State of Software Security 2023

Annual Report on the State of Application Security

An Ounce of Prevention is Worth a Pound of Cure:

How to Reduce Security Debt and Avoid Introducing Security Flaws That Accumulate Over the Life of Your Applications

VERACODE

Veracode is a 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 process automation, integrations, speed, and responsiveness, Veracode helps companies get accurate and reliable results to focus their efforts on fixing, not just finding, potential vulnerabilities.

Learn more at www.veracode.com, on the Veracode blog and on Twitter.

Contents

Section One

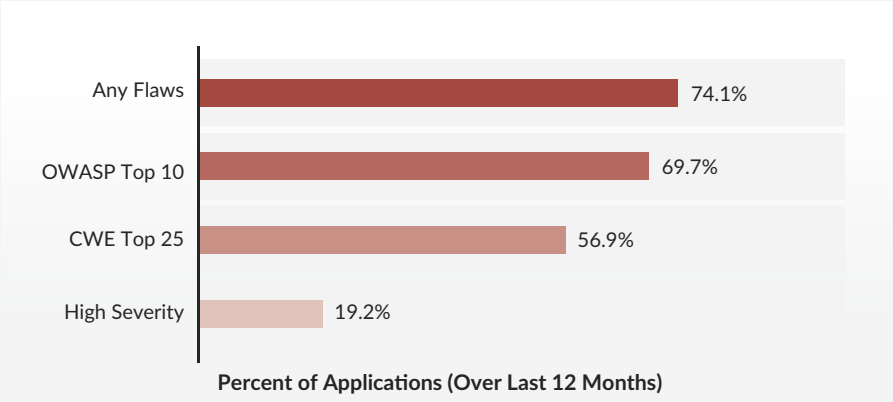
The State of Software Security at a Glance

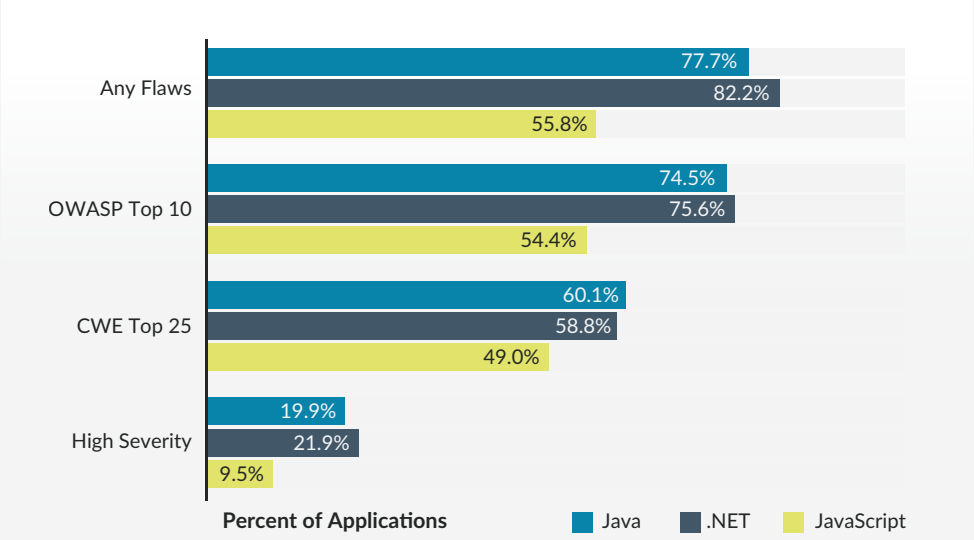
In the latest edition of the State of Software Security, we use hard data to establish what factors go into flaw introduction, faster remediation, and lower security debt. We also turn conventional wisdom on its head in our look at fragility and the health of the open-source ecosystem. Finally, we provide concrete steps you can take now to improve your application security program for 2023 and beyond, because an ounce of prevention is worth a pound of cure.

Flaw Prevalence

Over 74% of applications have at least one security flaw found in the last scan over the last 12 months.

These include over 69% have at least one OWASP Top 10 flaw, and over 56% have at least one CWE Top 25 flaw.



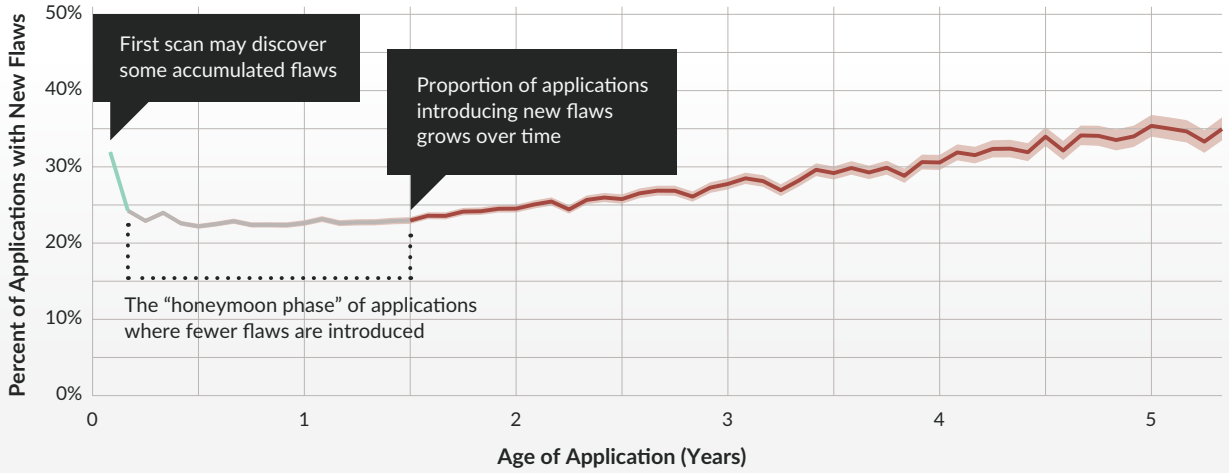


Flaw Prevalence by Language

JavaScript generally has fewer flaws with just over half of applications with any flaws reported, while about four out of five Java and .NET applications have any flaws.

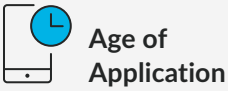
Flaw Introduction by Age of Applications

While over 30% of applications show flaws at the first scan, this number drops to approximately 22% shortly after before rising to 30% again at four years. The number of applications with new flaws then increases further to approximately 35% of applications over four and a half years old.



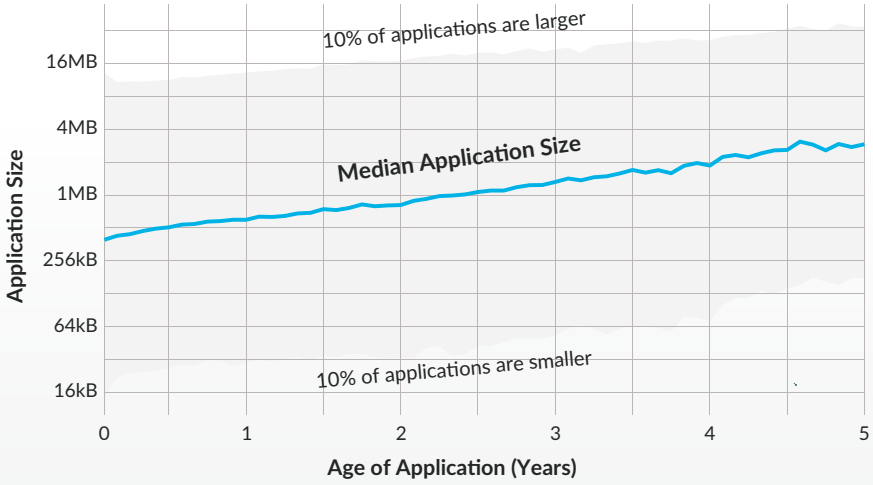
Application Size by Age of Applications

Applications grow in size by about 40% year on year irrespective of their original size.



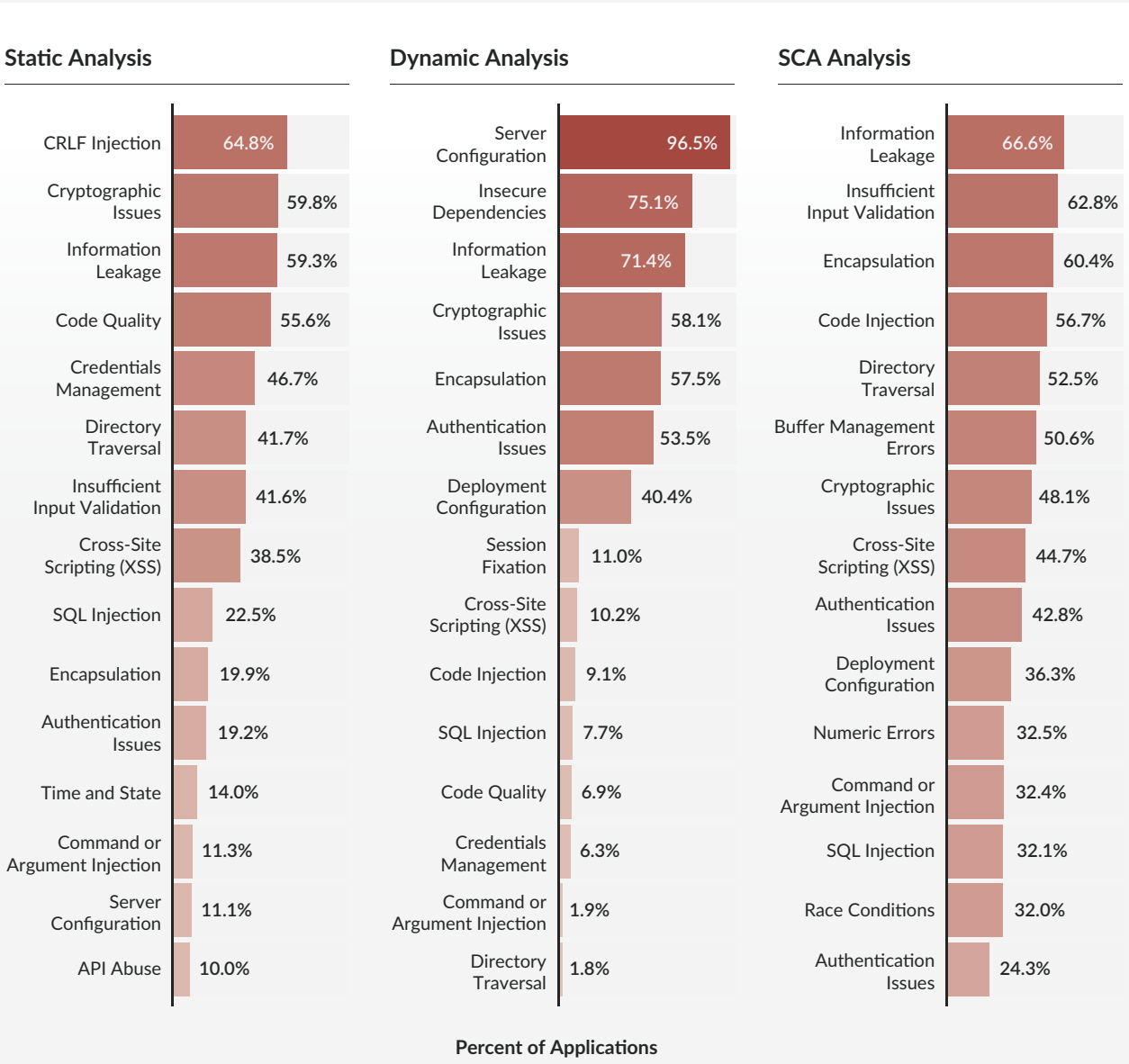
Age of Application

The number of years on the Veracode Platform



Top Flaws by Scan Type

The top flaws vary markedly by scan type. While this is not news, it does highlight the importance of using a variety of scan types to ensure finding hard-to-identify flaws that may only be detectable by one type of scan.



