VOLUME 11

State of Software Security

VERACODE

Contents

SECTION ONE

Executive Summary

"Every company is a software company."[1]

Whether you agree with that statement or not, it's becoming clear that software permeates practically every facet of our lives, even in areas we don't expect.

Over the past 11 years, we have explored the challenges in secure application development against the backdrop of new threats and evolving expectations in our annual *State of Software Security* report. For the 11th report, our focus is to look ahead and identify how developers can continue along their software development journey to make applications better and more secure.

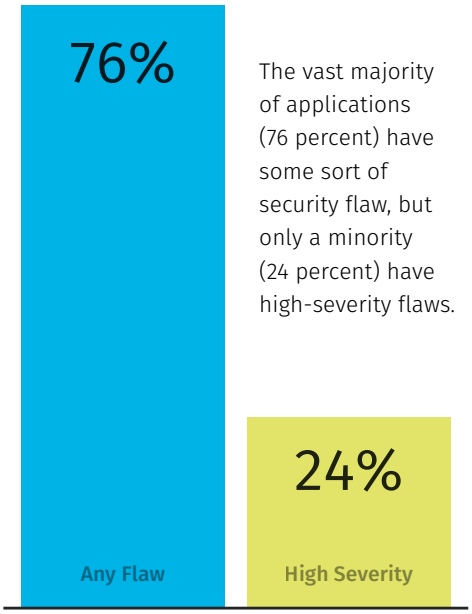
We continue our collaboration with the data scientists at the Cyentia Institute to plumb the dataset to find the untold stories of secure software development. In Volume 10, we studied scan results from over 85,000 applications. In this report, we looked at data from over 130,000 active applications.

[1] We had a tough time attributing the first appearance of this statement, but a likely candidate is: *Kirkpatrick, David. "Now Every Company Is a Software Company." FORBES 188.11 (2011): 98-+*

The State of Software Security at a glance

One rather significant change this year: we are looking at the entire history of active applications, and not just the activity associated with the application over one year. This change gives us a view of the full lifecycle of applications and enables more accurate metrics and observations.

FLAW SEVERITY



The vast majority of applications (76 percent) have some sort of security flaw, but only a minority (24 percent) have high-severity flaws.

FLAW TYPE

The most common types of flaws are much the same as previous years:

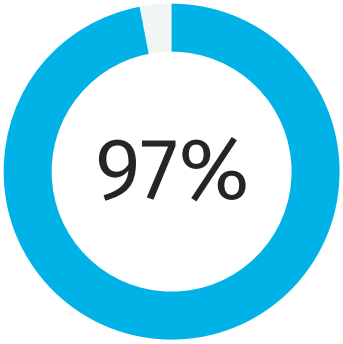
1. **Information Leakage**
2. **CRLF Injection**
3. **Cryptographic Issues**
4. **Code Quality**
5. **Credentials Management**

SQL injection and Cross-Site Scripting remain in the top 10.

SCANNING TYPE

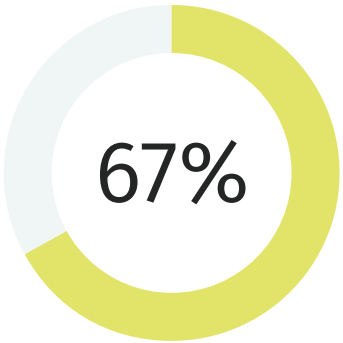
Dynamic scans
+ Static analysis
= THE COMBINATION CLOSES FLAWS FASTER

While many teams focus on static analysis, dynamic scanning can uncover types of flaws that might be hard for static analysis to find. And even though adding dynamic application security testing (DAST) will cause more flaws to be discovered, teams that combine dynamic scans with static scans end up closing more flaws faster.



OPEN SOURCE

Open source libraries can be a significant cause for concern. For example, 97 percent of the typical Java application is made up of open source libraries.



FLAW REDUCTION

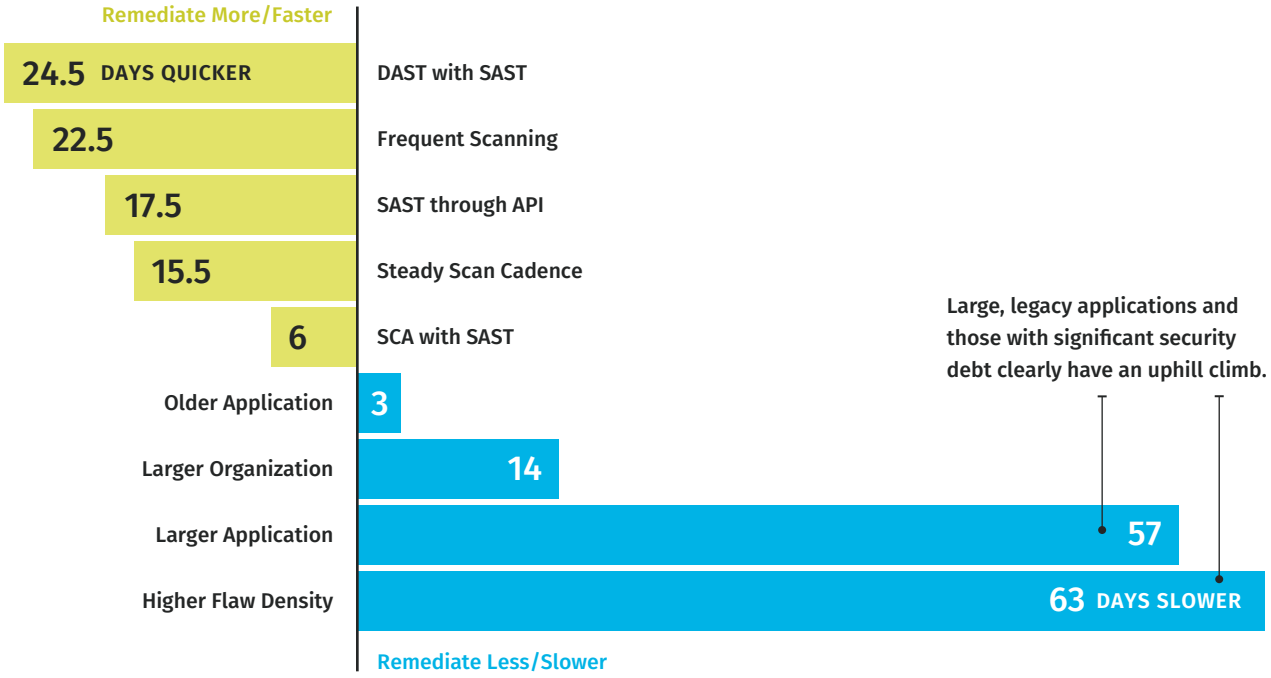
Two-thirds of applications are either maintaining or reducing the total amount of observed security flaws between their first and last scan.

Nature vs. nurture

This year, we researched how significant factors can contribute to (or detract from) closing more flaws and closing them faster.

We found there are some factors that teams have a lot of control over, and those they have very little control over — we're thinking of them as "nature vs. nurture." On one side, the **"nature"** side, we looked at factors developers have very little control over — size of the application and organization, security debt,[2] and others. On the other side, the **"nurture"** side, we looked at factors that developers have direct control over, such as scanning frequency and cadence and scanning via API.

WE PRESENT HOW EACH OF THESE FACTORS AFFECT THE HALF-LIFE (TIME TO CLOSE HALF OF THE OBSERVED FINDINGS).



EXPECTED CHANGE IN HALF-LIFE

The goal of software security isn't to write applications perfectly the first time, but to remediate the flaws in a comprehensive and timely manner. We know that it is easier to find and fix issues in applications that have less coding baggage — small application size, using modern languages and frameworks — but even with the "baggage," development teams that use secure coding practices, such as frequently scanning for flaws, integrating and automating security checks, and taking a broader look at the application's health, are more likely to have better success with their secure software development efforts.

[2] Security debt actually straddles both nature and nurture. Developers may inherit debt (nature), but it is a choice whether to accumulate it or pay it down (nurture).

SECTION TWO

Current State of Software Security

"This one goes up to 11." NIGEL TUFNEL, SPINAL TAP

Last year, Veracode celebrated 10 editions of the *State of Software Security* report. This year, we figured we might as well turn it up to 11. Over that decade++, the *State of Software Security* report has grown as software security has grown. Veracode has seen exponential growth in applications scanned this year compared to the first edition in 2009 (over 130,000 applications this year). But the number of applications isn't the only thing that's grown in 11 years. New languages and frameworks have appeared, and old standbys have risen, fallen, and risen again. Development practices have evolved. New threats and pitfalls rear their ugly heads. This report has always kept pace with the shifting sands of secure application development, and this year is no different.

This year, we are also expanding the scope of the data we are analyzing. In previous volumes, we looked at the active development of applications in a one-year time frame. This year, we are going back in time a bit further, and looking at the complete history of applications that were actively developed in the past year. So we'll get a fuller view of the origin story of an application, along with all its flaws.

With Volume 10, we spent some time looking at how much things had changed in the decade spanning Volume 1 to Volume 10. With Volume 11, we are going to look forward and consider the direction software development is headed. We are not trying to decide if we are doing better or worse than before, but looking at what kind of impact the decisions developers make have on software security.

We asked some of the same questions: how common are application flaws? Which flaws are more common? But we also dug deeper in some areas than we have in the past, such as examining third-party libraries in applications. It may not make sense to directly compare this report with previous volumes because of the underlying differences in the data and findings, but there is plenty of insight that developers and application security teams can use to make decisions on how to improve their applications.

How common are application flaws?

Let's begin with a simple snapshot of the most recent scans of applications, and ask "What percentage of applications have some sort of security flaw?"

We've tracked flaw prevalence, or the proportion of applications with at least one flaw, since Volume 1 of the report. We aren't talking about how "bad" the application is or how many flaws it has, but just whether it has at least one problem that could be fixed. In Volume 1, we found that 72 percent of the applications had at least one flaw. In this report, we find that at least 76 percent of the applications have at least one flaw in the latest scan run by customers.

When we look at the last 10 years for context, we can see that this year's report falls within the range between Volume 1's 72 percent and Volume 10's 83 percent. The slight increase from Volume 1 could be explained partially by the fact that more applications across more languages are being scanned, and the downward shift from Volume 10 could be explained by the fact that different types of scanning capabilities (static scanning, dynamic analysis, and software composition analysis) are included in this year's report. In short, we can't look at this year's results on this question to definitely say that things are better or worse.

