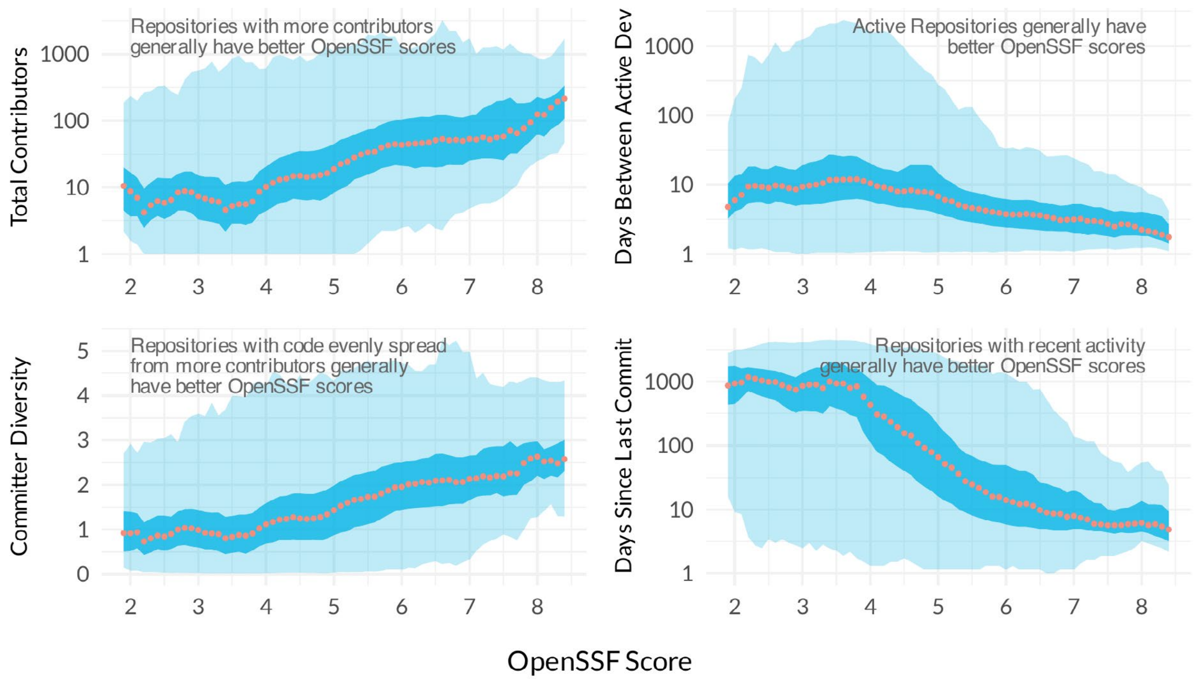


Figure 38: Relationship between open-source repository activity indicators and OpenSSF Score

The red line in Figure 38 traces a rolling median for all libraries at each score level. The darker blue region extends from the 25th to the 75th percentile, and the lighter blue covers the full range (it's basically a box plot in motion over the OpenSSF score). Since there aren't many libraries that receive an OpenSSF score of 1 or 9 (0.2% of the scores were below 2.0 or above 9.0), we removed those from the chart. Even with those removed, the wide shaded regions reveal there's a high degree of noise or uncertainty around each of these indicators.

Having said that, there does appear to be at least some signal here:

- Libraries with more contributors tend to have better security scores.
- Other than some outliers with very insecure ratings, frequently updated libraries have better security scores.
- Higher committer diversity (equally shared workload) is a good indication of better security scores.
- Long lags between commits are a red flag for worse security scores.

So, if you are faced with using a library that hasn't been scored by OpenSSF and can't be scanned directly, examining commit activity may offer some indication of code that's likely to be more (or less) secure. But it's a weak signal with low confidence. Inspection will always trump inference when it comes to the security of open-source software.

Addressing security debt in the software supply chain

The final thing we'd like to examine when thinking about the software supply chain is whether the underlying security of a library affects how upstream applications accrue security debt. To test that, we'll borrow the OpenSSF scores for libraries again for which we also have SCA scans (the overlap in the Venn diagram of Figure 36).