Factors That Impact Introduction and Accumulation of Flaws

Scan Frequency

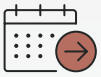
Scans Last Month

0.4

Reduction in the ***probability*** that new flaws will be introduced into applications.*

1.6

Reduction in the ***number*** of flaws introduced when flaws are introduced into the application.



Every Month Since Last Scan

1.3

Increase in the ***probability*** that new flaws will be introduced into applications.*

5.1

Increase in the ***number*** of flaws introduced when flaws are introduced into the application.

Scan Type

Scanning Via API

2.0

Reduction in the ***probability*** that new flaws will be introduced into applications.*

17.9

Reduction in the ***number*** of flaws introduced when flaws are introduced into the application.

Developer Education

Completion of 10 Security Labs Trainings

1.8

Reduction in the ***probability*** that new flaws will be introduced into applications.*

12.1

Reduction in the ***number*** of flaws introduced when flaws are introduced into the application.

*From a base of 27% in any given month.

Section Two

Introduction

Welcome to another installment of the State of Software Security. We are teamed up with the superstar data analysis team at Cyentia once again. We have some great research in store for you this year. We wanted to iterate on some of our previous work and talk about some of the things you can do to make your program better.

To start with, we are focusing this year on things that influence flaw introduction and what it means for an application's lifecycle when flaws are introduced.

To do that, we wanted to drill down a little bit and break things out by the top three languages used by developers. We've seen similar flaw profile differences among languages in the past, but we want to be clear up front: Rather than trying to knock one language, we want to establish what factors go into flaw introduction, faster remediation, and lower security debt regardless of the language to set a pattern that any application written in any language can follow to achieve similar results.

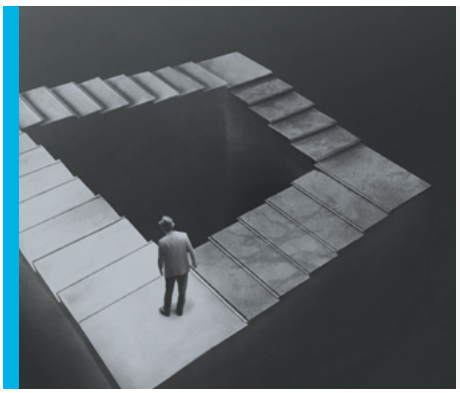
The lifecycle of an application and the pattern of flaw introduction turns out to be more predictable than one may think. Once we understood how an application's security posture (flaws, growth rate, etc.) progresses through its production lifecycle, we were able to establish what appear to be fairly distinct phases. We wanted to look at what could be done to improve, so we dug in to find out.

Now that we had analyzed some of the factors that define the profile of flaw introduction and remediation over time, we saw patterns emerge. What can be done to improve that picture? To reduce the chances that a flaw is introduced in the first place and then the number of flaws when they are introduced? We devote an entire section to showing the significant insights we found.

We looked at fragility and the health of the open-source ecosystem this year. We think that what came out of our analysis somewhat turns conventional wisdom on its head. It did not start that way, but the research raises more questions than it answers. We are ready to show what we found and challenge some accepted principles.

Flaw

A flaw is an implementation defect that can lead to a vulnerability, and a vulnerability is an exploitable condition within your code that allows an attacker to attack.



Read all the way to the end and we'll reward you by taking what we have learned and making some suggestions for us all to put into action. The whole report describes a path to our recommendations and should give everyone plenty to think about. An ounce of prevention is indeed worth a pound of cure.

We hope you'll enjoy the journey.

Section Three

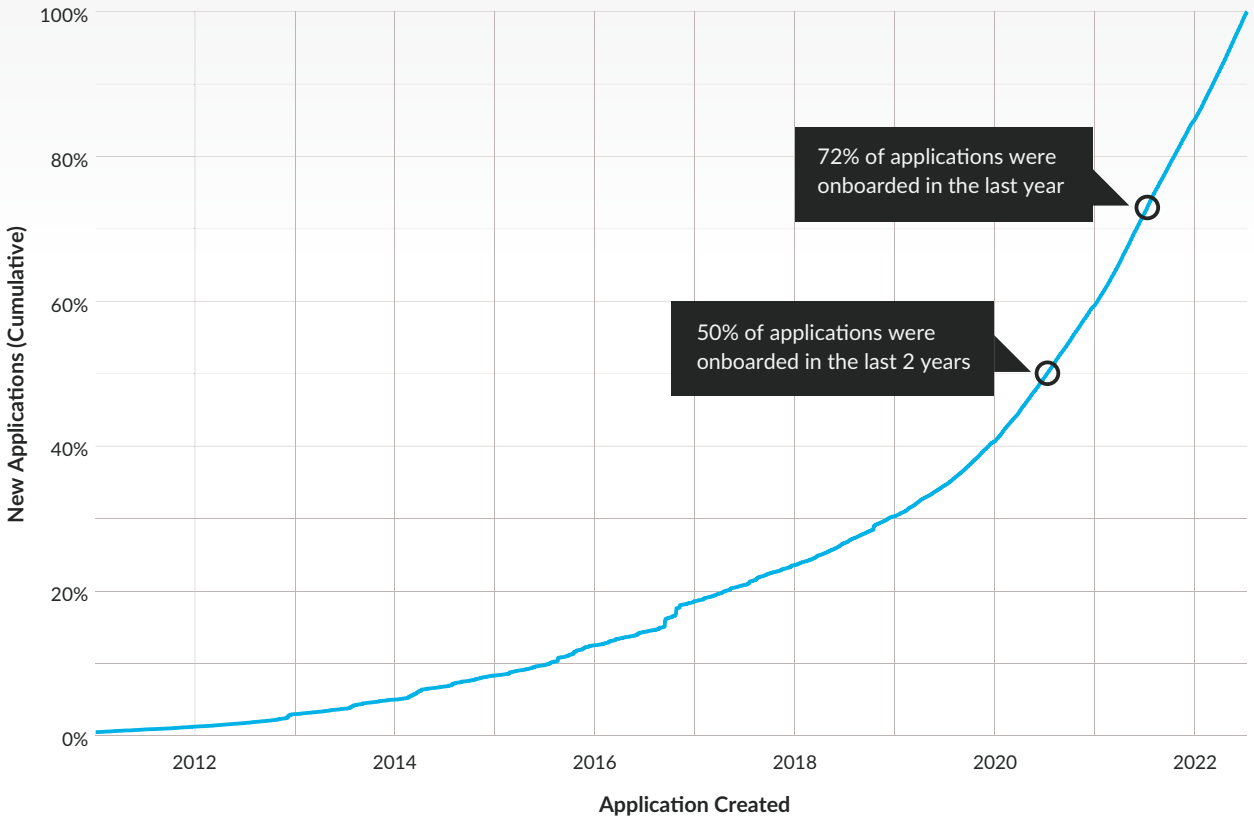
Where We Are

Effective cloud security means that the inherent exposure (potential for reachability) for cloud-native apps must be addressed in layers. Aside from technical access controls it means that secure coding practices are all the more crucial.

Studying Applications

We begin with a look back in time. The growth shown in Figure 1 is the cumulative number of applications that Veracode customers have created, which continues to grow unabated. Based on the well-distributed demographics of our customer data we can infer that many other organizations are currently experiencing similar growth. While the number does skew upwards via the increased adoption of microservices as an architectural design choice, it's pretty clear from the growth rate that application count is on the rise.

Figure 1: Cumulative Applications Created by Veracode Customers



By Languages

The distribution shown in Figure 2 is weighted more heavily towards those languages that organizations (not individuals) use to deliver their applications. While we have seen other languages growing quickly in popularity, Java, .NET, and JavaScript take spots one, two, and three respectively as the most used in this research. That is not just some random thing we want to show.

We found distinct trends that indicate that the choice of programming language has an effect on the:

1. Types of flaws introduced
2. Ecosystem of libraries and third-party software

A look back at SoSS v11

Over 90% of Java applications are third-party code.[1]

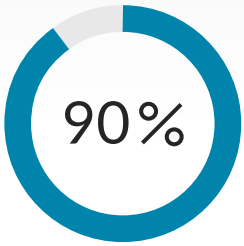
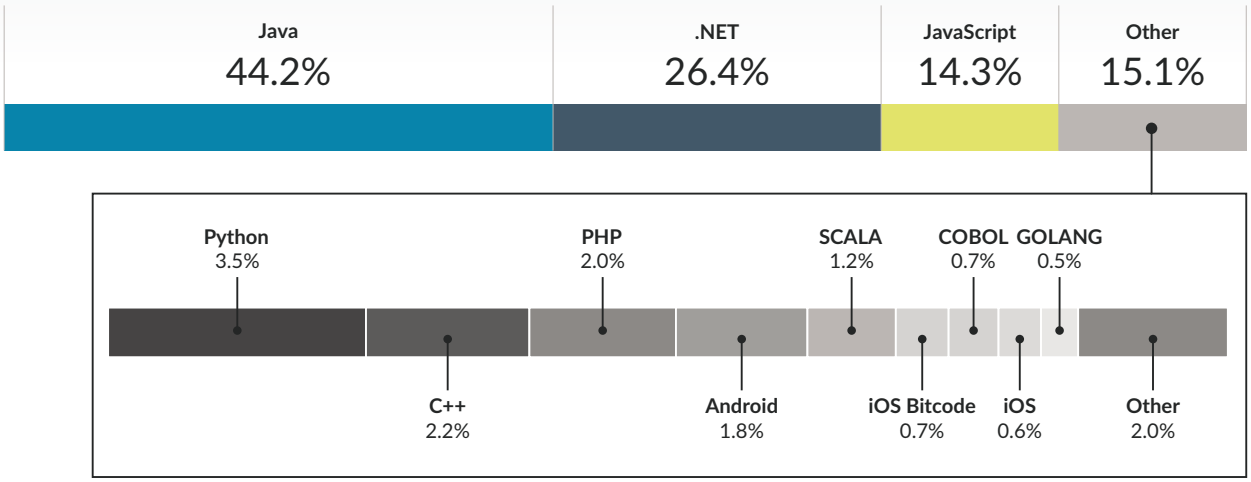


Figure 2: Languages in Use by Proportion of Applications



We've shown in the past, for example, that over 90% of Java applications are third-party code,[1] while most other languages include more homegrown code. A look at a breakdown by language shows how different they can be when just scanning with static application security testing (SAST). We continue to see that some languages are simply more or less prone to some flaws over others, which we'll get into later on.