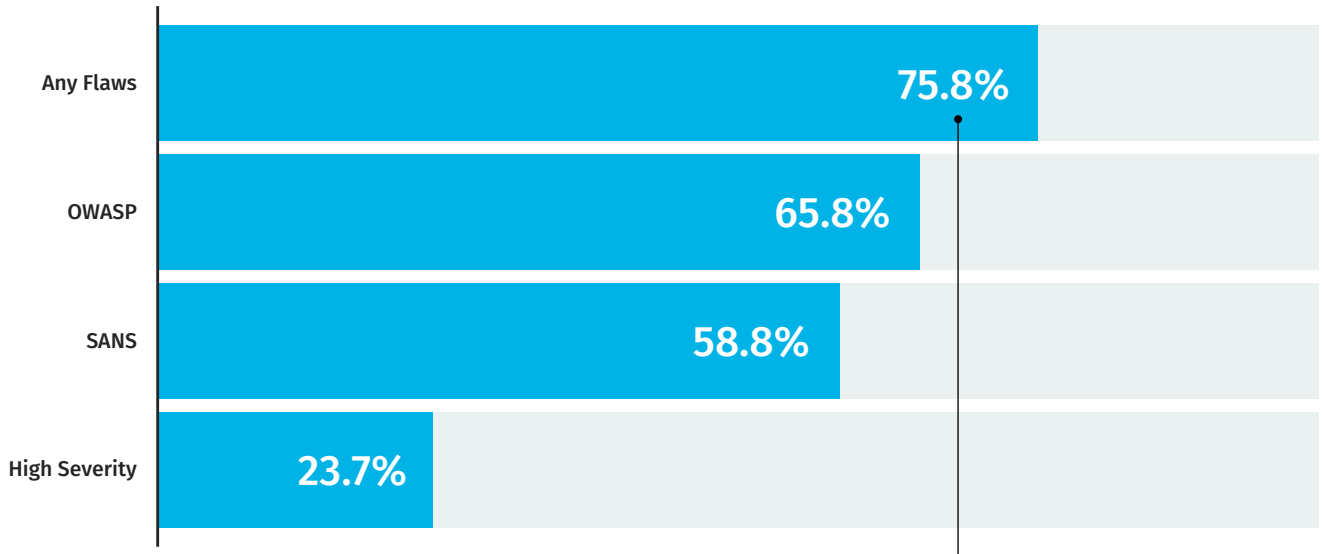


Figure 1: Percent of applications with various flaw types

Applications with at least one flaw:

VOLUME 1	VOLUME 10	VOLUME 11
72%	83%	76%

THE GOOD NEWS

It appears we are moving in the right direction when we consider the severity of the flaw.

There are fewer applications with severe flaws than ordinary run-of-the-mill flaws.

The revelation that most applications have some form of flaw should not be earth-shattering to anyone reading this report. Even so, we want to be clear that having a flaw in the application is just part of the story. We know that developers treat different types of flaws differently. Some flaws are fixed quickly, while some are considered less severe and can be moved to the back burner. It's instructive to compare applications based on how many have severe[3] flaws.

The good news is that it appears we are moving in the right direction when we consider the severity of the flaw: There are fewer applications with severe flaws than ordinary run-of-the-mill flaws. Sixty-six percent of applications have at least one flaw that appears on the OWASP Top 10, and 59 percent of applications have at least one flaw that appears on SANS 25. After the most recent scan, 24 percent contain high-severity flaws (those rated by Veracode as level 4 or 5), which is a slight increase from the previous report's 20 percent, but still within range of past years' results.

A message that we've previously shared, but it bears repeating: This is a good sign. Most applications have flaws, but not all flaws are catastrophic, and the more severe the flaws are, the more likely it is that any particular application will be free of them. A little over three-quarters of the applications may have at least one flaw, but most of them aren't the critical issues that pose serious risks to the application.

We analyze the types of flaws discovered later in the report.

[3] We take several different approaches to viewing the severity of a flaw. The OWASP Top 10 (2017) lists the most common critical flaws in web applications, and SANS 25 (recently renamed to CWE/SANS Top 25) lists common critical flaws found in modern software development. Lastly, we assign our own severity rating (a scale of 1 to 5) based on the flaw type and language. Developers can adjust that severity rating manually, since they are the ones with the context on how a flaw would impact their application.

How flawed are the applications?

Knowing the overall percentage of applications with flaws is a good, time-tested metric, but as we noted earlier, it tells only part of the story.

We want to understand the extent of the problem, but we can't just count how many flaws there are and compare the numbers. It's not fair to compare the number of flaws in a massive enterprise desktop application to the number of flaws in a tiny microservice. We account for different application sizes by counting flaws per megabyte in each application. This measure of flaw density allows us to make an apples-to-apples comparison of applications of different sizes.[4]

In the figure below, we examine flaw density across all applications that have one or more flaws. Each point represents 0.1 percent of applications placed horizontally based on its flaw density. The 1,000 points avoid each other by spacing themselves out vertically. The resulting shape gives us an idea of the relative frequency of different densities.

With flaw density, we observe a trend similar to what we saw in applications with flaws. Flaw density is lower when we focus on high-severity flaws. Figure 2 tells us that applications have problems that need to be fixed, but most of them are not riddled with catastrophic issues.

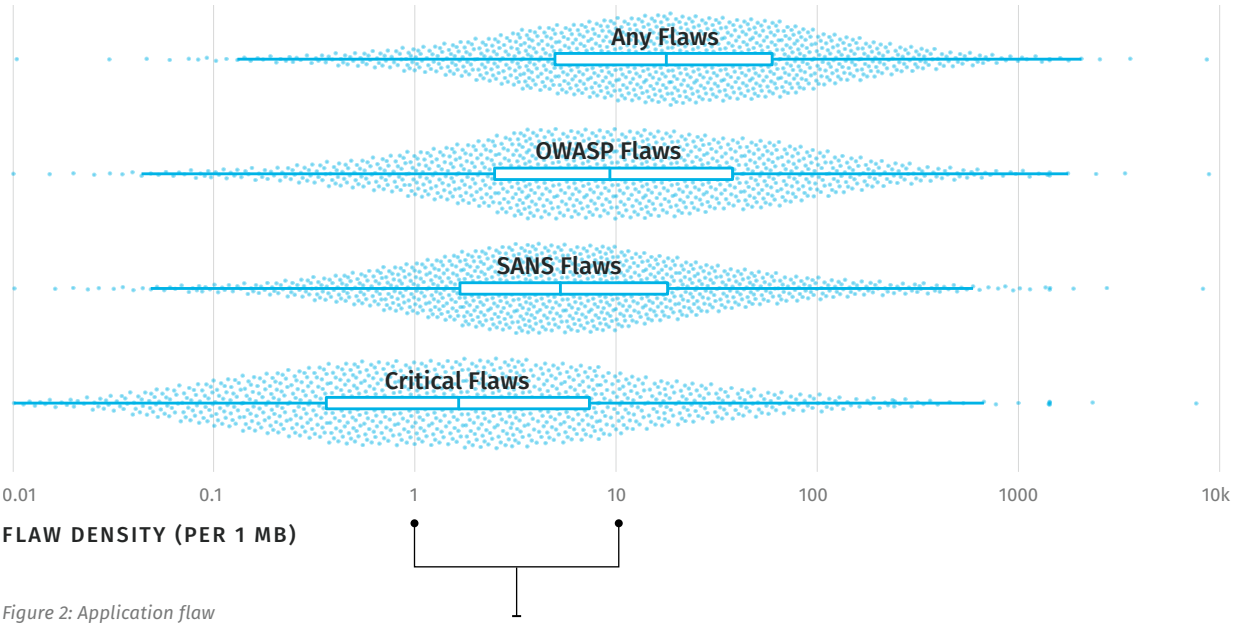


Figure 2: Application flaw density for various measures of flaw severity

We use the log scale for this figure's horizontal axis, because of the exponential growth in flaw density in applications. Each step along the axis represents an order of magnitude change. So moving from 1 to 10 or from 10 to 100 means the flaw density has increased 10 times, not just that there are 10 more.

[4] We understand that this is not a perfect apples-to-apples comparison for all applications. For example, different languages are more or less verbose when producing semantically identical code.

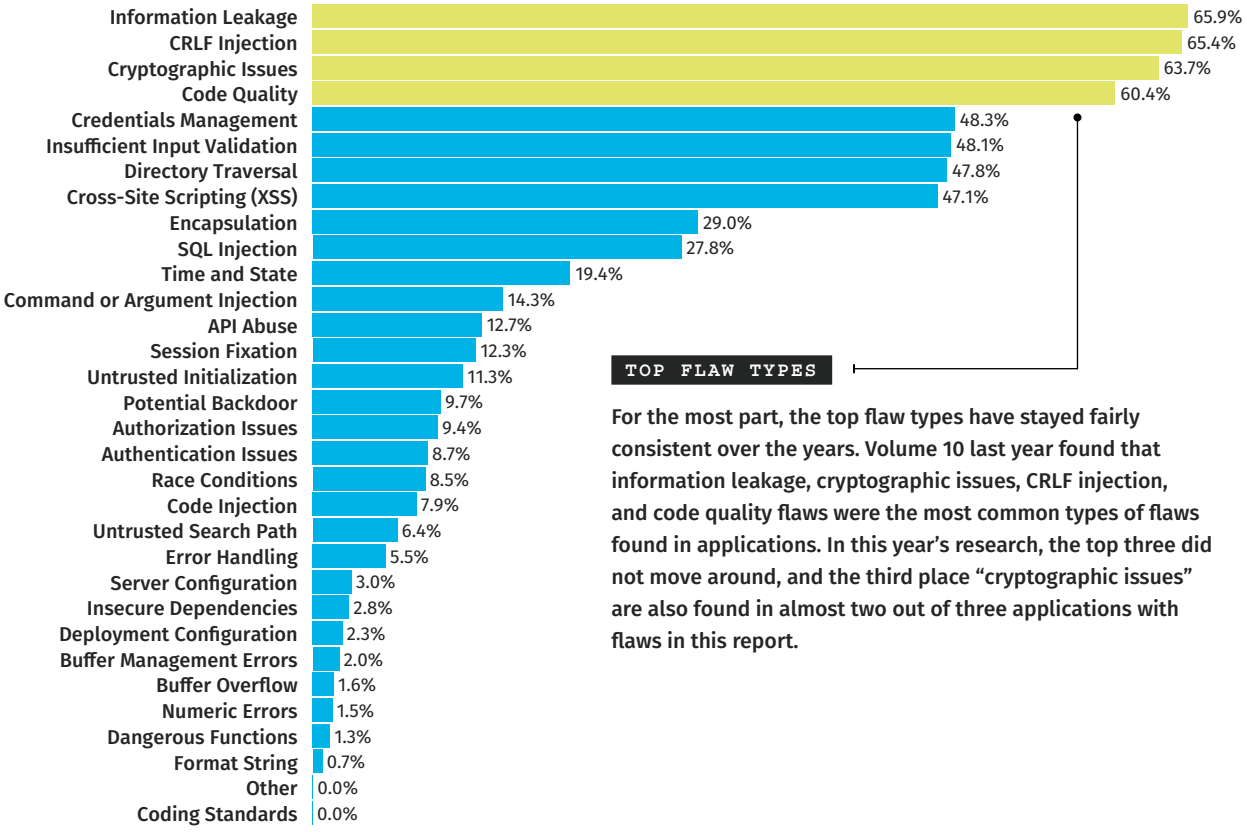
Which flaws are more common?