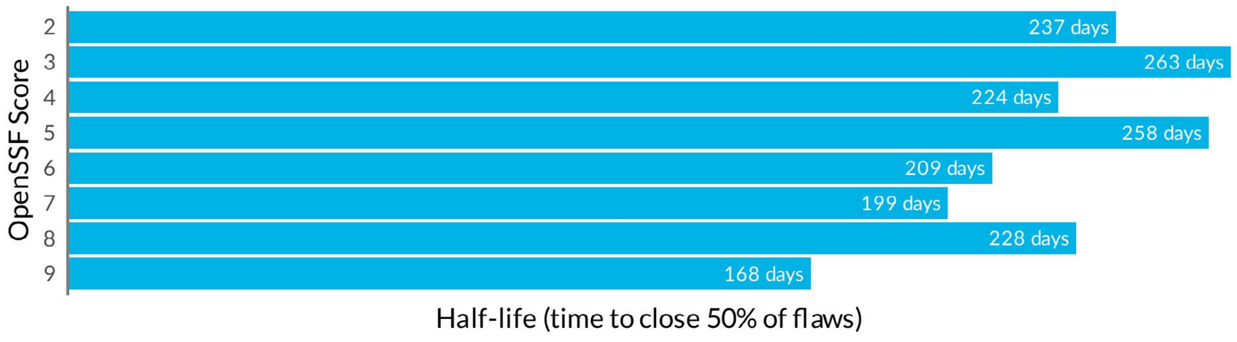


Figure 33: Number of libraries per application over time (rolling geometric mean)

Figure 39 gives a view of how quickly flaws in open-source libraries of varying OpenSSF scores are addressed by the developers using the open-source libraries. The general pattern depicted here is that libraries with better scores tend to be fixed more quickly (i.e., the 69-day shorter half-life for OpenSSF score of 9 vs. 2). But there's quite a bit of inconsistency in that trend, and it certainly doesn't convey a message like "more secure libraries are always fixed much faster."

We did see a substantial difference between libraries with an OpenSSF score and those that, as best we could determine, had not been scored (see Figure 40). In other words, any score is better than no score at all. This may be due to the fact that libraries that are open source with available code and contribution histories may be more likely to update faster and be transparent about any flaws. Perhaps open-source collaboration does indeed make things more secure.