[1] SoSS 11 Open Source Edition info.veracode.com/fy22-state-of-software-security-v11-open-source-edition.html

Discovering Flaws

For a quick "at a glance" look at the general state of software security, we first examine a point in time snapshot, focusing on those applications scanned within the last 12 months.

We see in Figure 3 that, over the past 12 months, about three out of four applications have at least one flaw in the last scan result. In this figure "high severity" is defined as high or critical severity flaws. Slightly fewer than one in five applications have reported such flaws in their last scan.

Figure 4 shows the fluctuation over time, looking at a sliding time window. While Figure 3 looked at all of the applications over a single 12-month window, Figure 4 slides a rolling two-month window, counting up the proportion of applications that had at least one flaw in each flaw category. This rolling view smooths out short-term irregularities in the data, showing us the evolution of flaw findings, as opposed to the previous snapshot bar chart. Generally, we can see that things are improving. Every measurement trends downwards over the last six years.

Figure 3: Existing Flaws in Applications

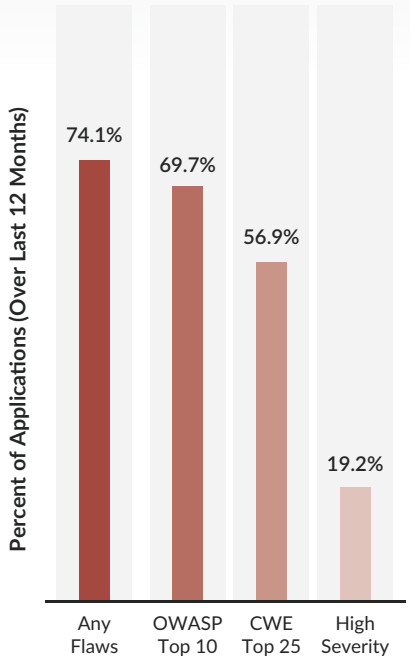
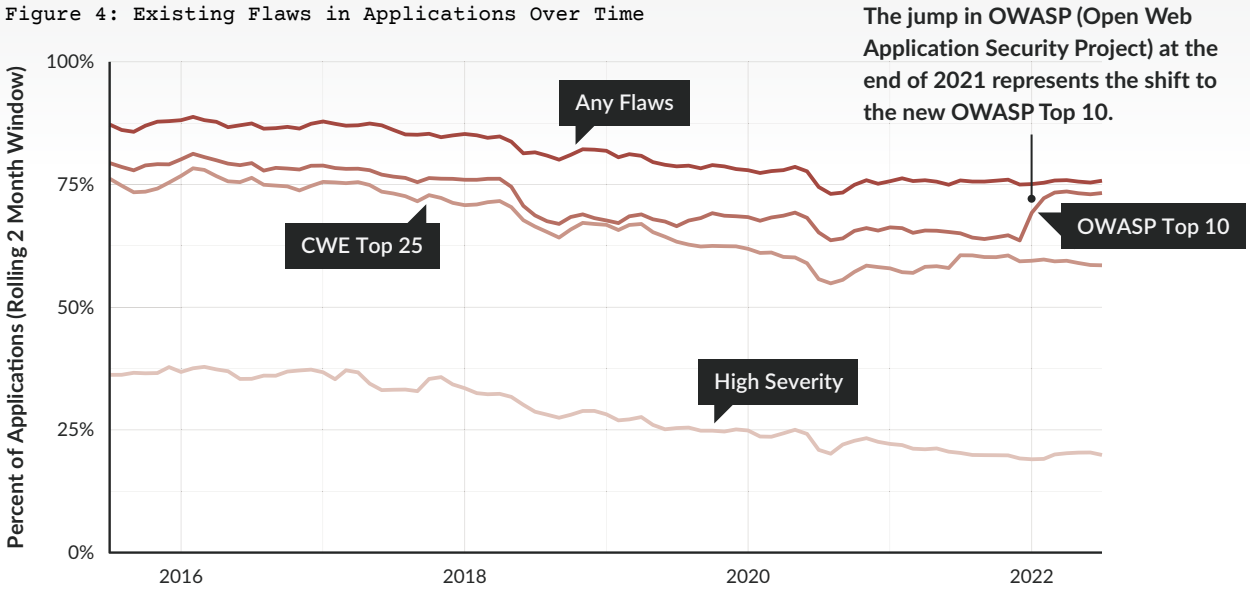


Figure 4: Existing Flaws in Applications Over Time



Beginning to Break It Out by Language

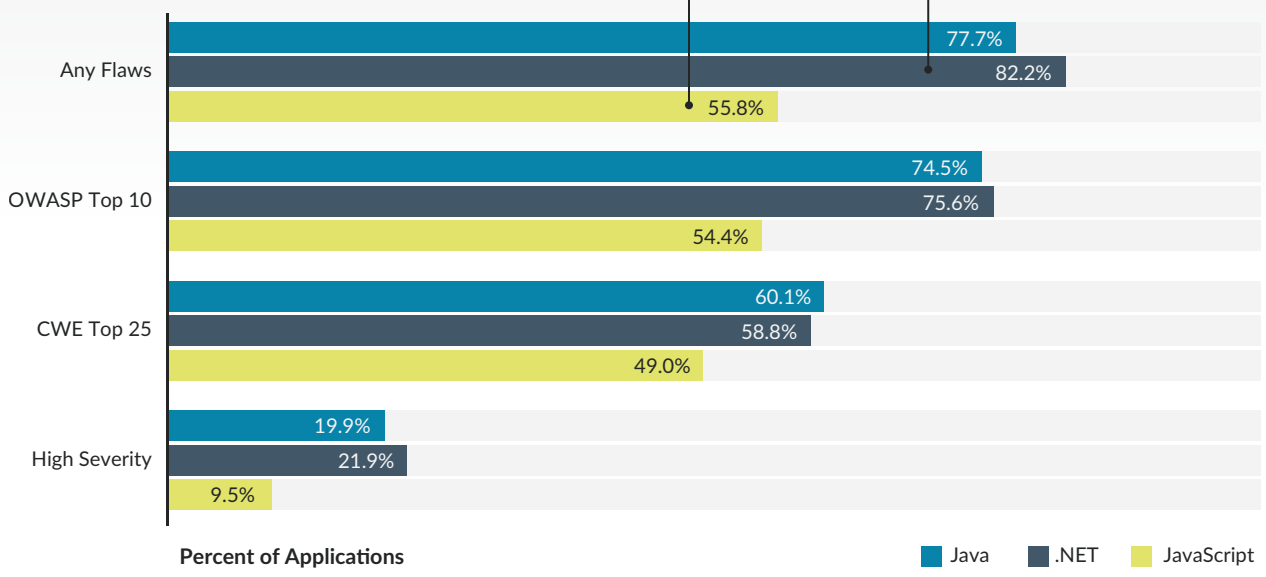
We've looked at the top flaw category by language in the past, so this time we wanted to take it a step further. Rather than merely noting the top flaws by language, we wanted to find out whether there were variations over the lifetime of an application in production. Perhaps more importantly, what steps can be taken to reduce the introduction of flaws in the first place? Being informed and then vigilant should enable that. Ideally the data will help us get a better handle on flaw introduction, security debt accumulation, and application lifecycle management.

The first thing that jumped out during analysis was that there were different inherent security postures for the "Top Three Languages." There is also a different rate at which flaws were fixed, leaving a higher (or lower) de facto chance that flaws will simply accumulate over time.

Accumulation of flaws can be referred to as security debt and is a subset of technology debt. This is defined as the number of net flaws remaining when considering flaw introduction and remediation rates. Different languages pay down at different rates than they build up and that makes a positive or negative difference in accumulation over time. We'll examine this in the next section, as we break out the top three languages in this research. First have a look at the "any flaws by language" view in Figure 5 below. While Java and .NET appear similar, JavaScript is different enough to warrant further examination. What we discovered were differences that were not visible at the top level. That's why we went down this rabbit hole, and it's where you are going next.

JavaScript generally has fewer flaws with just over half of applications with any flaws reported, while almost five out of every six .NET applications have reported flaws.

Figure 5: Existing Flaws in Applications by Language



Depending on your role, you may get different things out of the data we will be presenting. A developer might be interested to find out the most common flaws introduced and, once those are identified, take conscious steps to learn how to avoid them. A person responsible for security might be interested to see the rate of flaw accumulation and what that means to the overall risk posture.

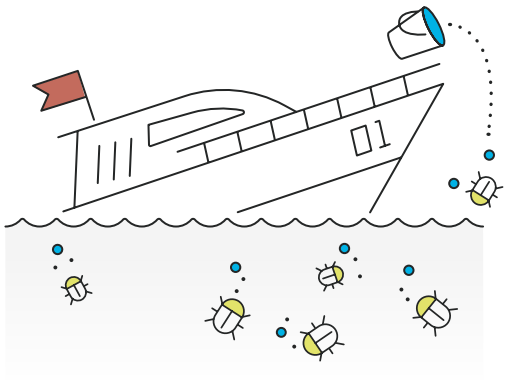
Finally, application stakeholders may wish to think about application lifecycle management and rethink the benefits of planned obsolescence or additional investment in maintenance. We know application rewrites are expensive and the idea of planned obsolescence sounds radical but bear with us and keep an open mind.

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.

Collecting Flaws

Paying down technical or security debt could be likened to bailing a sinking boat at different rates depending on multiple factors both technical and organizational. Whatever strategy an organization has to deal with it, flaws generally collect over time until the boat fills faster than one can bail.