So far, we have used two metrics to measure the security of an application: "does the application have a flaw?" and flaw density.

Before we can even start thinking about application security and how to fix the flaws, we need to understand the myriad different types those could be. There are a wide variety of frameworks available for categorizing and organizing different types of software flaws. We've already used two (OWASP Top 10 and SANS 25) to break out by severity, but we would be remiss if we didn't dive deeper. These security frameworks and types can guide application security teams into making different decisions about how to proceed with fixing and addressing flaws.

One of the most comprehensive frameworks is Common Weakness Enumeration (CWE). The CWE framework organizes hundreds of possible flaws into vast flexible hierarchies. Indeed, our two previously mentioned measures of severity, OWASP Top 10 and SANS 25, can be viewed as a more manageable subset of the elephantine CWE.

Developers and security teams rely on these lists to figure out which flaws are considered to be highest risk and to prioritize getting them fixed. Injection flaws make up the first item in the OWASP Top 10 Web Application Security Risks, and with good reason, as our chart shows. CRLF injection was found in more than 65 percent of applications with a flaw, and SQL injection was among the top 10 list of most common flaws found.



For the most part, the top flaw types have stayed fairly consistent over the years. Volume 10 last year found that information leakage, cryptographic issues, CRLF injection, and code quality flaws were the most common types of flaws found in applications. In this year's research, the top three did not move around, and the third place "cryptographic issues" are also found in almost two out of three applications with flaws in this report.

Figure 3: Percentage of applications with specific CWE types

CRYPTOGRAPHIC ISSUES

As developers are increasingly tasked to protect data as they move in transit or in storage, there are opportunities to make mistakes in how they handle cryptography. Cryptographic issues include a variety of weak password mechanisms, weak pseudorandom number generators, and generally bad cryptography implementations — many of which are the result of using outdated cryptographic libraries, or trying to roll their own.[5] Implementing cryptography incorrectly can be just as problematic — if not more — for the application than not having any cryptography at all.

CODE QUALITY

Code quality is a tricky category, since it refers to weaknesses that indicate the application has not been carefully developed or maintained, and does not directly introduce a vulnerability in the application. Code quality is an issue because it causes the application to behave unpredictably, and that erraticness can be abused.

UNCOMMON FLAWS

What is heartening is that flaws that we might think of as particularly damaging are also relatively uncommon. Less than 5 percent of applications have the types of flaws (buffer management, buffer overflow, code injection, etc.) we could expect to be abused and lead to remote code execution or other problematic results. Part of that is because many modern languages and frameworks have built-in capabilities to address whole classes of flaws. The shift away from C++ in newer applications means fewer buffer management flaws, broadly speaking. Using higher-level languages (or language frameworks) and standardized libraries makes it easier for developers to avoid certain types of flaws.

Turning our attention to Figure 4, we look at how flaw density and flaw prevalence are related across different CWEs. In general, we see a trend — the more prevalent a flaw type is across applications, the more densely it occurs within applications.

[5] In these crazy times, it may seem nigh impossible to come together as a community and agree on a single, universal truth. But for the good of our society (and application security), let's all agree that nobody should be rolling their own crypto.

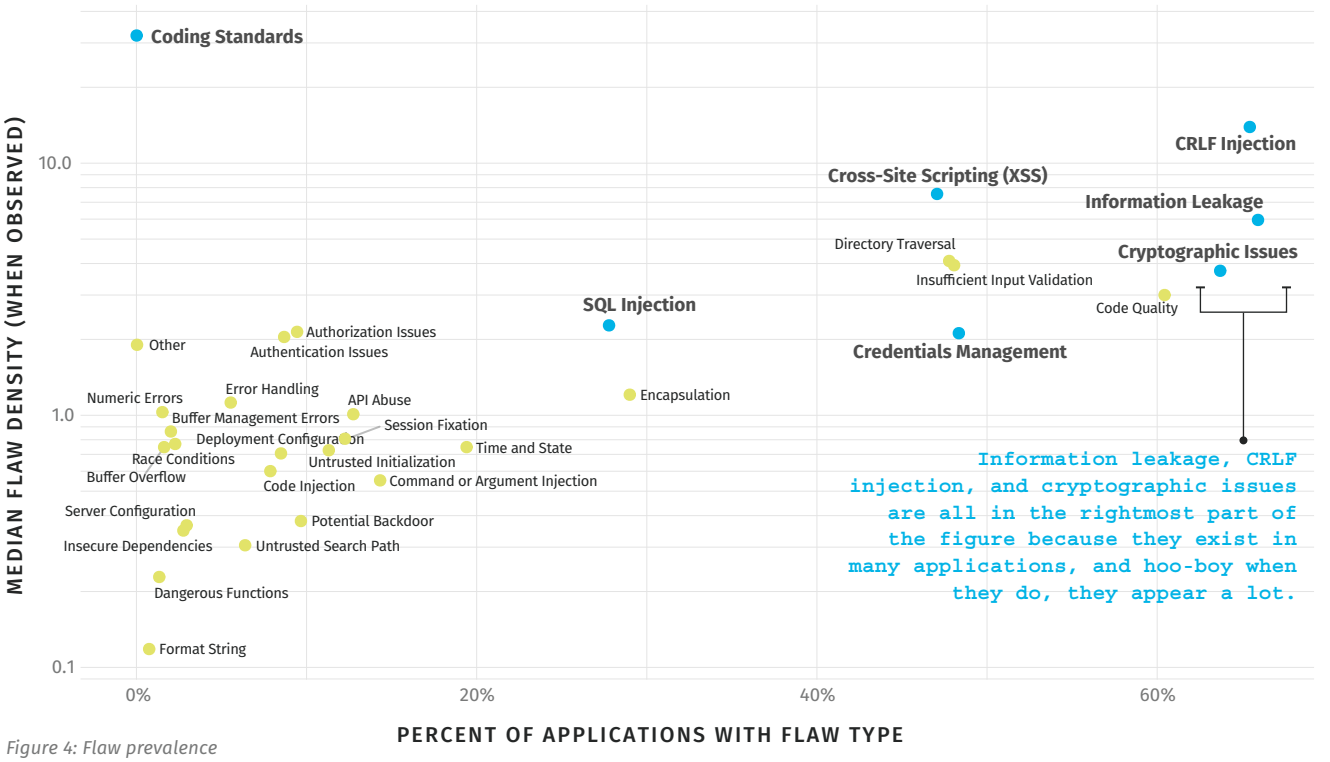


Figure 4: Flaw prevalence by median flaw density for CWE categories

CODING STANDARDS

Coding standards is a good example to look at for flaw density. Flaws related to coding standards are pretty rare overall, made evident by the fact that they show up on the leftmost part of the chart, but appear in droves (over 30 flaws per 1mb) when one exists in an application. Whether or not developers are adhering to coding standards is something that would be consistent during development, so it is logical that those issues permeate the application.

CRYPTOGRAPHIC ISSUES

In comparison, cryptographic issues are on the rightmost part of the chart as they exist in 60 percent of applications with flaws but at a density of three flaws per mb of code. Developers tend to implement cryptography in specific parts of the application, though, so we expect to see lower density as the issues are concentrated to a fewer number of places. Additionally, fixing cryptographic issues can range in complexity and effort from one-line fixes to multi-release transitions to new technologies.

CROSS-SITE SCRIPTING AND CREDENTIALS MANAGEMENT

Cross-Site Scripting and credentials management flaws appear in a little less than half of the applications, but Cross-Site Scripting issues show up in greater numbers within the application than credentials management does. Again, that is logical because credentials management flaws will likely be an issue only in parts of the code relating to authentication and authorization, while there are many opportunities for making mistakes that result in Cross-Site Scripting. Although Cross-Site Scripting flaws are generally a quicker, and usually easier, fix than credentials management flaws, they both appear in just under half of applications, suggesting the ease of fixing is just one of many factors considered when fixing flaws.

SQL INJECTION

SQL injection is another flaw type to pay attention to in this figure. As you may recall, injection flaws are among the most common flaw types, and SQL injection was among the 10 most common flaws found in applications that had at least one flaw. The flaw density for SQL injection is close to the middle, suggesting there are many areas within the application with this type of flaw. SQL injection would be an issue in any part of the application that interacts with the database, which, depending on the application, could mean a pretty significant chunk. Most (maybe all?) modern languages support parameterizing database queries, so fixing SQL injection flaws is usually a straightforward task, but not always a quick change depending on the prevalence and density we observe.

How are applications scanned?

Until now, we've been looking at the results of scanning the applications without considering how those flaws are discovered.

Academics in the software engineering community have spent decades devising multiple ways to unearth flaws from code, but for the most part, the methods fall into one of the two categories of scanning: static application security testing (SAST) and dynamic application security testing (DAST).

Static analysis relies on scrutinizing the codebase for flaws and is typically the method most developers think of when scanning applications for flaws. It excels at finding many of the common issues, such as directory traversals, Cross-Site Scripting, and various injection flaws. Dynamic analysis looks while it's running and evaluates how the application interacts with its environment. Dynamic excels at finding issues with server and deployment configuration and authentication issues.