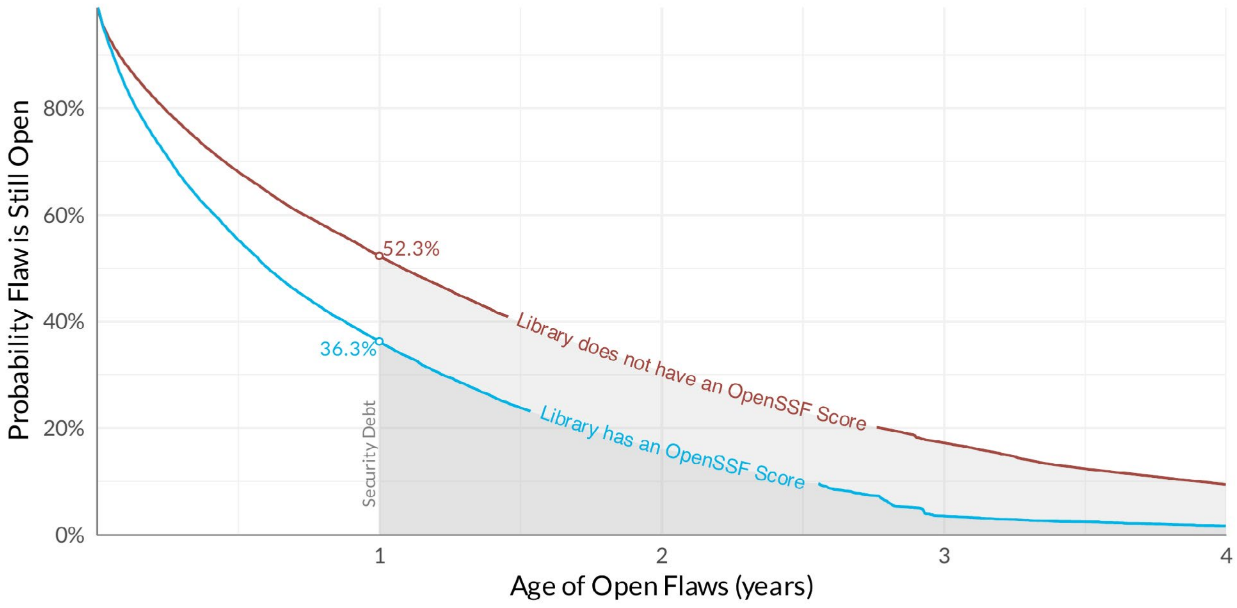


Figure 40: Fix rate of SCA flaws when the repository has an OpenSSF score and when it doesn't.

Next, we were keen to identify specific open-source libraries that are more prone to contributing to security debt in the applications of Veracode customers. After examining the 50 most popular libraries for each language, we found that some of them are far more likely to add to your backlog than others.

Figure 41 attests to this fact, showing the top eight contributors to security debt for each language. Some interesting observations can be made here. First and foremost, libraries in all languages are prone to some level of debt. JavaScript is on the low end, perhaps due to its large ecosystem of small libraries. Python and .NET sit on the higher end, with the debt ratio for several libraries exceeding the 50% mark.