Flaw Types, Rate of Introduction, and the Uniqueness of Flaws by Languages

Overall View

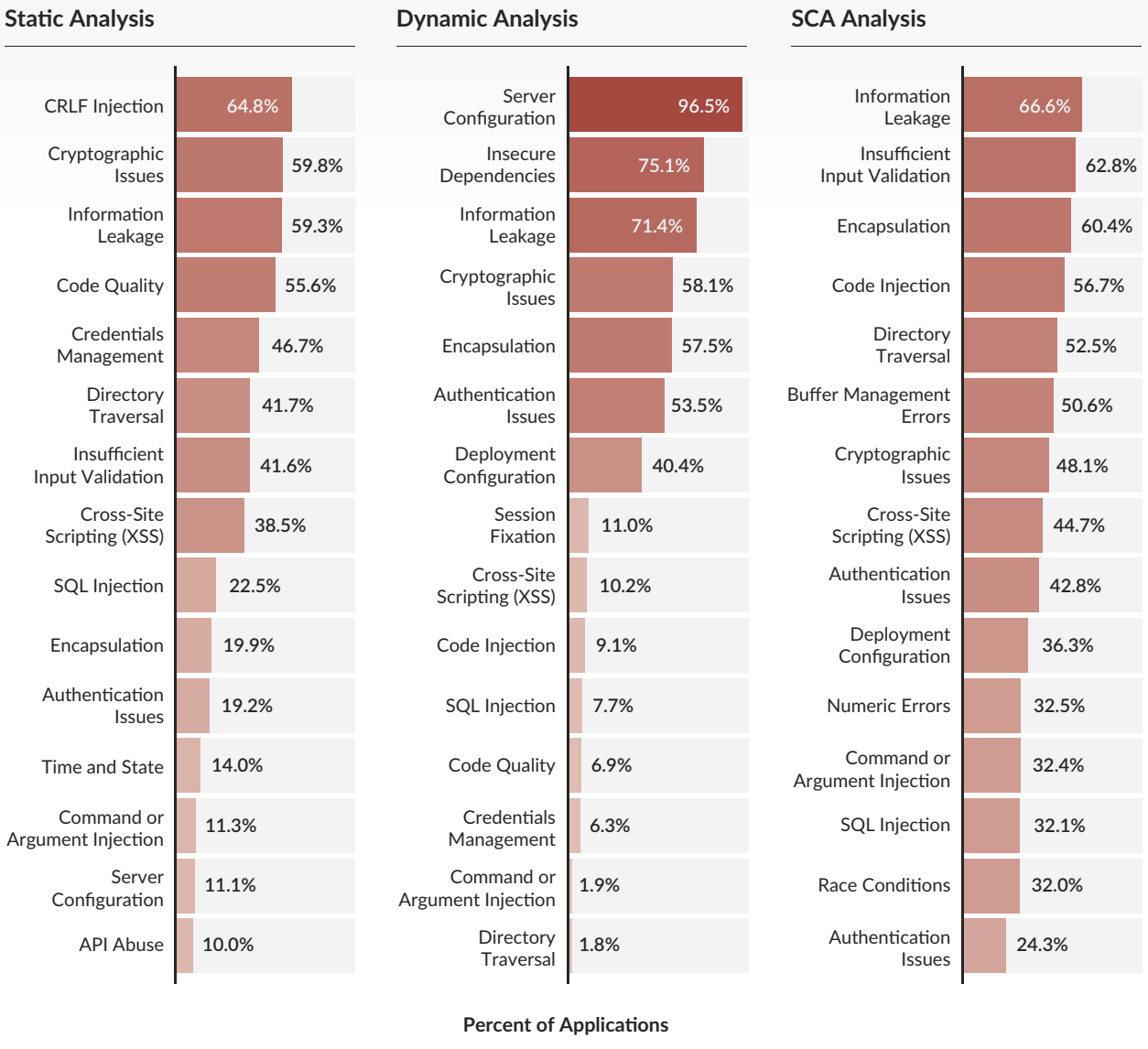
What security challenges are developers in each language facing? Our first view is the overall types of flaws by scan type and how prevalent they are. As we explored the results this year, we felt that this plot (Figure 6) can be challenging to interpret, and that's where things became heated in our behind-the-scenes discussions. For example, you may look at Figure 6 and take away a message that all developers should learn more about Carriage Return Line Feed (CRLF) Injection flaws, but those disproportionately affect Java applications.

While .NET and JavaScript applications still have those flaws present, they generally do not take the top slot of where those developers should focus their attention. Plus, breaking out only by scan type tells us more about what types of flaws the scan types may find and less about where developers focus. To see what we mean there, take a look at Figure 6 and observe that static, dynamic, and composition analysis find completely different flaws in their top five.

In its defense, Figure 6 does give a nice overview of where broad challenges exist, and it does highlight that if you only focus on a single type of scanning you may be missing some hard-to-identify flaws that may only be detectable through another type of scan.

Regardless of what any of us may think of Figure 6, we thought it was clear that this year we should focus more on the challenges developers are facing in the individual languages.

Figure 6: Top Flaw Types by Scan Type



Top Flaws by Language

The choice of programming language has an effect on the types of flaws that are most commonly introduced, and it affects the ecosystem of libraries and third-party software. Slowing down and taking a look at this reality is useful for those individuals or organizations that wish to prioritize their training to know what the most common flaws are, and how they might be introduced. This basic awareness can influence code as it is being written — the best time to avoid introducing a flaw that could hang around throughout the lifecycle of an application.

Developer awareness of what the most common flaws are, and how they are introduced, can increase diligence and reduce the probability of introducing them at all.

Keep in Mind

We want to be crystal clear that when we are talking about your preferred programming language, we are not calling out specific languages or programmers. Whether we admit it or not, flaws happen, and they happen in any and every programming language.

These flaws, however, are not evenly distributed. The way different languages are architected and implemented can make some security mistakes easier (or harder) to make and that's what we want to highlight to make us all better.

Remediation Timeline

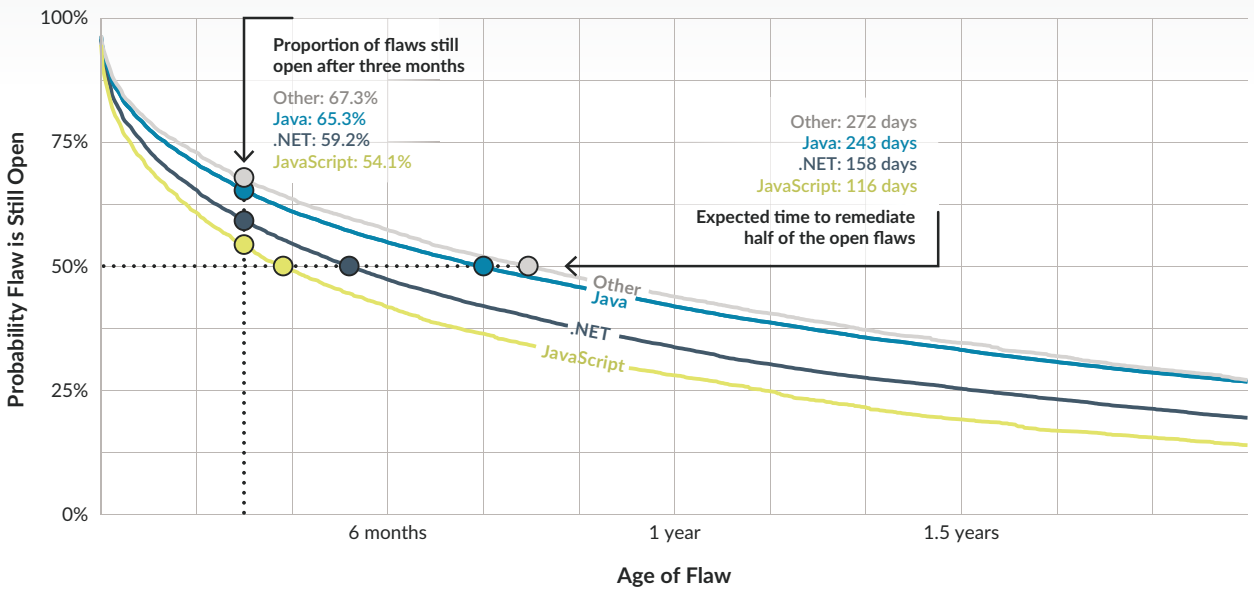
In SoSS v12 we looked at flaw remediation rate by scan type, but this year we wanted to discover differences by language. The best way to look at time-to-event is with a set of techniques collectively referred to as "survival analysis."[2] This type of analysis accounts for both the closed flaws and those still open (not yet closed). It provides a much more realistic picture around remediation timelines.

The curves in Figure 7 look at the probability that a finding is still open as a function of the time since the flaw was first discovered. A few things jump out immediately. Our top three languages reach the halfway mark (expected time it takes to remediate half of the open flaws) at different times. At first glance, the lines appear to be descending together closely, but they definitely aren't.

With Java we see the probability that a flaw is still open after three months at 65.3%. That is not just a "so what?" moment. Given the corresponding remediation rate, over a two-year timeline these applications only get just slightly below 27%. The lower remediation rate for Java quickly compounds over time.

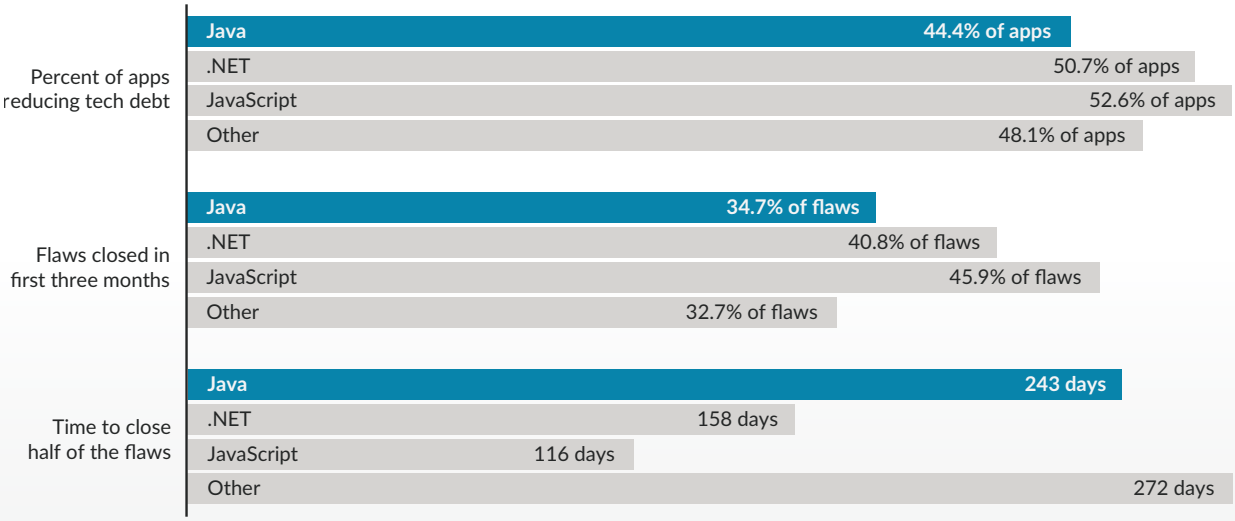
In the language-specific section below you'll see why this matters when we combine factors such as time, flaw introduction rate, density, and others. You'll also see why these things become a differentiator when compared to JavaScript in a straight-up flaw survival competition. Remember where the JavaScript line is for the next couple of sections as we seemingly quibble over what appears to be minor percentage differences. To call it out clearly, only 14% of flaws in apps written in JavaScript are still open after two years.