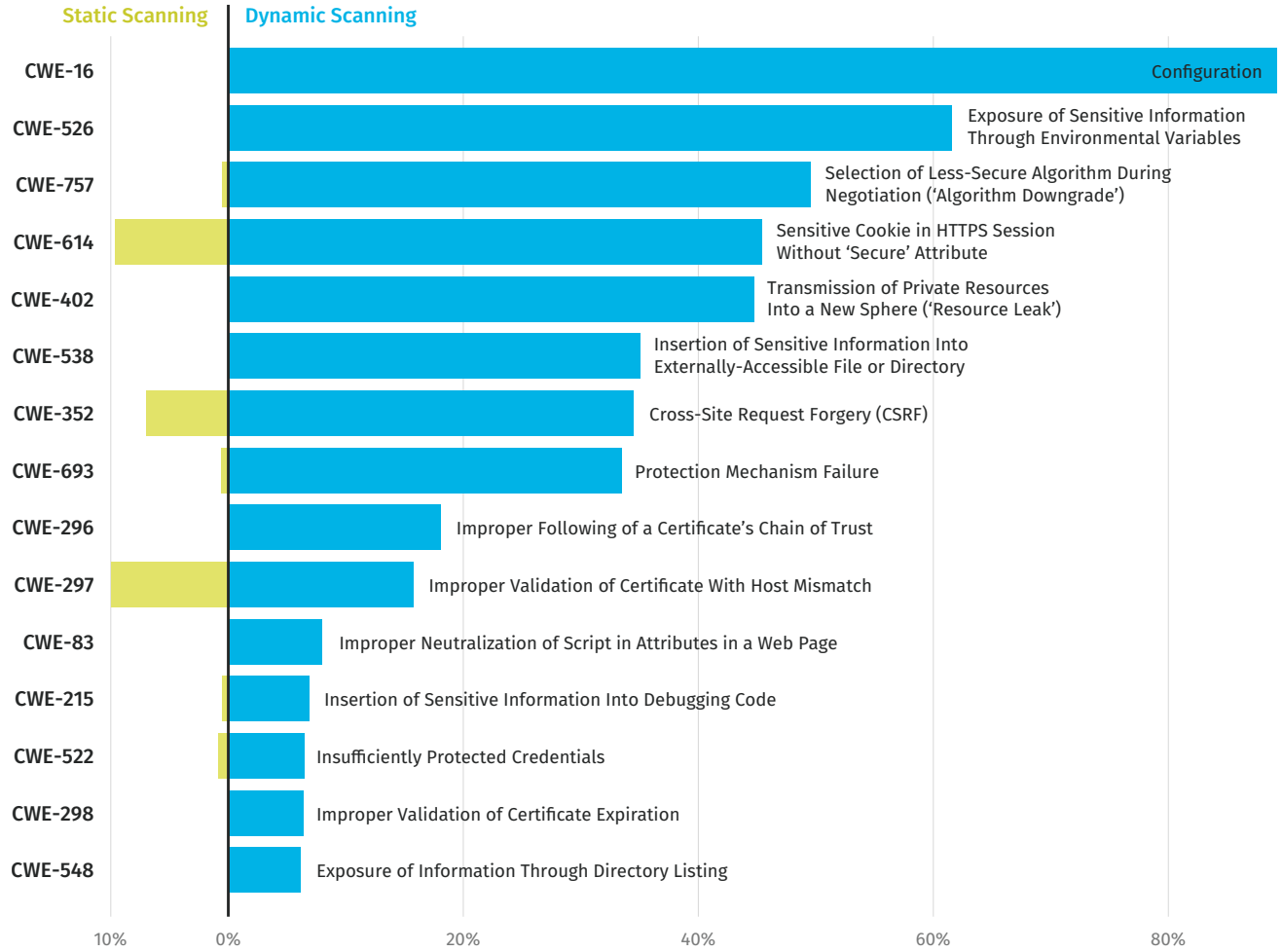


Figure 5: Percentage of applications with various CWE types in static vs. dynamic scanning

DYNAMIC SCANNING

Some flaws become more prevalent when dynamic scanning is used in conjunction with static analysis.

These are complementary methods and should not be considered as subsets of each other or replacements as they bring their own strengths to application security. Think about annual health checkups. You get bloodwork done and have a physical because they look for different things. You don't assume you are healthy on the basis of one test; you wait for all your test results.

Figure 5 highlights how much deeper the scanning goes when dynamic scanning is added to the static scanning that is already being done. Some flaws become more prevalent when dynamic scanning is used in conjunction with static analysis. Both static scanning and dynamic scanning can find issues such as using sensitive cookies in HTTPS sessions without the secure attribute, but dynamic analysis scanning is more likely to find them much more frequently than static analysis.

Dynamic scanning will uncover issues that are not part of your code, but rather in how the environment is set up. The application exposing sensitive information through environmental variables is a problem that exists in 60 percent of applications. It is a flaw that will not be uncovered if the developers are relying only on static analysis.

Focusing on only one type of scan means a whole set of potential flaws may not be discovered, leaving developers in the dark about a significant swath of issues across their application.

SECTION THREE

The Tale of Open Source Flaws

Earlier this year, we published a “spinoff” version of the SOSS focused on open source flaws. We’ll likely do that again but felt compelled to include some statistics related to open source code in this report as well. Even if developers wave a magic wand and voila! all the flaws we’ve discussed so far disappear from their own code, that doesn’t mean applications would become flaw-free. It’s never that easy in software security.

The expanding attack surface

Virtually no modern application can avoid including open source libraries that provide functionality that would be difficult or time-consuming to write from scratch.

Pulling in those components means the flaws they contain become part of your application. Let's look at some of the data specific to the security challenges of including open source libraries.

The most basic question we could ask is exactly how much of an application is composed of open source libraries. The answer can be found in Figure 6. Each dot in this chart represents 1 percent of the applications in each language, and the horizontal position of the dot indicates the percent of the application's code composed of third-party libraries.

For example, Java applications (shown at the top of Figure 6) cluster to the right, indicating that they tend to be almost all third-party code — and indeed, the typical Java application is 97 percent third-party code! However, that pattern does not emerge with other languages. JavaScript and Python applications cluster at both ends, much like a barbell — so from a pure code-volume perspective, applications tend to be mostly homegrown or composed mostly of third-party libraries. C++ and PHP cluster completely in the opposite direction of Java, indicating the codebase is mostly homegrown. Only .NET applications seem to be fairly spread out, suggesting developers tend to be a bit more flexible in how libraries are used.

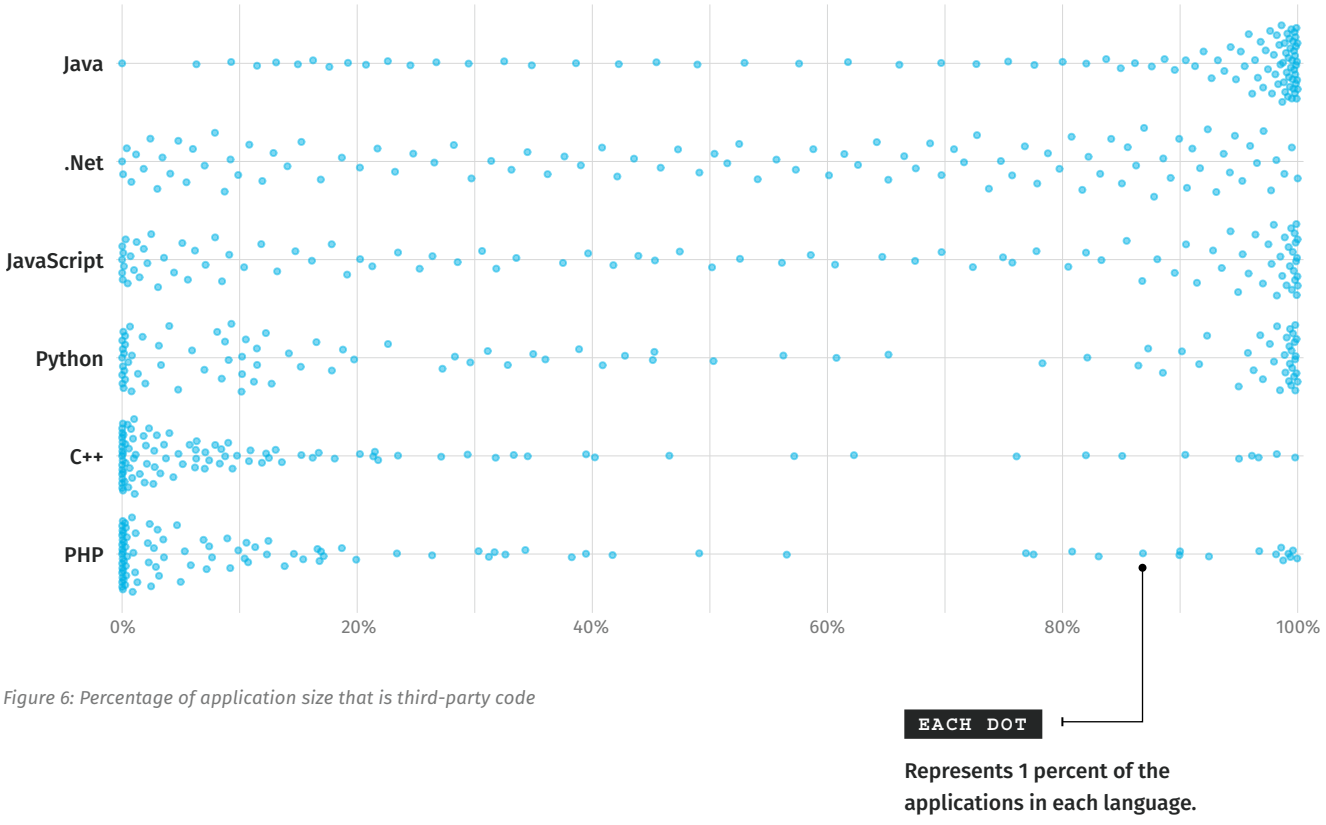


Figure 6: Percentage of application size that is third-party code

The ubiquitousness of open source libraries was evident when we released the *State of Software Security: Open Source Edition* report. However, there was something else we learned there; about seven in every 10 applications were found to have flaws in their open source libraries (on initial scan). This alone should warrant adding software composition analysis into any software security program. But we can take this one step further this time around. We looked at how many flaws were found in open source libraries and compared that to how many flaws were found in the primary application (code written in-house), and we found about three in every 10 applications have more flaws in their open source libraries than in the primary code base.