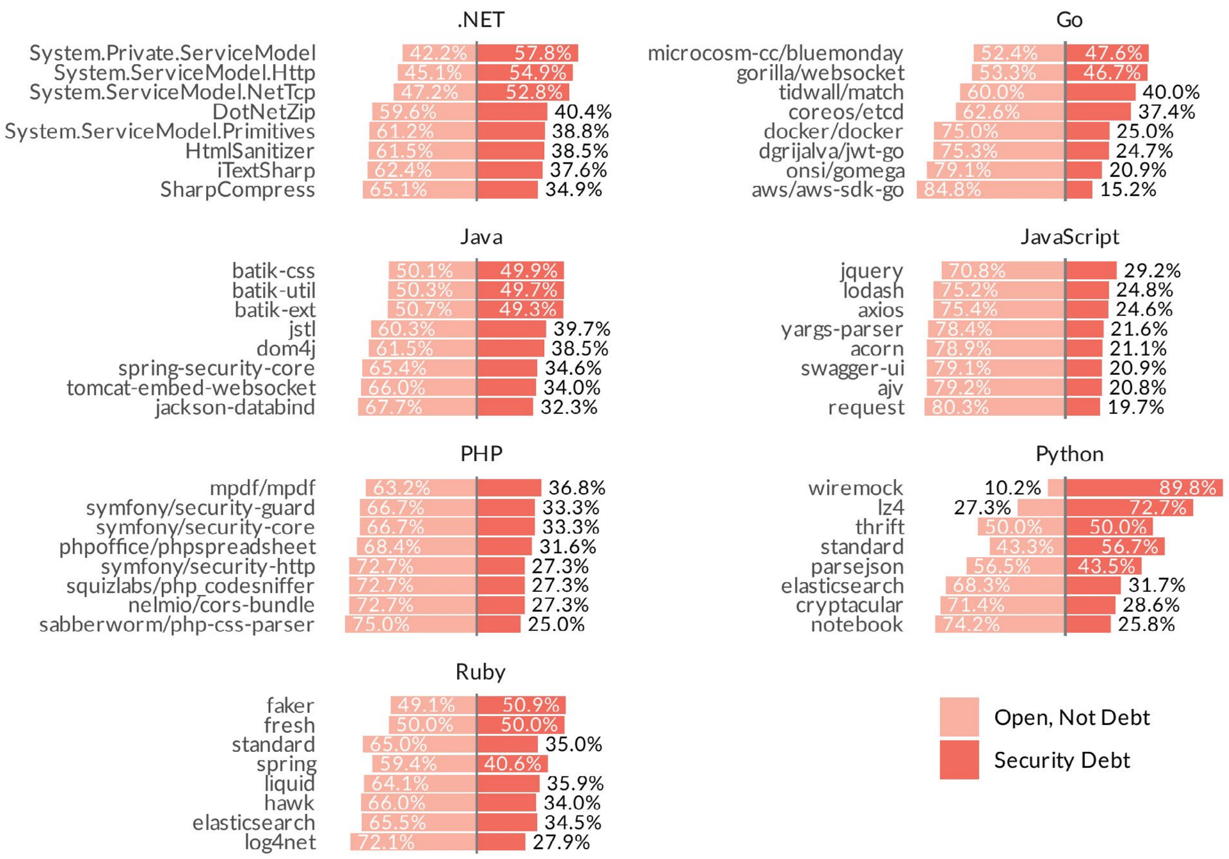


Figure 41: Third-party libraries most prone to security debt in upstream applications

What Figure 41 doesn't tell us is how common these libraries are and what share of overall security debt across applications can be attributed to them. Figure 42 examines this in a little more detail, comparing how much of a library's third-party flaws end up being debt vs how often it's used.

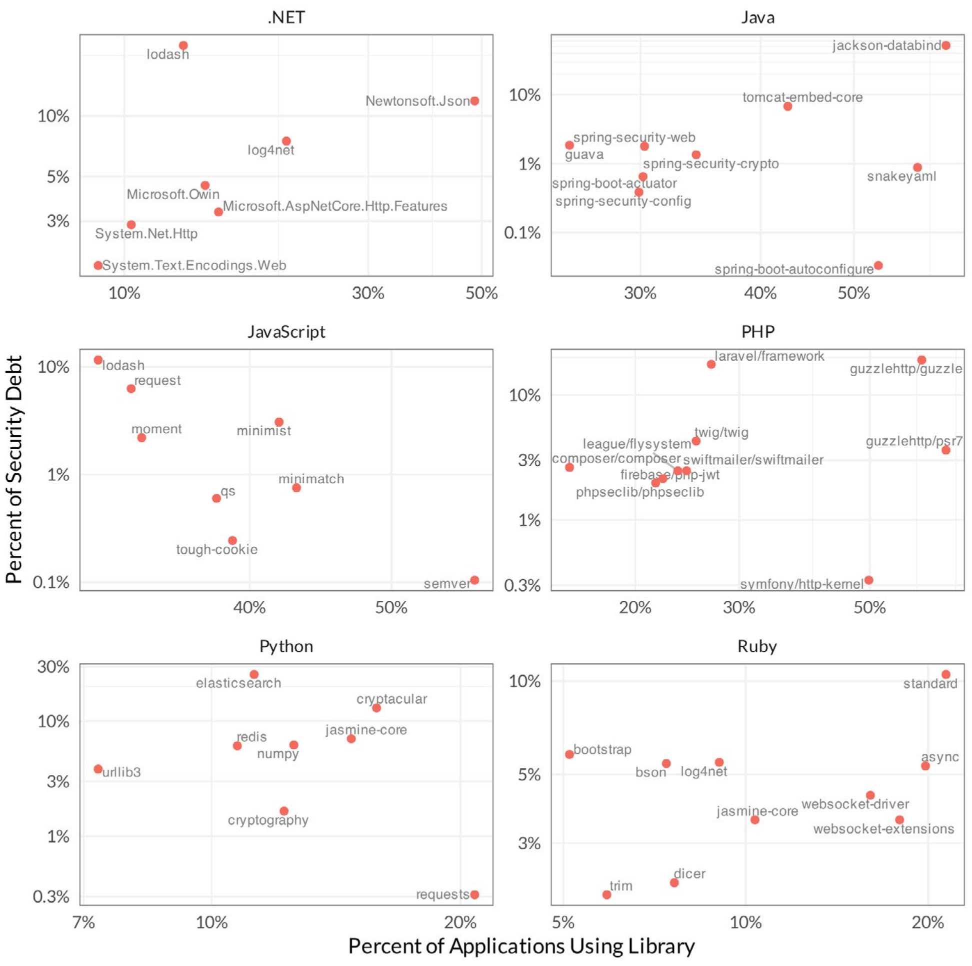


Figure 42: Comparison of library popularity and share of security debt in applications

Libraries that contribute to the largest share of security debt occupy the upper half of the charts for each language. Those in the upper-right are prevalent across applications and also accrue a lot of debt (e.g., jackson-databind for Java apps). The upper-left region is interesting because those libraries are relatively less common (though these are all bangers) but account for a disproportionate amount of all security debt (e.g., lodash for JavaScript apps).