Figure 7: Time to Remediation by Language



[2] For a full discussion on what flaw survival analysis is, Cyentia wrote this blog post to discuss the topic after our 9th SoSS report. Vintage cool. www.cyentia.com/survival-in-application-security

Figure 8: Various Metrics Across Languages (Java Emphasis)

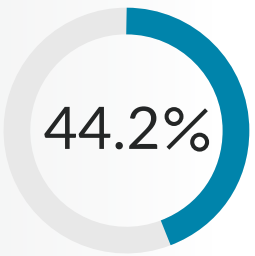


Java

To begin with, let's look at Java applications versus other languages with a straightforward bar chart (Figure 8). In every category Java underperforms the pack. If you look at the percent of apps reducing tech debt this means that the line of 50% divides a net upward or downward trend. You can turn the number around to see that about 56% of Java Applications are increasing rather than reducing their security debt.

Another benchmark number is the first three-month mark. Flaws not remediated by that time frame are likely to be carried forward into eternity. The three-month mark is a good target, but lagging at that mark is bad. Referring back to the remediation curve, if an application's line is not rapidly declining early, then remediation efforts seem to flatten out. Looking down to our bar chart, it takes 243 days to close out 50% of flaws compared to .NET and JavaScript, which accomplish this goal *months* sooner.

Finally we look at the overall percentage of flaws fixed, which at 44% is again lower than .NET or JavaScript.



Java is by far the most common application language scanned by Veracode — accounting for 44.2% of scans.

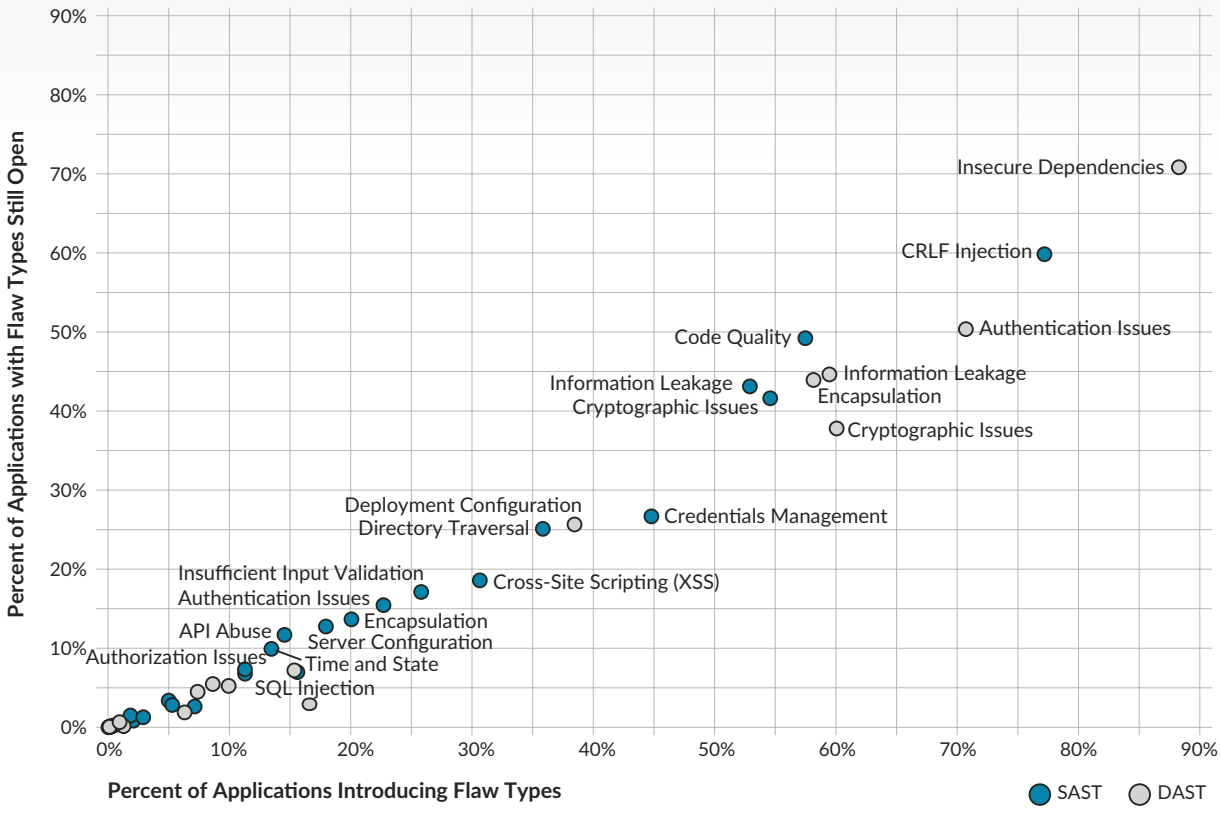
The next plot (Figure 9) looks at the introduction of flaws by Common Weakness Enumeration (CWE) category[3] over the last 12 months, comparing the percent of applications that have introduced one or more flaw types to the percent of applications that have that flaw type still open.

Figure 9 shows the percentage of apps introducing flaws (horizontal axis) and then the chances of that flaw *still* being around (vertical axis), by type of flaw.

This figure should provide a consolidated perspective in terms of guiding a training program, since it rolls up multiple CWEs by category. For example, 77% of Java applications have introduced at least one CRLF injection flaw in the last 12 months (horizontal axis in Figure 9) and 60% of Java applications had at least one CRLF Injection flaw present in their last scan (vertical axis in Figure 9).

Note: This is not filtered by severity.

Figure 9: Introduction of Flaws vs Still Open (Java)



[3] CWEs are numbered individual flaws that are classified into CWE Categories.

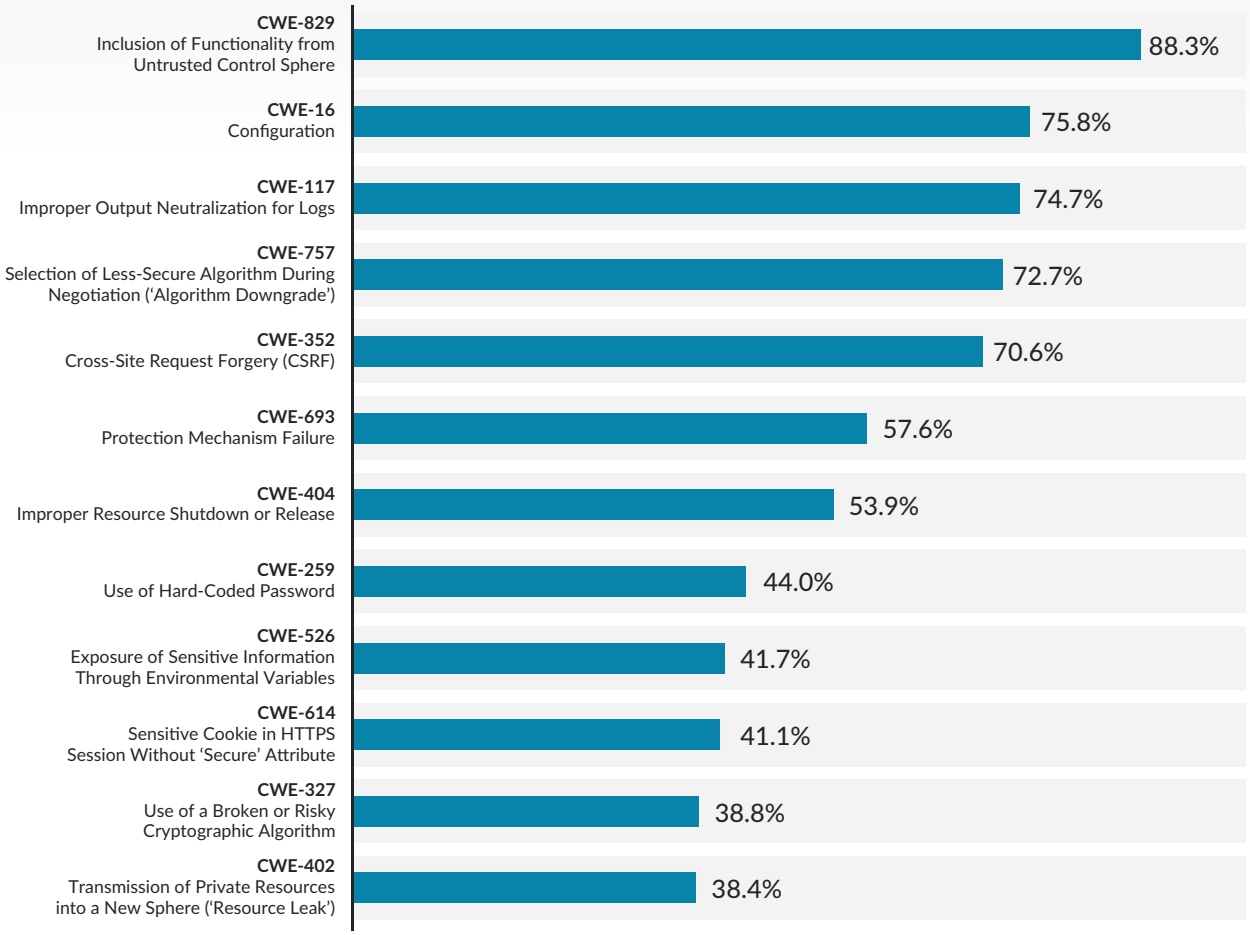
To continue down our language-specific rabbit hole and see where it leads, let's look at all of the applications that introduced flaws in the last 12 months (Figure 10). 88% of the applications that introduced a flaw in the last year introduced at least one CWE-829 weakness (CWE-829 Inclusion of Functionality from Untrusted Control Sphere). As you look top to bottom at the list in Figure 10 you can see an additional dimension for training prioritization, but any program must also consider severity of the flaws and how they might get introduced.

Remember the bar chart view at the beginning of this section (Figure 6) that we said was challenging to interpret? Not Figure 10. It is language specific. We put this together by combining the highest proportion of all flaws across static and dynamic scans. Since we touch on open source later in this research, we are ignoring the scan types beyond those.

Keep in Mind

Figures 10, 13, and 16 show the highest proportion across static and dynamic scans.

Figure 10: Percent of Applications with New Flaws with a CWE in Past Year (Java)



.NET

We turn our attention to .NET and see a slightly different picture as compared to the other languages. 51% of .NET applications are reducing tech debt, which means application developer teams seem to be getting *slightly* more than half the flaws — faster than other languages in this data (with the oft-recurring exception of JavaScript).