One last little insight we found here: There is almost no correlation between the flaw density in open source library flaws versus those in the primary application. This means that it's possible to have a very well buttoned-up application, yet vulnerabilities may be exposed through third-party libraries.

Figure 7: Third-party vs. first-party security flaws

KEY LESSON

Software security comes from getting the whole picture, and that means identifying and tracking the third-party open source libraries used in your applications.

SECTION FOUR

Fixing Software Security

It is inevitable that software will have flaws, so until now we've focused on understanding what it means when we say that applications have flaws. However, accepting there will be flaws does not mean there is nothing that can be done. Indeed, many companies (including Veracode!) make it their business to help developers write more secure code. Software security depends on how development and application security teams address the issues that exist in the applications. We look at the question of how applications are fixed from multiple perspectives.

What proportion of flaws are fixed?

To understand how flaws are dealt with, we look at the proportion of discovered flaws that are closed or remediated in the following figure.

In this year's report, we see that 73 percent of discovered flaws have been closed or remediated, which is a shift compared to recent years[6] (52 percent in 2018, 56 percent in 2019). From earlier, we know that the prevalence of flaws hasn't changed dramatically as compared to previous years, so we can chalk up the fact that roughly three out of four flaws are being fixed as an improvement in how flaws are being handled. We explore the reasons that may explain the improvement shortly.

Logically, we anticipate that flaws aren't fixed in the order they are found, and that there is some kind of prioritization that happens first. We see a slight preference for fixing flaws that may be considered more problematic over general ones, such as those from OWASP and SANS lists. We also see that high-severity flaws were 18 percent more likely to be addressed than general flaws. As we discussed earlier, we found high-severity (level 4 or 5) vulnerabilities in 24 percent of applications, indicating that while many applications have flaws, few of those flaws pose serious risk to the applications.

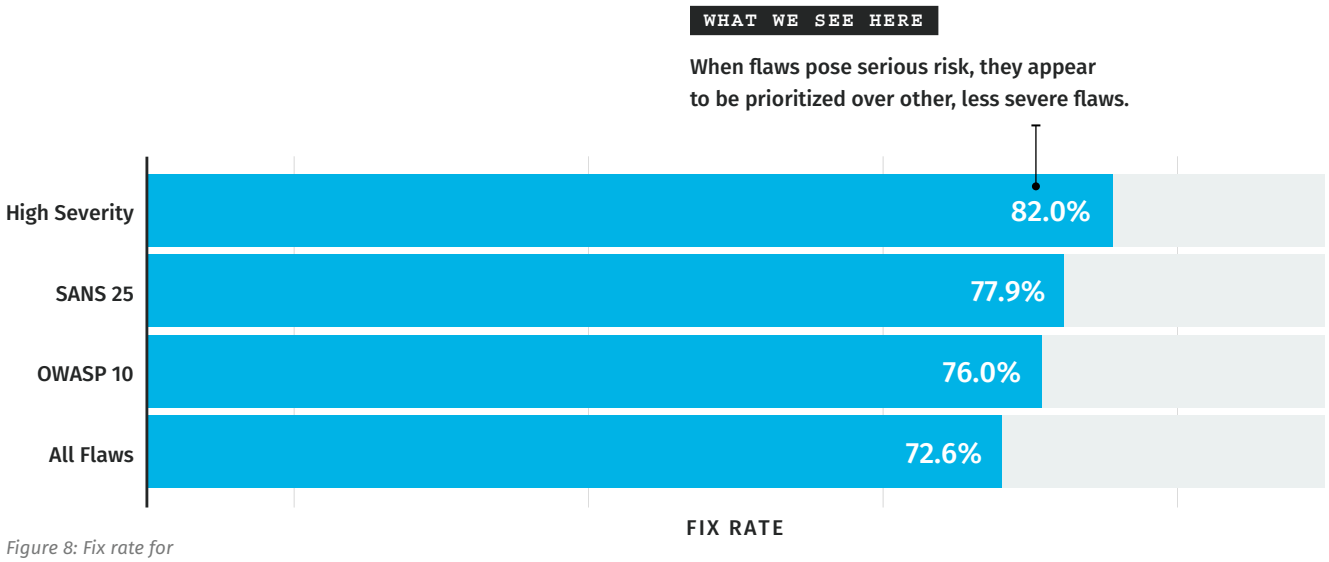


Figure 8: Fix rate for various severity types

One thing to remember about the report is that we are comparing the application's first scan results with the latest one within a 12-month period (April 2019 to March 2020). While it's heartening to know that nearly three out of four flaws are being closed, bug hunting becomes a game of whack-a-mole if new bugs are being introduced at the same pace as the fixes being made. The chart comparing flaw density between the first and last scans illustrates whether the fixes are being made faster than new ones are introduced.

[6] The increase can be explained partly by the fact that we changed how we looked at the data in this year's report. In previous years, we looked at flaws only that were active during the report period (including everything still open from before the year). However, for this report, we analyzed the full history of active applications, so the number of closed flaws reflect all the flaws, including those that were closed before the report period.

While the number of flaws in applications ebb and flow over time, for the majority of applications, the overall flaw density is decreasing over the course of development. Generally speaking, more applications reduced the flaw density, as half of the applications had fewer flaws on the latest scan than in the first scan. Flaw density was higher for 34 percent of the applications, suggesting the development teams were not prioritizing fixing flaws as they went along, but were perhaps saving them for later.

That picture gets sharper when considering the seriousness of the flaws. When looking at only high-severity flaws, roughly twice as many applications (23 percent) reduced the overall flaw density than those that increased (12 percent).

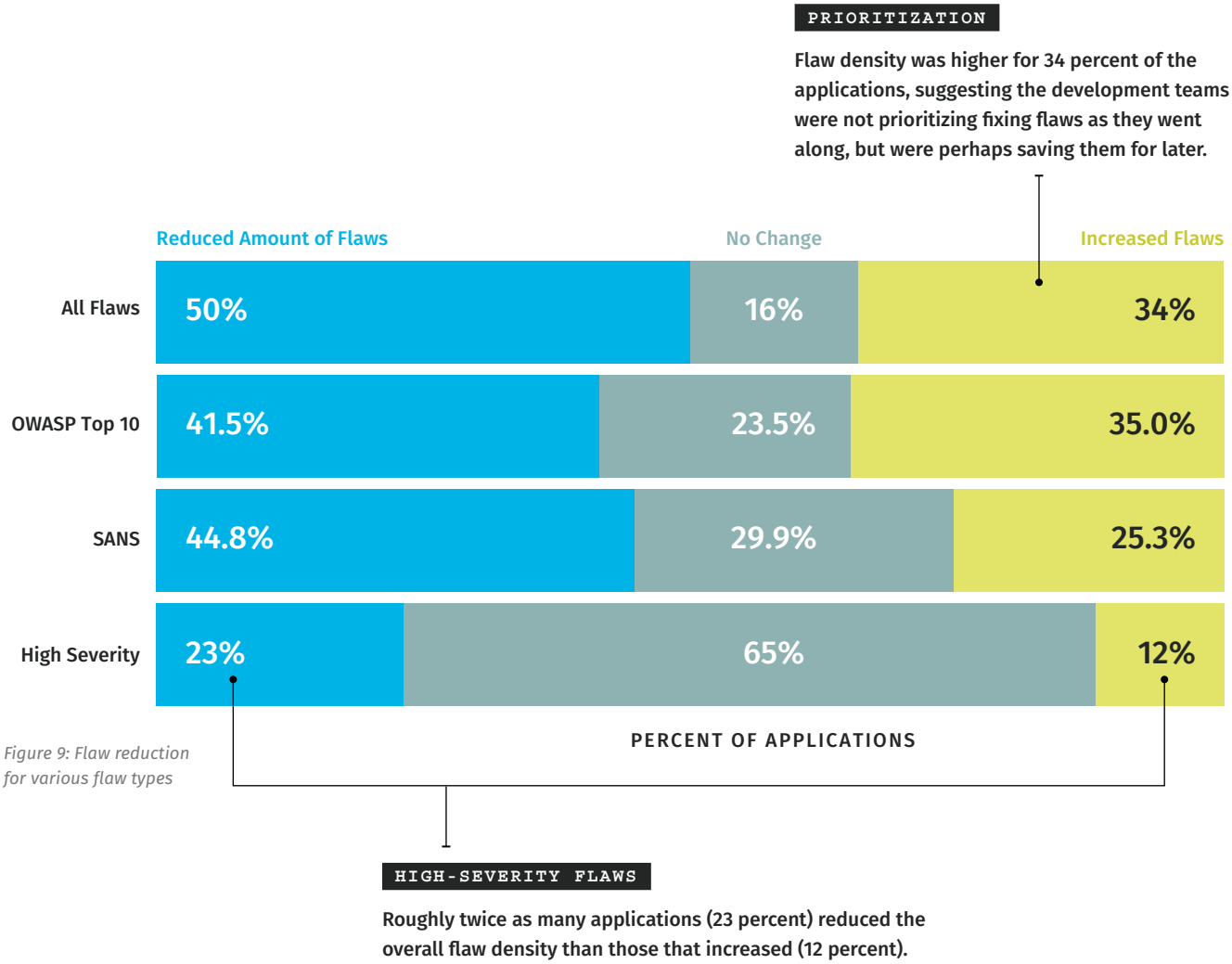


Figure 9: Flaw reduction for various flaw types

Regional differences

We wondered if there were differences in how quickly flaws were being fixed across different geographies.

While there are some variations, for the most part, developers don't change their behavior based on where they are located. Roughly three out of four flaws are being closed in the EMEA (Europe, Middle East, and Africa) region, as well as in the Americas (North America, Central America, and South America). Closer to three out of five flaws are being fixed in APAC (Asia-Pacific), although that picture is reversed when we focus on only the high-severity flaws. For high-severity flaws, EMEA and the Americas continue to keep pace with each other, at 85 percent and 82 percent, respectively, but 91 percent of high-severity flaws are being closed in APAC.

FLAW DENSITY

The overall pattern holds true across regions for flaw density, as well. For most applications, flaw density is decreasing between first and latest scan, and about a third have an increase.