[5]Coincidentally we also see "lodash" in .NET applications, which is a library that someone took the time to implement with similar functionality as it's JavaScript brethren. Curious that it also occupies roughly the same space—popular, but not too popular—and accounting for a large amount of security debt.

What's it all mean?

As we mentioned above, managing your software supply chain is a different beast entirely compared to managing your own code. Regardless, there are some heuristics that you can examine to hopefully guide your decision-making process around selecting third-party libraries. The primary advice is to select libraries that are open source and actively developed by a diverse community of contributors. These libraries are more likely to have security controls in place in their repository to make them more secure, and first-party developers tend to be able to address flaws in these libraries more quickly.

Reviewing What We've Learned

We threw a ton of data points and visualizations at you in this report, so thanks for sticking with us to the end. One of the challenges with dense content like this is that it can be easy to miss the forest for the trees. Sure, each tree may be worthy of individual study, but sometimes a quick stroll through the whole forest is the only way to get the lay of the land.

In that spirit, we'd like to take a quick stroll with you through key lessons we've gleaned from this analysis. Our hope is that it will grant a fresh perspective on the application security landscape and how your team can navigate it successfully.

1. Security debt is endemic

Over 70% of organizations have security debt and nearly half have critical debt. Security debt accrues at similar rates in first and third-party code and affects applications both large and small. It's not hyperbole to describe the state of software security as drowning in debt.

2. Fix capacity is constrained

"Drowning" turns out to be an appropriate word because most development teams truly are in over their heads when it comes to security debt. Only two out of ten applications show an average monthly fix rate that exceeds 10% of all security flaws. And though it's true that observed capacity often stems from choice (not to remediate) rather than inability, the end result is the same. Few teams bail fast enough to reverse the tide of debt once it starts rising.

3. Risk prioritization is key

So, if the rate of new and existing flaws will always exceed the capacity to remediate them, should AppSec teams just up on stemming the tide of security debt? No—there's still hope for rescue!

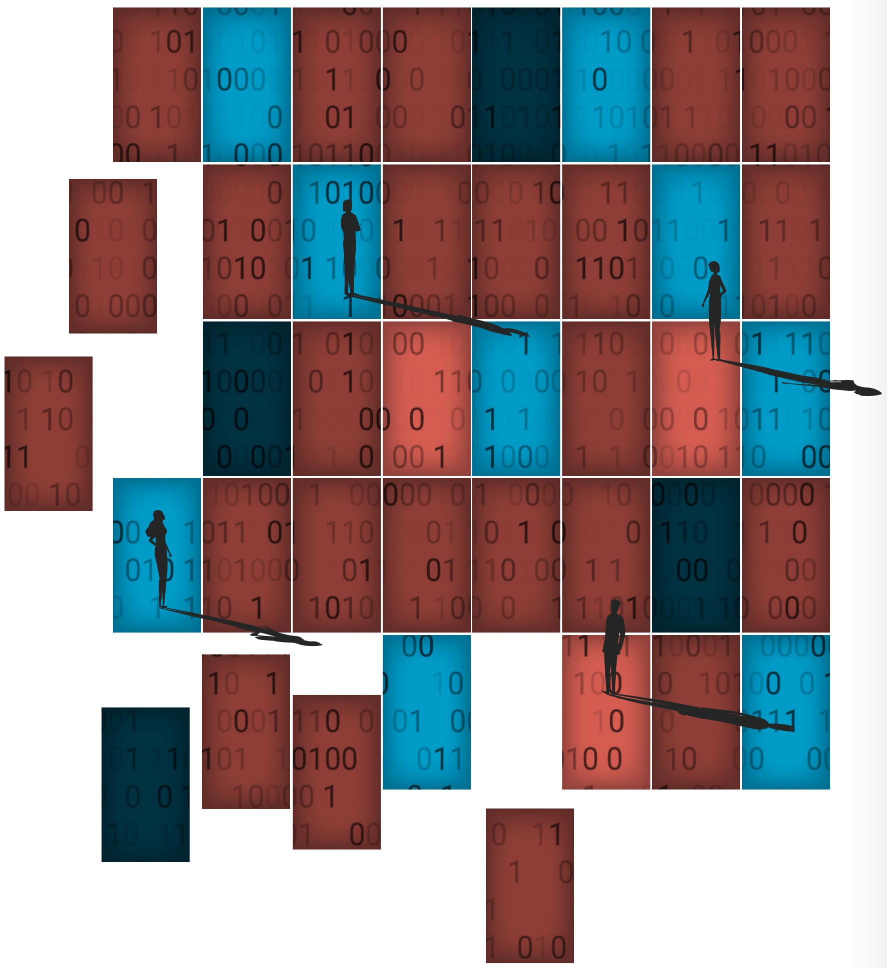
Our analysis revealed that just 3% of all flaws constitute critical security debt. This small subset of high-severity, persistent flaws represents pound-for-pound greatest risk exposure to your applications. Since fixing 3% of flaws is an achievable target for the vast majority of dev teams, why not prioritize those? Granted, this won't eliminate security debt from your applications. But it will minimize risk—and that's the main goal.