When you look at these bars in Figure 11, what at first you might have thought is only a slightly more aggressive remediation percentage rate translates into differences in the remediation curve that you can begin to count in your head. You can see this in the time to close half the flaws as .NET pulls away from Java by close to 100 days. That's encouraging news for .NET, and a peek at the remediation curve (way back up in Figure 7) shows that at the two-year mark about one in five flaws are still open in .NET. Java comes in at just over one in four. Then with the remediation curve in mind, if you compare Figure 12 versus Figure 9 it is clear that a slightly more aggressive remediation percentage translates into reduced chances that something is still open. Overall it is a second place sweep for .NET.

The Problem

The problem here can be the severity of flaws that are introduced and the time it takes to fix them. As mentioned previously in the Java section, each language seems to have its own predisposition to high- and critical-severity flaws that then wind up appearing in large numbers.

Figure 11: Various Metrics Across Languages (.NET)

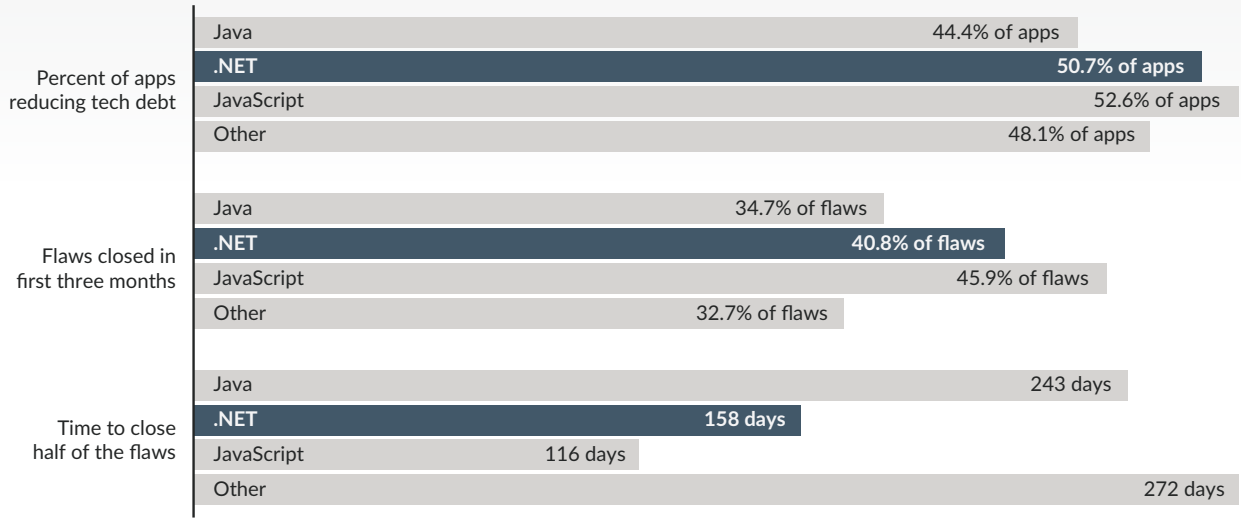
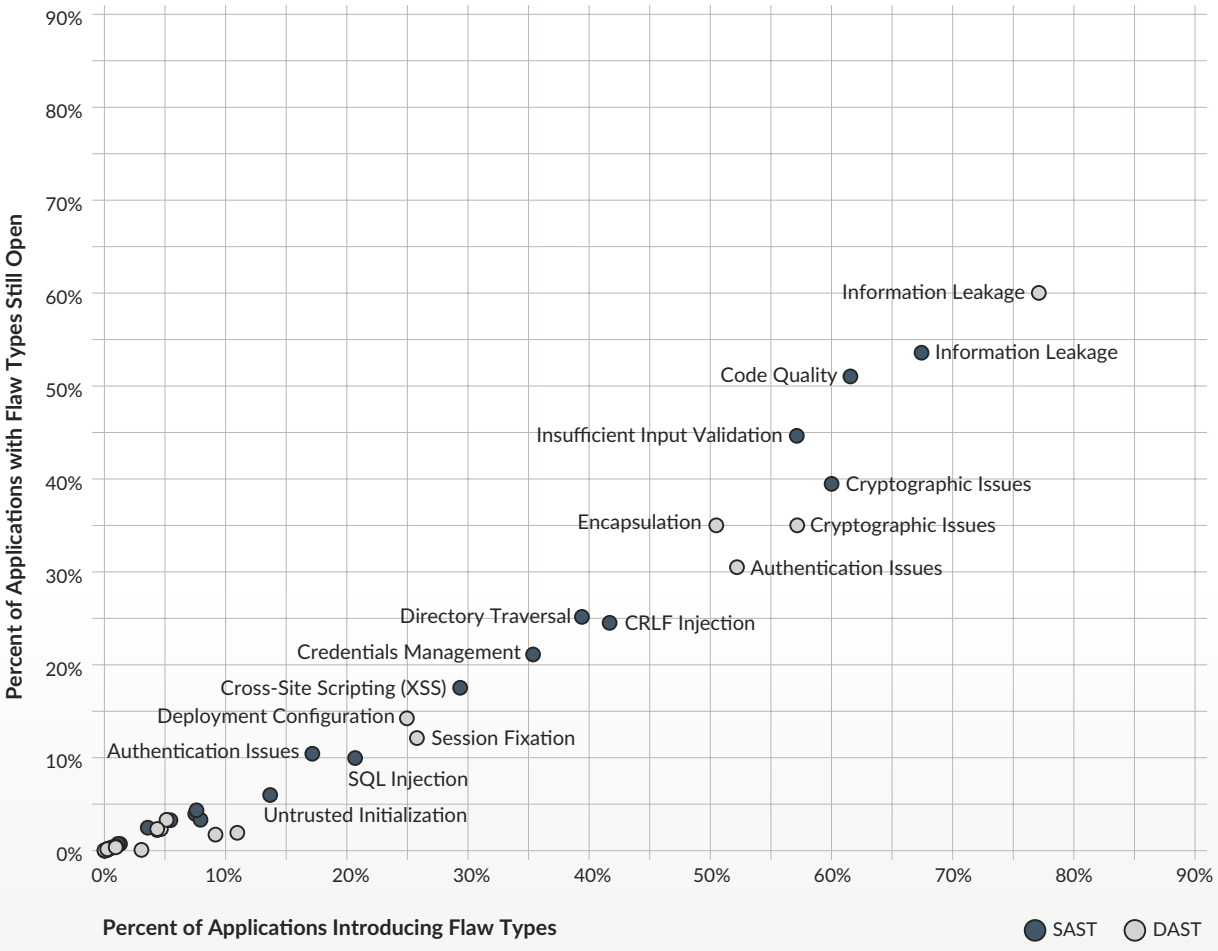


Figure 12: Introduction of Flaws vs Still Open (.NET)

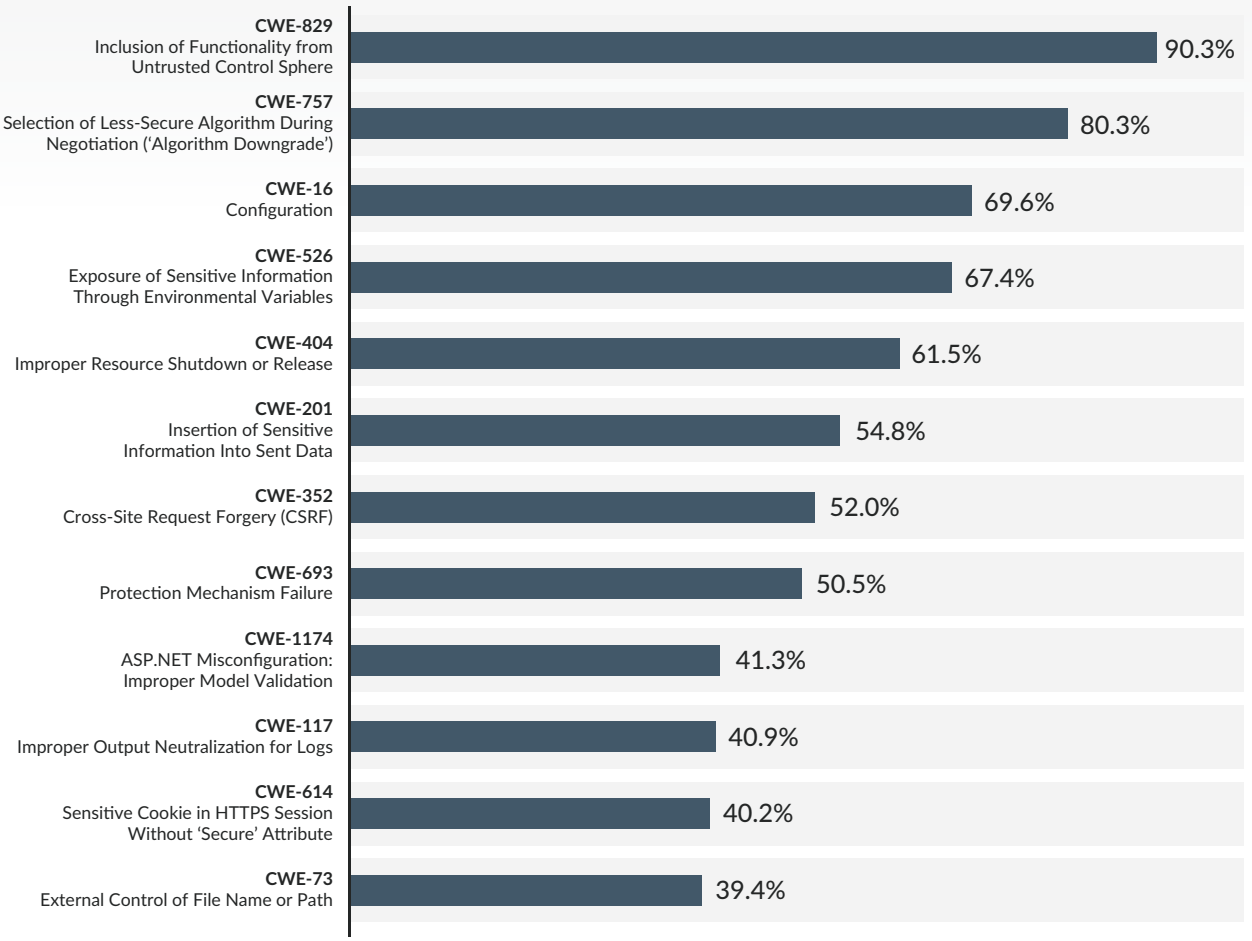


At the CWE category level, just looking at the last 12 months, we compared the percentage of applications that have introduced one or more flaw types to the percentage of applications that have introduced flaws and that have a flaw type still open. There is not a severity limit; we were just looking at types of flaws in Figure 12. For comparison between other languages, pick a flaw type you are concerned about and see where it may fall among languages. The location and density are very different among the three. The best grouping or location is down and to the left, and any progression from that lower left anchor point of zero is progressively worse.