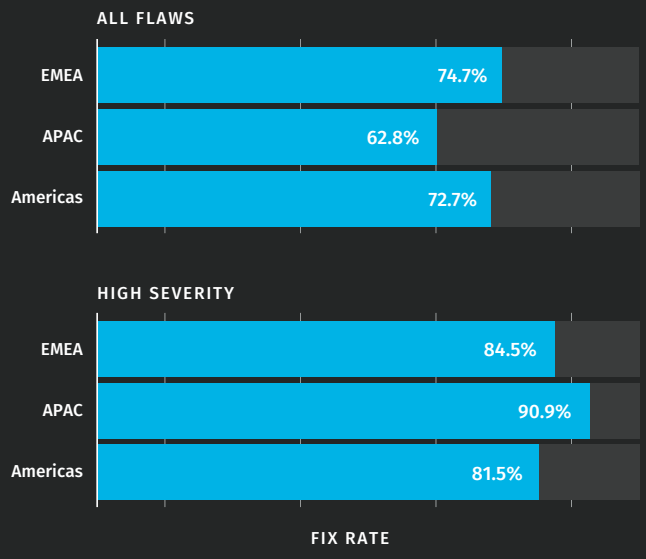


Figure 10: Flaw prevalence across regions

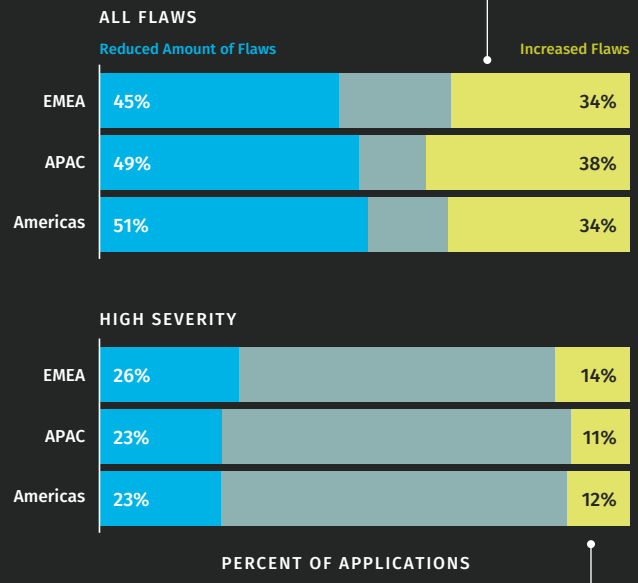


Figure 11: Fix history for various regions

FIXING FLAWS

Even when we look for regional variations, the behavior remains the same: development teams are fixing flaws, and they are prioritizing fixing the worst flaws over general ones.

How fast are flaws fixed?

We can show how applications get better over time by looking at all the flaws fixed between the first and latest scans, and counting how many days it took for the flaw to show up as fixed in the scans.

But it's trickier to measure the teams' reaction times, to establish how quickly the developers are addressing the flaws as they are discovered. With the top half of Figure 12, we know that 50 percent of the closed flaws were closed within 86 days. This is fairly consistent with other industry reports, showing that flaws tend to be fixed within the first three months of discovery.

Looking at just fixed flaws misses an essential point: many flaws are not closed, and the older the flaw, the less likely those flaws will ever be fixed. The bottom half of Figure 12 shows flaws that were still open as of the latest scan, and we can see that 50 percent of flaws have been open for at least 216 days, and counting. This is especially problematic since some of the open findings may never be closed, and so will never be factored into the metric for expected close time.

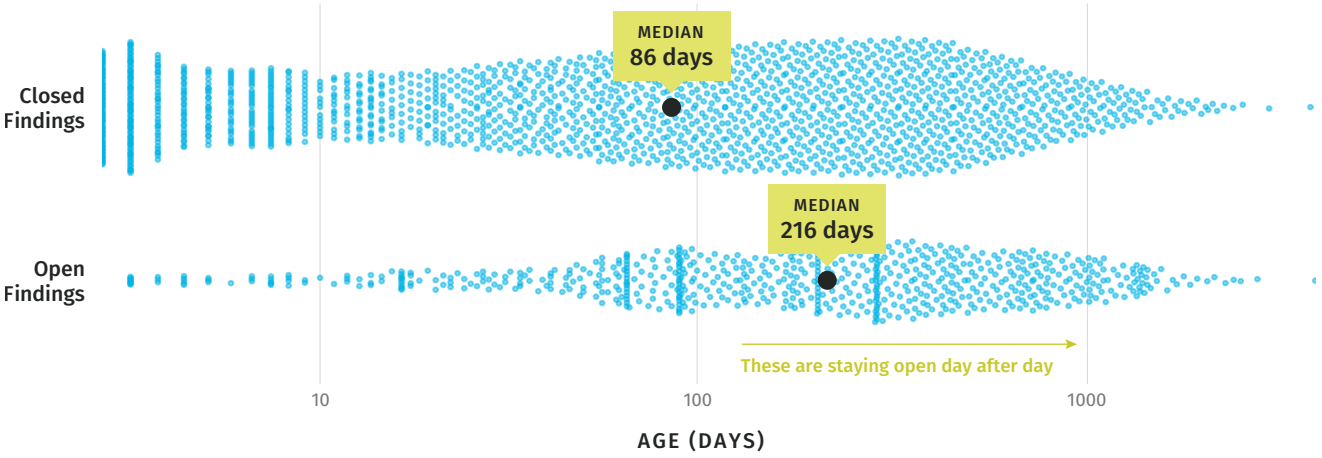


Figure 12: Median flaw closing times for applications

The median time-to-close only focuses on part of the data (the remediated flaws), and so it tells only part of the story. It tells us something about 76 percent of the flaws that were actually closed, but when a new flaw is discovered, we don't know if it will be like the 76 percent of closed flaws, or like 24 percent of flaws that remain open. Luckily, we aren't the first people to run into this, and there are better techniques[7] we can apply. When we account for both the closed and open flaws, we find it takes about 180 days (6 months) to close half of the flaws discovered. That's a far cry from the 86 days, but it paints a much more realistic picture since it leverages all the information at our disposal.

[7] We apply statistical methods collectively referred to as "survival analysis" and the label given to flaws that are still open (or events that haven't occurred yet) is "censored data" if you'd like a fun search term.

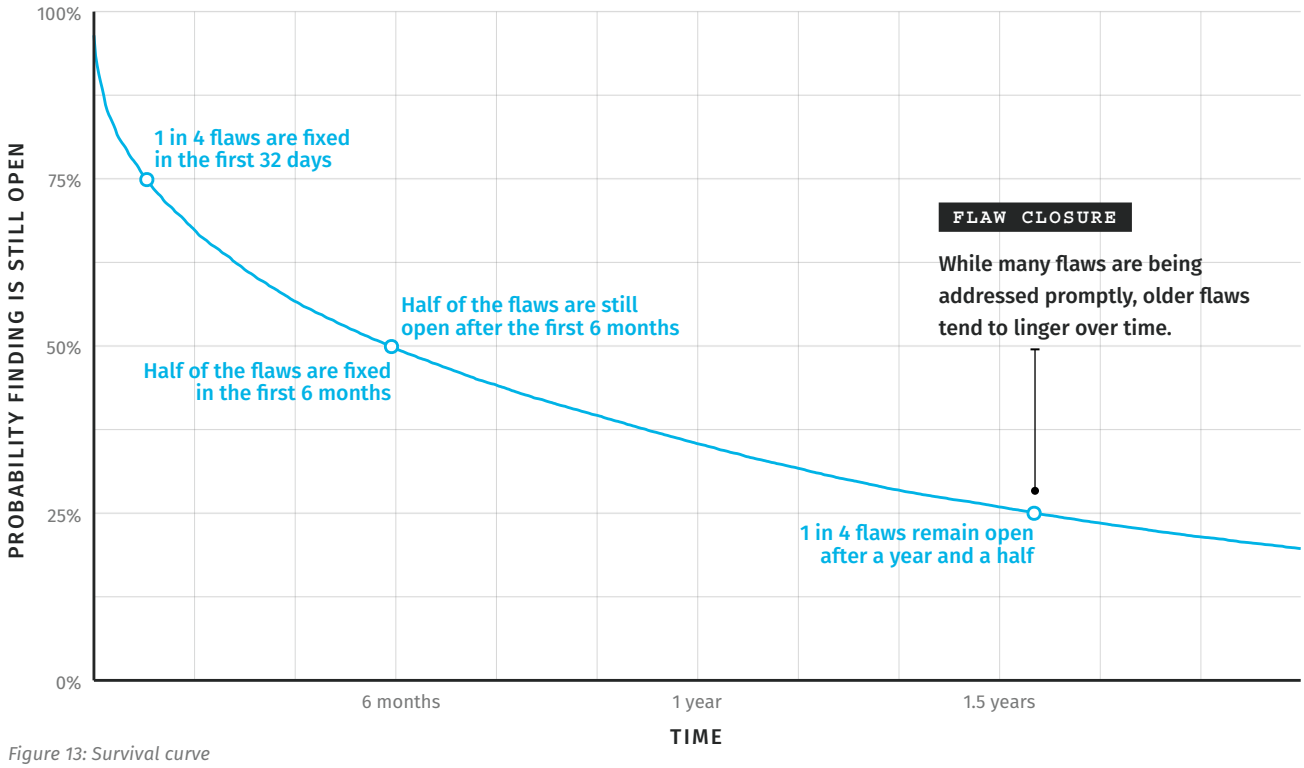


Figure 13: Survival curve of flaw closure

Figure 13 shows the full picture of the expected remediation timeline and has a few annotations calling out milestones along the remediation path.

Simply put, while many flaws are being addressed promptly, older flaws tend to linger over time. There are several reasons to explain why. The development team may be rationalizing against fixing the flaw because it hasn't caused any problems, yet, or thinking that there is no need to spend additional time on a legacy application. Another reason for the lingering flaws could be logistics. The team may not have the capacity to devote the time to fixing flaws, especially if there is more emphasis placed on developing new features rather than reducing security debt.

While Figure 13 paints the remediation timeline based on the applications we observed, we can go deeper than just speculation about what may contribute to remediating more software flaws faster. The next section digs into various attributes of applications, the actions developers can take, and what effect that has on remediation times.

Finding factors for faster fixes

It would be naive to argue that software security is just about development teams writing code and fixing flaws when they find them.

In past SOSS reports, we've observed that there are things development teams can do to improve software security, especially in regards to how quickly flaws are fixed. In v9, we focused heavily on the number of security scans and measured substantial improvements. In v10, we examined the cadence of those scans and demonstrated that more regular scanning led to better performance than irregular activity such as scanning in bursts followed by periods of no activity. Scanning applications frequently and on a regular cadence is representative of a DevSecOps approach, while timing fixes around major releases tend to be more common in teams taking a waterfall approach.