4. Can AI scale fix capacity?

Let's presume for a moment that focused development teams could stay on top of critical security debt over time. What then? Even the best teams can only fix a minority of flaws in their applications (Lesson #2). How can organizations begin addressing the long backlog of security debt as well as new flaws that emerge?

At the top of this report we discussed the impact that AI already has on code development and how this helps development teams scale. We shared research showing that untrained Large Language Models (LLMs) produce code at pretty much the same quality as that produced by humans. Expect us to follow up on this topic with a more detailed look in next year's report. We strongly believe that AI can make the dream of accelerating code fixes a reality, especially when using LLMs that have been trained on specific CWEs, to work alongside developers to suggest secure fixes at scale.

That's the kind of scaling factor that can break through existing fix capacity constraints to help eliminate all security debt—not just the riskiest subset. As a bonus, AI-augmented fixes can free up developers to spend more time creating value rather than fixing flaws. Less risk, more reward—everyone wins. How often do we get to say that in AppSec?



Methodology

The data represents large and small companies, commercial software suppliers, software outsourcers, and open-source projects. In most analyses, an application was counted only once, even if it was submitted multiple times as vulnerabilities were remediated and new versions uploaded. For software composition analysis, each application is examined for third-party library information and dependencies. These are generally collected through the application's build system. Any library dependencies are checked against a database of known flaws.

The report contains findings about applications that were subjected to static analysis, dynamic analysis, software composition analysis, and/or manual penetration testing through Veracode's cloud-based platform. The report considers anonymized Veracode customer data as well as information that was calculated or derived in the course of Veracode's analysis.

This research draws from the following:

1,007,133
applications across all scan types

1,553,022
dynamic analysis scans

11,429,365
static analysis scans

All those scans produced:

96.0 million
raw static findings

4.0 million
raw dynamic findings

12.2 million
raw software composition analysis findings

[6]Here we mean open-source projects who use Veracode tools on applications in the same way closed-source developers do. This is distinct from the software composition analysis presented in the report.

A Note on Mass Closures

While preparing the data for our analysis, we noticed several large single-day closure events. While it's not strange for a scan to discover that dozens, or even hundreds, of findings have been fixed (50 percent of scans closed fewer than three findings; 75 percent closed fewer than eight), we did find it strange to see some applications closing thousands of findings in a single scan. Upon further exploration, we found many of these to be invalid. These large collections of flaws were both added and removed in single scans: Developers would scan entire filesystems, invalid branches, or previous branches, and when they would rescan the valid code, every finding not found again would be marked as "fixed."

These mistakes had a large effect:
The top 0.02-percenters, or about 1 in every 6650 scans (0.02%) accounted for almost a quarter (24.6%) of all the closed findings

These "mass closure" events have significant effects on measuring flaw persistence and time to remediation and were ultimately excluded from the analysis.