Figure 13 shows the percentages of flaws, and we take a slightly deeper view by CWE, rather than the CWE category of flaw seen in Figure 12. We were looking for prevalence of occurrence while focusing on all flaws. To compare Figure 12 to Figure 13, you can see in .NET as with Java when certain CWE categories and CWEs are introduced, they generally are introduced a lot, have a higher percentage of being introduced, and, according to Figure 12 for .NET, a fair percentage of them (based on those up and to the right) seem to hang around for at least 12 months. Interestingly enough, .NET and Java (JavaScript had one) turned up some similar groupings between static and dynamic scans. However, we still can see that top to bottom there are flaw categories that are found in one or the other. That is the advantage of using static and dynamic together.

Note: Figure 13 is a 12-month, CWE-specific, view into the percentage chance of any flaw being introduced in the event that flaws were introduced.

Figure 13: Percent of Applications with New Flaws with a CWE in Past Year (.NET)



Weaknesses Introduced by Percent of Applications with Flaws

JavaScript

JavaScript is the stand-out in our analysis when it comes to every single category we looked at (see Figure 14), and this is the story of small percentages at the start making a big difference towards the bottom line. If we start with a quick glance at Figure 14, we can see that applications in our dataset written in JavaScript perform best of the three. If we look back at the remediation curve (Figure 7) and our seemingly non sequitur comment about "getting to 14%" still open at the end of two years, the next few sentences describe the required type of performance metrics or profile (Figure 14).

It requires:

1. That an application is reducing rather than increasing tech debt
2. An aggressive three-month close rate
3. That 50% of flaws are closed closer to the three month mark, rather than the half year (or year) mark as with other application languages

Quite simply, the remediation curve must drop early like JavaScript to get similar results.

Figure 14: Various Metrics Across Languages (JavaScript)

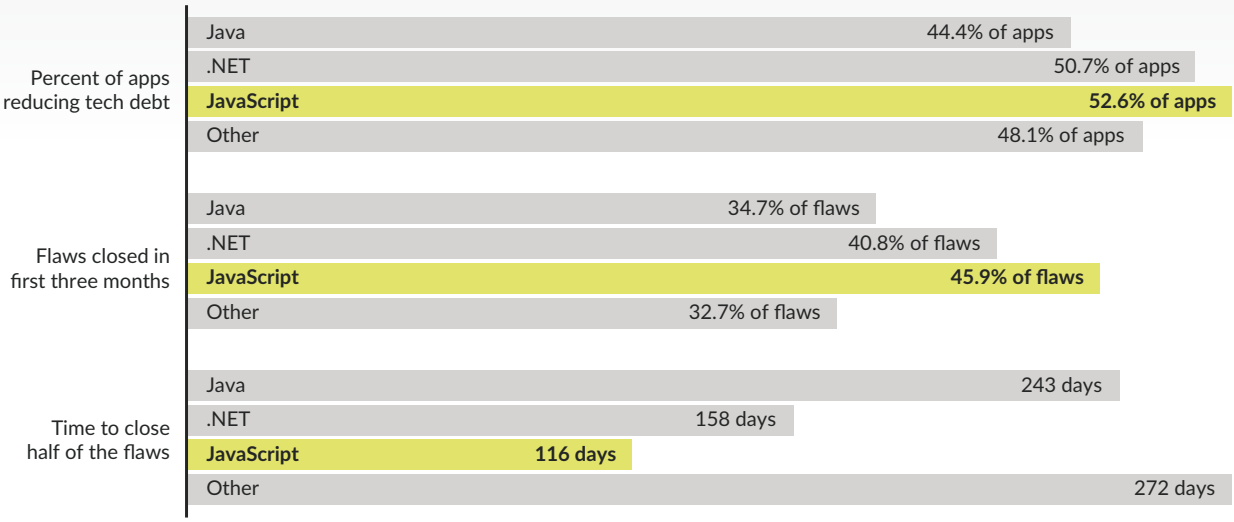
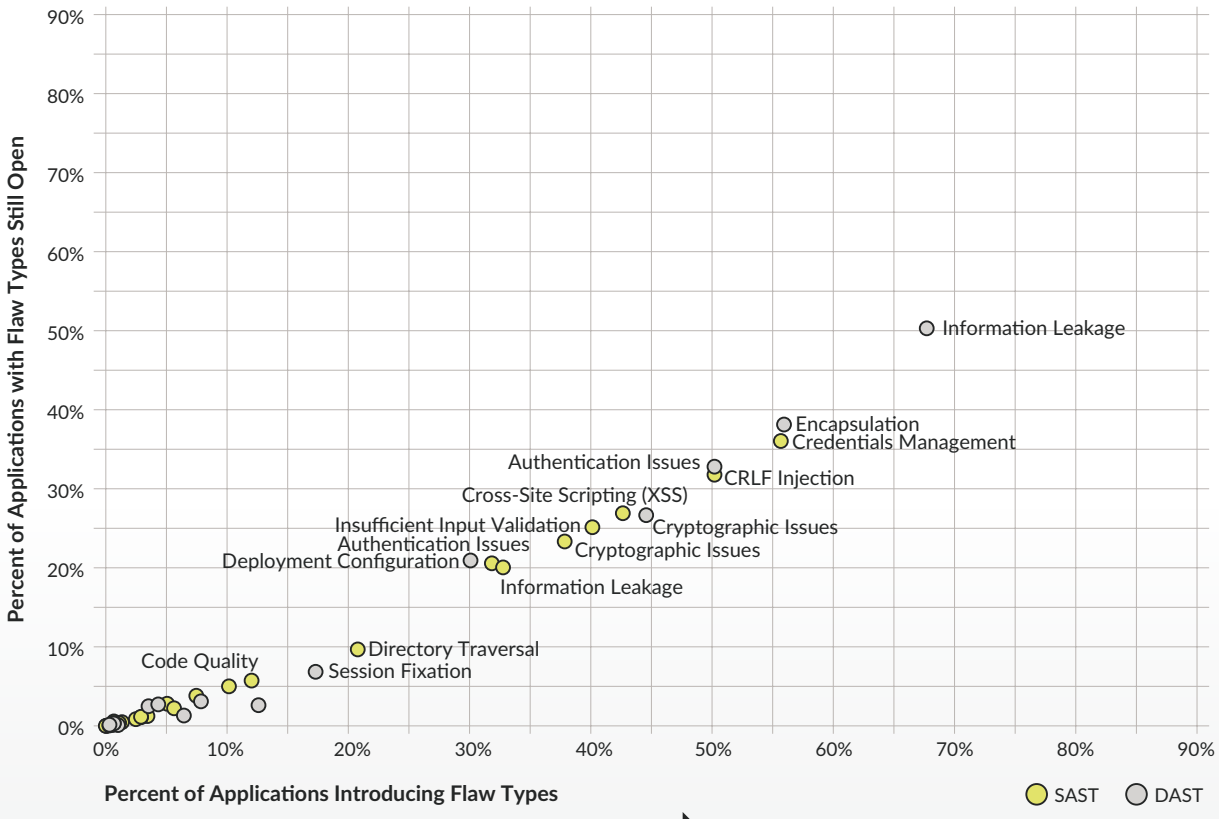


Figure 15: Introduction of Flaws vs Still Open (JavaScript)

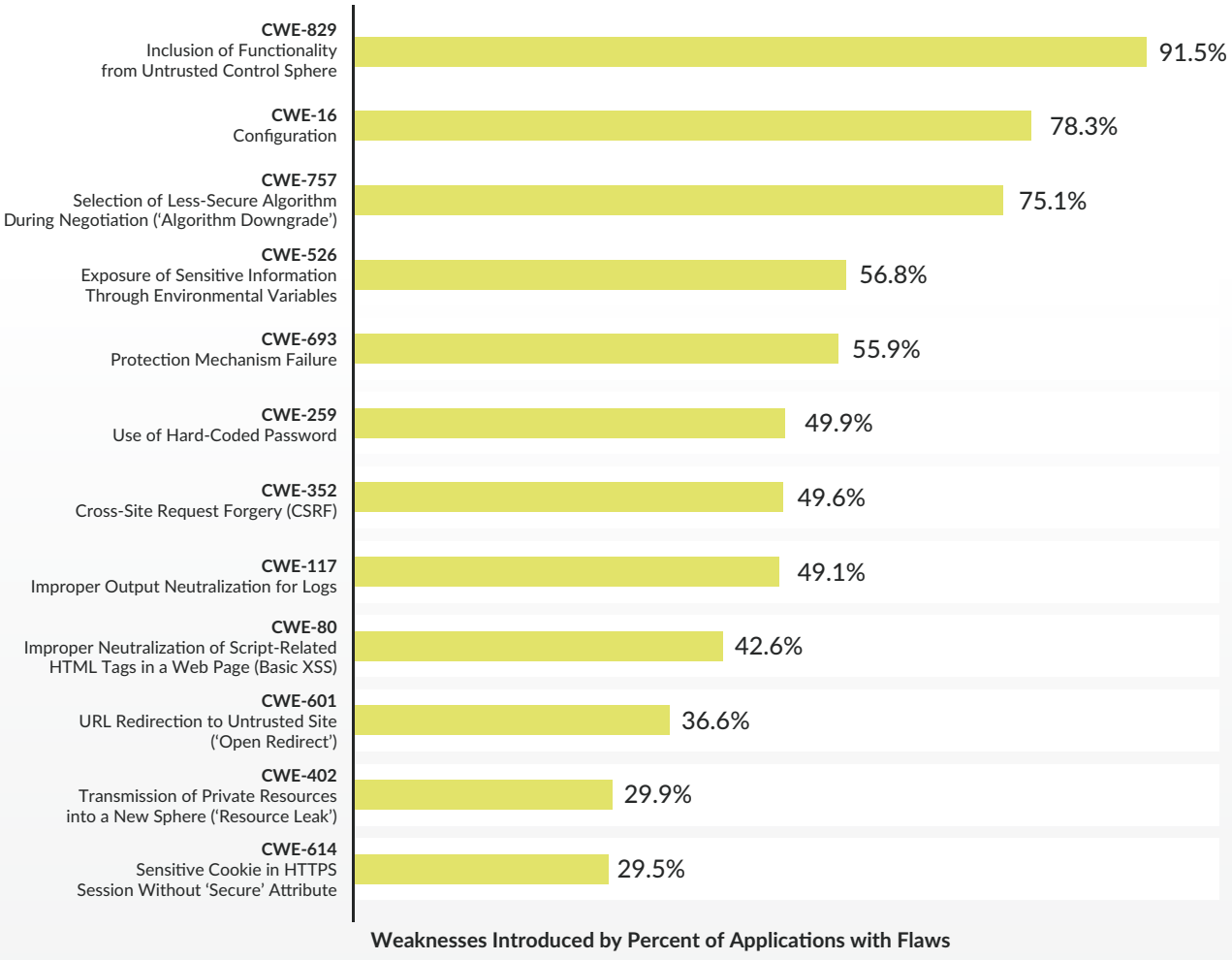


Note: We're not limiting or filtering based on the severity here (any flaw).