Scanning frequency and cadence are but two aspects of software development in a sea of possibilities. For example, Figure 14 depicts how long applications took on average to close 50 percent of their open flaws split out by their scan frequency. Clearly, applications that scan infrequently (less than 12 times in a year) spent about 7 months to close half their open findings, while applications that scanned at least daily (on average) reduced that time by more than a third to close 50 percent of flaws in about 2 months.

Scan frequency and the cadence of scanning are two things the developer directly controls, but there are many others. Software security depends on a combination of the applications' environment and developer practices: nature and nurture. A developer dropped into an application has little control over the maturity of the codebase, its history, or size: the application's "nature." However, how the developer chooses to "nurture" that application is well within his or her control — how often the application is scanned, the cadence with which it's scanned, what types of scanning are done, and how third-party code is managed.

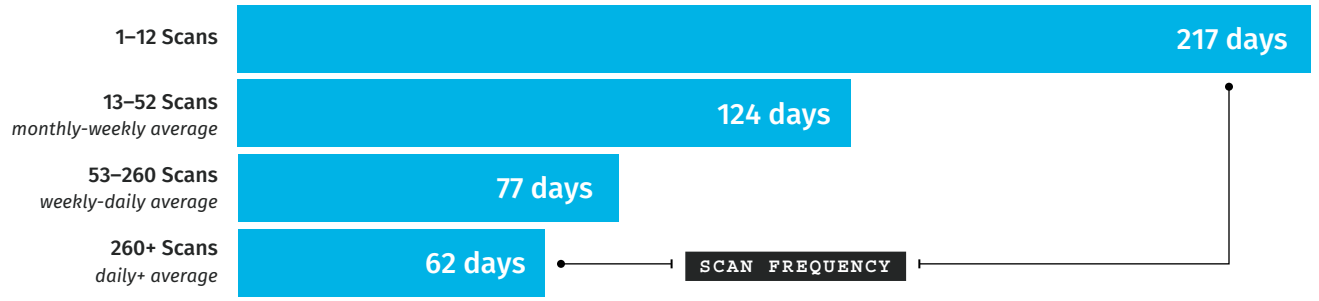


Figure 14: Time to remediate 50% of flaws based on scan frequency

Clearly, applications that scan infrequently spent about 7 months to close half their open findings, while applications that scanned at least daily reduced that time by more than a third.

We recognize that a developer inheriting a large, mature codebase that is just being maintained faces a very different set of challenges than a team that is starting out with a smaller, more focused application, but we can see that developers have some control over the security of their application.

The next set of analyses is an attempt to separate out the effects of nature vs. nurture on the remediation rates of flaws in an application. Because this is somewhat of a big undertaking, let's be precise about what we are examining. First, we look at the type of things that are, for the most part, out of developers' hands.

"NATURE" [8]

ORGANIZATION SIZE

Larger Organization

Size of the organization measured by revenue

APPLICATION AGE

Older Applications

How long an application has been using Veracode (days since the first recorded scan)

APPLICATION SIZE

Larger Applications

Size of the application measured in mb

FLAW DENSITY

High Flaw Density

Calculated as flaws per 1 mb of code, a way to think about and capture "security debt" in applications

"NURTURE"

SCAN FREQUENCY

Frequent Scanning

How many times in a year the application was scanned (with SAST)

SCAN CADENCE

Steady Scan Cadence

Measures the variation in how frequently the applications are being scanned and ranges from regular, steady scanning (typically because scanning is part of continuous integration) to bursty and sporadic scanning (followed by long periods of no scanning)

DYNAMIC ANALYSIS

DAST with SAST

The application is being scanned using dynamic analysis

SOFTWARE COMPOSITION ANALYSIS

SCA with SAST

The application's open source libraries are being scanned

API INTEGRATION

SAST through API

If the application uses the API to run the scanner, and suggests the developers are following continuous integration practices for pipeline automation

The only factor we haven't yet discussed thus far in the report is API integration. And that's on us, because what we found for applications that have integrated security testing with the API (that is, building security analysis directly into the development pipeline) is quite interesting. Not only does the integration make scanning more automated and easier, but it also negates the need for developers to remember to manually run the security scanner. More importantly, we found a significant change in the remediation rates for applications that integrate scanning into the pipeline. Incidentally, when we look at the data, we find that API usage also has an impact on scan frequency and cadence. Meaning when applications use the API, we observe applications being scanned more frequently and with a steadier cadence. Even though the three factors have some correlation, we were able to look at them separately to understand the impact each one has on the application.

[8] The size of the organization can impact the decisions developers make during the course of development, but it is very clearly not up to the developer. It is arguable whether the last two (application size and flaw density) are part of the application's nature, or the result of development. We take the perspective that these are artifacts of the history of development rather than something that the individual developer has direct control over. Similarly, application age, or how long an application has been scanned by Veracode's tools, is part of the development history and usually not the result of a single developer's decision.

In the previous section, we discussed that it wasn't enough just to see if a flaw gets closed or not because we also want to know how quickly that flaw gets closed. To that end, we built a model that accounts for both the open and closed flaws that is able to account for multiple facts and can quantify the effect of various "nature" and "nurture" factors on how quickly flaws are closed.

First, we extract from the model how each factor changes the median time to flaw remediation. We want to see which factors are likely to lead to flaws getting fixed faster, and which factors lead to slower fixes. The results are seen in Figure 15.

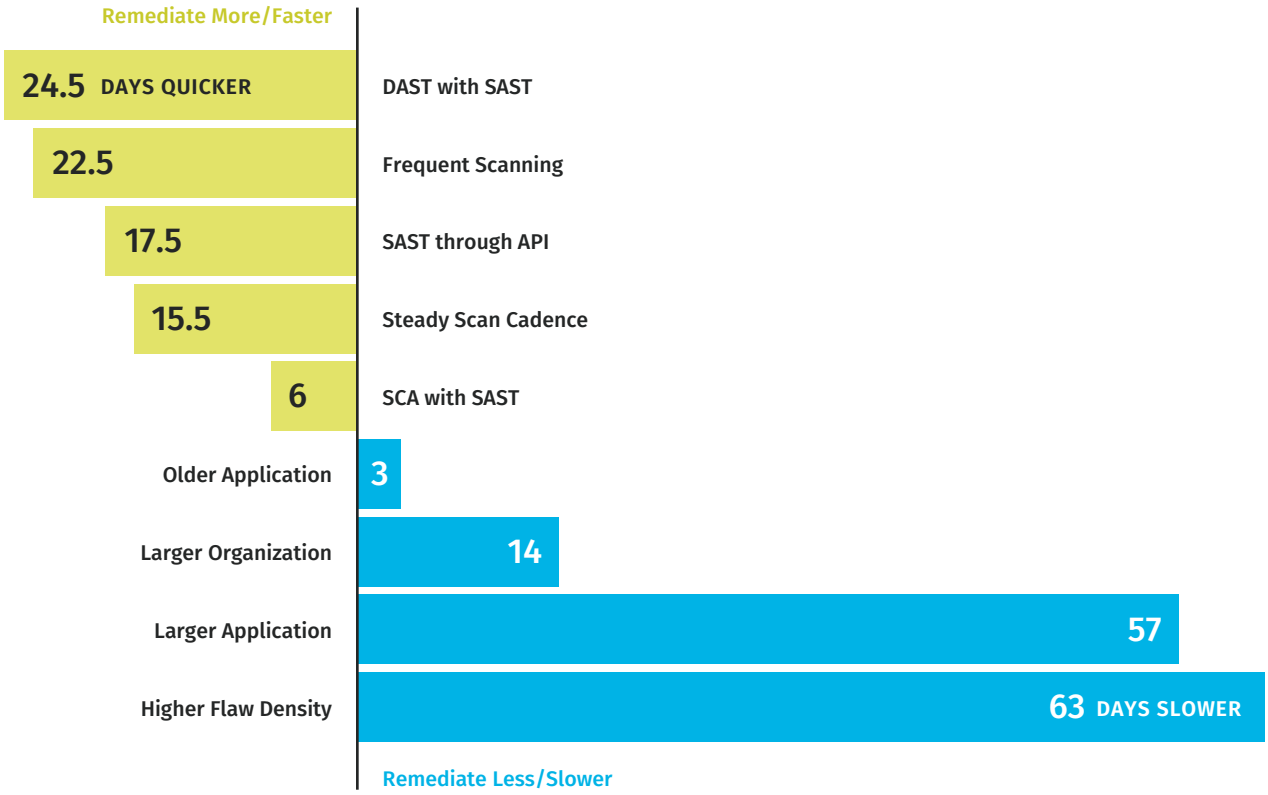


Figure 15: The effect of factors on flaw closure time

Factors pointing to the left are correlated with flaws being remediated more/faster, while those pointing to the right are associated with less/slower remediation. Some of the factors are binary, such as whether dynamic scanning is turned on or off for the application, and others are continuous, such as how frequently an application is being scanned. For continuous variables, the effect represents a shift of one standard deviation in the variable. Encoding the continuous variables this way allows a relatively easy comparison across the disparate scales for each variable.

THE "NATURE" OF APPLICATIONS

One thing that is clear is that one of the biggest obstacles for developers is a ponderous application with a dodgy security history. Large applications with high flaw density slow down the remediation rate of flaws by about 2 months each. Now with our nature versus nurture analogy, it's not typically possible to change any of the factors on the nature side. But we are talking about applications that we created, so we should have some influence in the nature we create or even the nature we are handed.

IF YOU ARE BUILDING UP A NEW APPLICATION

Keep an eye on the size and complexity of the application. Larger applications are clearly associated with slower/less remediation times. As development progresses, stay on top of the flaws as they are discovered — don't let that security debt pile up.

IF YOU HAVE INHERITED A LARGE APPLICATION WITH SECURITY DEBT

The data suggests that teams should consider rearchitecting their applications or retire legacy applications in favor of streamlined code. Refactoring applications to use microservices may help clean up some of the issues, especially if the same flaw is present in different parts of the application.

THE "NURTURING" OF APPLICATIONS

But there is hope, as there are several things that a developer can have more direct control over, and those are the things we generally associate with good development practices and faster flaw remediation.

SCAN FREQUENCY AND DYNAMIC SCANNING

Scanning frequently and using dynamic scanning in addition to static can each reduce the half-life by as much as a month. Dynamic scanning may improve fix rates because it highlights to developers that a vulnerability does, in fact, have "real-world" risk.

API INTEGRATION

There are positive, though smaller, effects for API integration (again building security scanning into the developer pipeline), software composition analysis, and setting up a steady scanning cadence. We noted earlier that the API integration can be linked to scan frequency, and we see that relationship in this chart. Developers should ensure that they reap the benefits of frequent application scanning by making sure the API is part of their development workflow.

We should pause for a moment and consider these results in a larger context. The results above echo what people in application security have assumed. But suspecting something is very different from actually having it confirmed empirically in the data, and to our knowledge, this is the first time someone has taken these assumptions and measured it. When we tell developers the performance of teams with specific behaviors are different from those without those behaviors, we can now show and talk about just how much they differ. We can clearly see the impact of security debt on older applications here, as it slows down the pace of fixing flaws by months and hampers future development.

Nature vs. nurture

Even though we looked at each of the factors individually, we know that very few of these things exist in isolation in real-world software development.

So we look at the factors as a group, and ask that age old question: nature or nurture? In order to determine whether the greatest impact comes from things the developer can't change, or decisions under the developer's control, we begin with two hypothetical applications. The first comes from an ideal environment (nature), which is defined as a small organization with an application that is small in size, has low flaw density, and is relatively new (application age). The second has less ideal properties, which is a large application built long-ago at a large organization with high flaw density. We also have two hypothetical teams working on these applications. One team follows best practices, including frequent, regular scanning, using a variety of scanning types, and including software composition analysis. The other team uses only static scanning, infrequently and at irregular intervals.

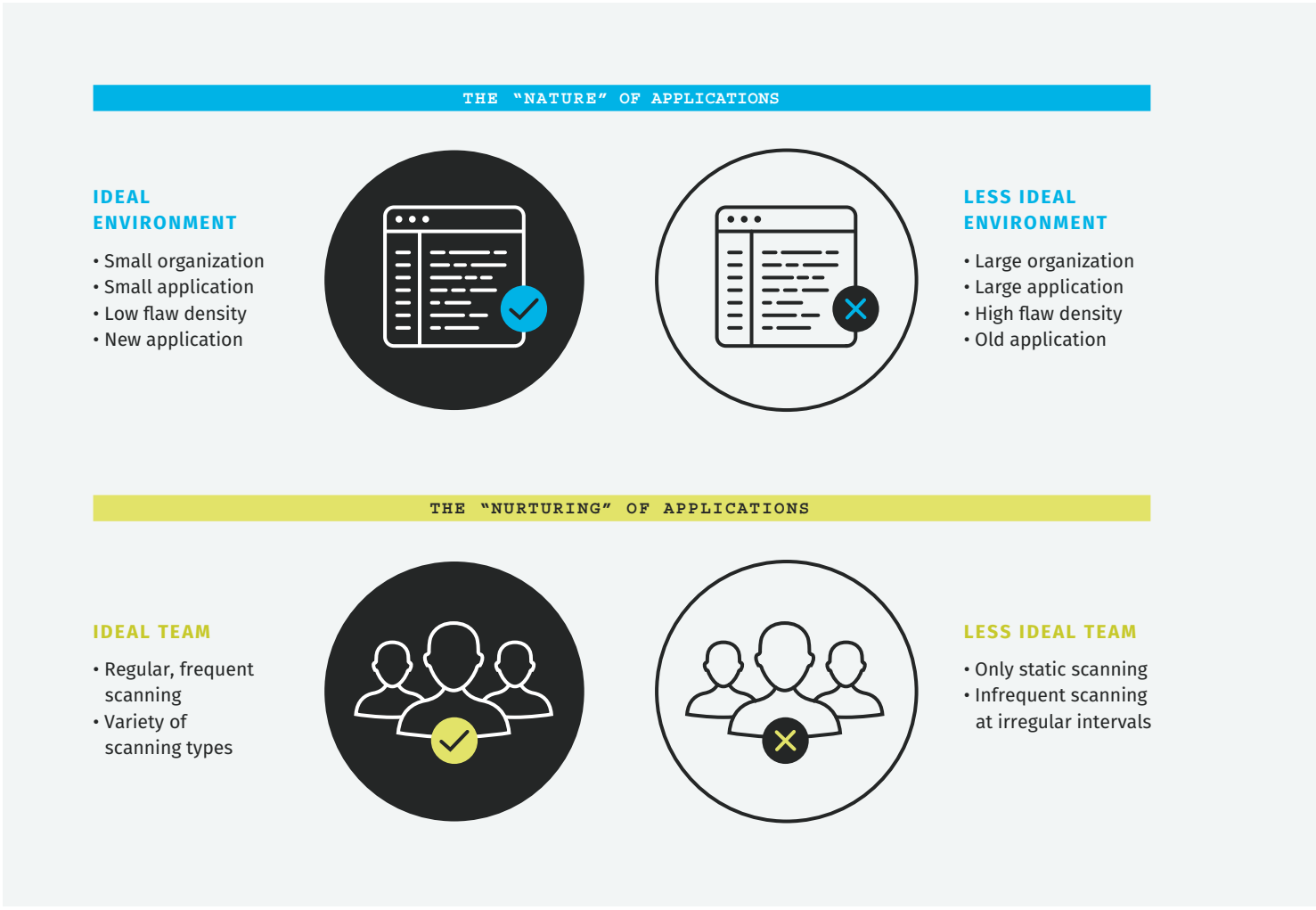


Figure 16 shows how quickly each team — with positive and negative behaviors — closes the flaws in each application — with positive and negative attributes. The slowest line here (the top line closest to the upper right) represents an application challenged with negative attributes and negative behaviors of the team. They tend to close quite a bit slower and less flaws than anyone else. The quickest line here (the bottom line closest to the lower left) represents the best case: positive attributes of the applications with a proactive development team with positive behaviors. In reality, most applications will fall between those two with a mix of attributes and actions.

What's interesting here is the impact good practices can make. In our idealized "good" application, having good practices mean 50 percent of flaws are closed in just under 2 weeks (13 days), while bad practices on that same application can mean it will take almost twice that time (25 days) to close 50 percent of flaws. The differences are even more stark when looking at "bad" applications. A team with bad practices working on a less-than-ideal application may take nearly a year (314 days) to close 50 percent of flaws. A team with good practices on that same unideal application would cut that time to about 6 months (184 days).

Figure 16: The remediation rates of positive and negative attributes and actions

In previous reports, we've discussed the link between frequently scanning applications and faster remediation times. With this year's data, we see a clearer pattern emerge with other developer behaviors. It makes sense that flaws are fixed quicker in applications that are being developed under ideal conditions, but the pattern shows that developers can also influence the outcome by changing their specific behaviors.

SECTION FIVE

Conclusion

Even if the developer has inherited an old, gargantuan application with heaps of security debt, and there is no one left who remembers why some things were coded that way, fixing flaws and adding new features don't have to continue being difficult. What the data tells us is that even when faced with the most challenging environments, developers can take specific actions to improve the overall security of the application. Several of the developer best practices we highlight in this SOSS align closely with behaviors we typically associate with DevSecOps.

Scanning applications frequently and on a regular cadence and fixing the flaws as they are found (and not waiting for major releases) is common practice among DevSecOps teams. We see the effects of using different types of scanning technologies in order to get a more comprehensive view of the application. We see how fixing flaws in smaller and newer applications tend to be quicker, which may encourage decisions such as re-architecting parts of the application to smaller components.

Embedding security testing into the pipeline (through an API) is another sign of the team's approach to continuous integration. That automation can tighten up the cycle of feedback developers receive and make security testing more effective, and indeed we see improved remediation times with that integration.

We've looked at the effect of nature and nurture on the security of our applications. We found that nurture — our decisions and actions — can overcome and improve the nature of the application and environment. There are many solutions available for developers to help them discover and manage the flaws that creep into applications.

But know this: you are able to take action and make decisions that will improve the security of your application!

TO LEARN MORE ABOUT SOFTWARE SECURITY, CONTACT US.

Appendix: Methodology

Veracode methodology for data analysis uses a sample of applications that were under active development from a 12-month sample window. The data represents the full history of applications that had assessments submitted from April 1, 2019 through March 31, 2020. This differs from past volumes of the *State of Software Security,* as we only looked at the assessments that occurred in a 12-month window and not the entire history of applications. This accounts for a total of 132,465 applications, 1,049,742 scans, and 10,712,156 flaws. The data represents large and small companies, commercial software suppliers, software outsourcers, and open source projects.[9] 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 these snapshots, we examine the most recent scan.

For the software component analysis, each application is examined for third-party library information and dependencies. These are generally collected through the applications 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 data that was provided by Veracode's customers (application portfolio information such as assurance level, industry, application origin) and information that was calculated or derived in the course of Veracode's analysis (application size, application compiler and platform, types of vulnerabilities, and Veracode Level — predefined security policies which are based on the NIST definitions of assurance levels).

Any reported differences (between languages, scan types, flaw types, etc) are statistically significant at the $p < 0.001$ level. Because of the large data size we are able to discern even incredibly small effect sizes.

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% of scans closed three or less findings, 75% closed less than 8), 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: developers would scan entire filesystems, invalid branches or previous branches, and when they would rescan on the valid code, every finding not found again would be marked as "fixed." These mistakes had a large effect: the top one-tenth of one-percent of the scans (0.1%) accounted for almost a quarter of all the closed findings. These "mass closure" events have significant effects on exploring flaw persistence and time-to-remediation and were ultimately excluded from the analysis.

A NOTE ON "SANDBOX" SCANS

Developers will sometimes create a "sandbox" for the purpose of a one time evaluation of a piece of code. Unfortunately, these scans are divorced from any information about the application and its history. In the future we may examine how the use of these sandbox scans might affect the mainline analysis of applications. For now, these scans are excluded from the analysis.

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

TO LEARN MORE ABOUT SOFTWARE SECURITY, CONTACT US.

VERACODE

Veracode is the leading AppSec partner for creating secure software, reducing the risk of security breach and increasing security and development teams' productivity. As a result, companies using Veracode can move their business, and the world, forward. With its combination of automation, integrations, process, and speed, Veracode helps companies get accurate and reliable results to focus their efforts on fixing, not just finding, potential vulnerabilities. Veracode serves more than 2,500 customers worldwide across a wide range of industries. The Veracode cloud platform has assessed more than 14 trillion lines of code and helped companies fix more than 46 million security flaws.

www.veracode.com **Veracode Blog** **Twitter**