Take a look back at the two similar graphs to Figure 15 for .NET and Java (Figures 9 and 12), and look at the locations of the individual flaw categories. We see a very different picture with JavaScript, and we won't repeat the reason. OK, maybe we will. It is the more aggressive remediation curve. Figure 15 is at the CWE category level and is just looking at the last 12 months, comparing the percentage of applications that have introduced one or more flaw types to the percentage of applications that have that flaw type still open. Again, the lower left-hand corner is best, and note how much more compact and lower JavaScript is overall.

Even though JavaScript is the top performing language of the three we're exploring, applications in JavaScript are written by humans, and those humans are just as prone to introducing flaws (of any severity) as any other. When JavaScript developers *do* introduce certain CWEs they introduce a lot of them as well — a whole lot. Just like other languages. See Figure 16. Remember that Figures 16 and 15 are both looking at all flaws, as in previous similar plots.

Figure 16: Percent of Applications with New Flaw with a CWE in Past Year (JavaScript)



The difference and the actionable takeaway here is that the teams that own JavaScript applications address the issues more quickly, as we saw back in our remediation curve in Figure 7. That type of progress is what yielded a much more compact scatter plot with a lower percentage of still open points closer to the bottom on the Y axis of Figure 15. As a reminder, the scatter plots for JavaScript, .NET, and Java in Figures 15, 12, and 9 display a one-year view of percentage introduced and percentage still open, where lower and to the left is better. JavaScript teams continue to remediate with a sizable head start, and the remediation curve continues to drop like a stone to that 14% chance that any flaw was still open at the end of two years. That should be the target for any language.

Summary of

We are not afflicted by the Cassandra Complex where we know the future but are unable to do anything to prevent it. We know what the top flaws are for each language. We know that when flaws are introduced, they are introduced a lot. We can see that this likely affects the remediation curve and that when even one flaw is introduced it tends to stick around for a while. Compound this one flaw to many and these numbers become grim. Visualize that flaw volume and its impact. Think of the productivity of the teams stuck trying to burn down flaws after nobody clearly remembers what that part of the code does. Then of course consider the security posture of an application should flaws simply go into the backlog and either be mitigated in some fashion or accepted and forgotten. Through that volume, individual flaws may never be remediated.

We will now begin to explore what other complications, in addition to those we've already discussed, await our applications as they age, and soon what we can do to reduce new flaw introduction.

Section Four

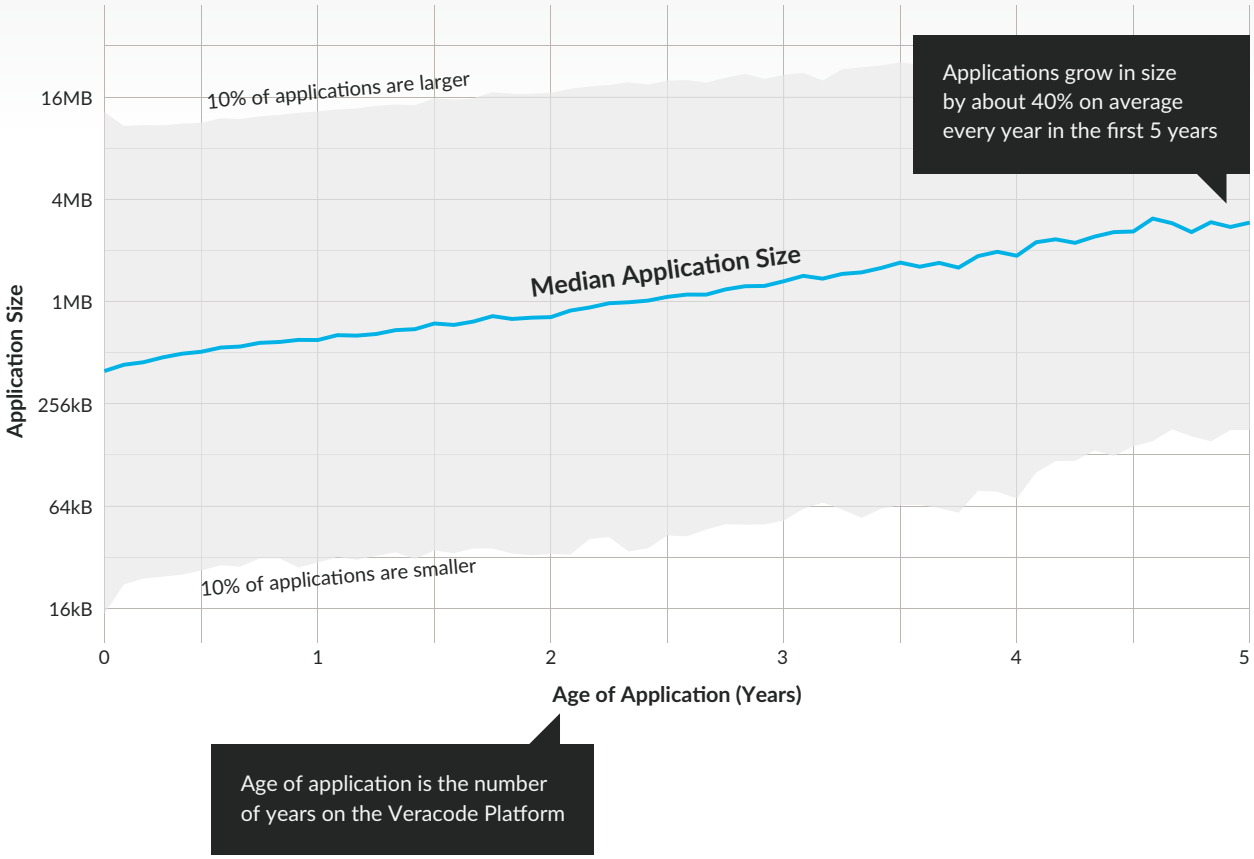
How We Got Here

Application Size

While we might think there is a correlation between application growth and the introduction of flaws, the overall picture is not that simple. Interestingly enough, the average application grows at about 40% per year regardless of its original size. In early analysis, we found an interesting parallel track of growth lines of small to large applications. We evolved that early analysis into Figure 17, which depicts the five-year size range. Growth tends to slow down after that to the 10-year mark, but flaws tend to pile up over that period.

This yields two spots in the life cycle where growth and flaw accumulation seem to be decoupled. This calls into question the idea that growth always has a direct relationship with flaw introduction. Let's have a look at how flaw introduction works. Our first graph shows that the growth range through five years is fairly constant and fairly predictable.

Figure 17: Application Size by Age of Application



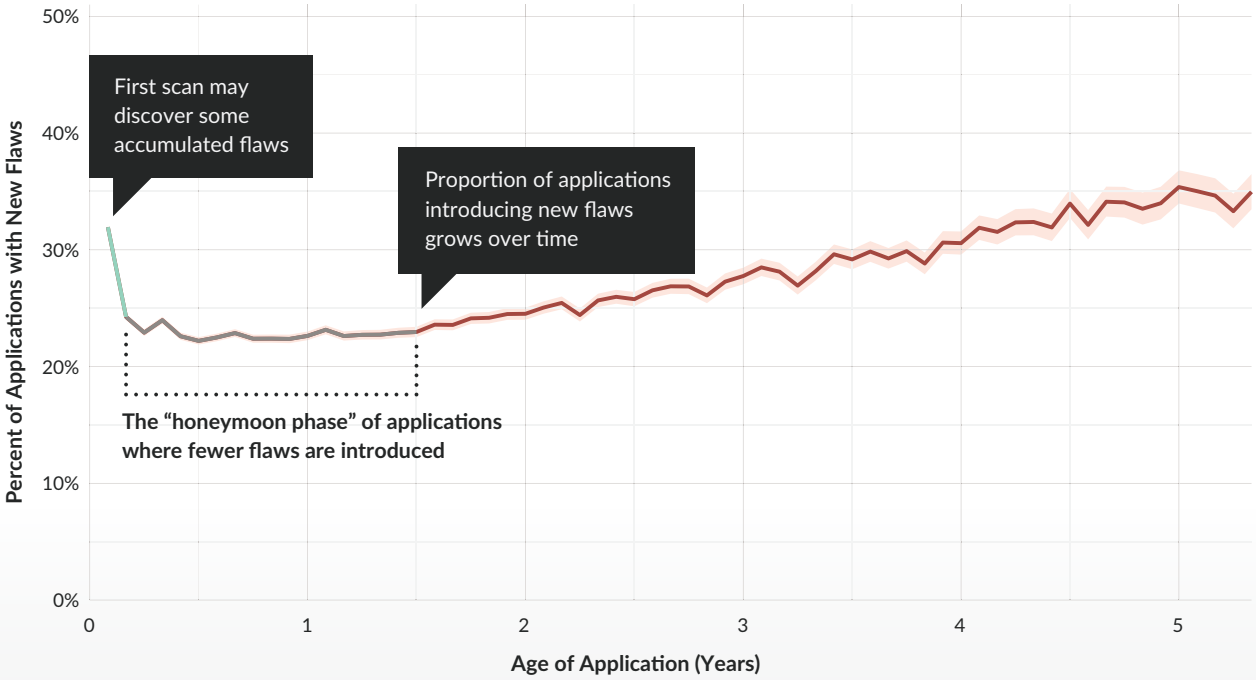
The Evolution of Applications and Their Flaws

When we combine the remediation and fix rates (definitely not 100%) with the newly introduced flaws, it's not a surprise that applications have more flaws over time. The sudden initial drop and the long tail growth in Figure 18 caught our attention. We could easily explain a new app being scanned for the first time with a load of flaw findings, followed by the teams quickly hammering the flaws and that security debt down. What happens next, though, breaks the relationship with application growth, but only temporarily. Seeing this pattern repeat in many different cuts of data, it became clear we were looking at a clearly defined application lifecycle from compiler to bitbucket.

Many small and large decisions regarding budget, timelines, priorities, focus, and — perhaps most importantly — staffing, come together here. Other factors such as staff turnover, whether from attrition, promotion, or new hires, clearly come into it as well. A lot of hands and decisions find their way into the code after launch. For better or worse, this is how application owners, business stakeholders, and teams of developers choose through action and inaction, and these are the results.

Figure 18 reflects flaws being introduced based on the age of an application as shown by a static scan. Following the initial onboarding of an application we see a rapid decrease. The application then enters what we are calling the "honeymoon period," and for the first couple of years, things are stable. Despite the fact that we know that applications grow at about 40% per year, that trend is not matched by a commensurate number of new flaws. To the contrary, close to 80% of applications do not introduce flaws at all during this early life cycle phase. This is the first period where flaw introduction and growth are decoupled. Flaw introduction begins to climb steadily at around the one-and-a-half year mark and continues to climb until year five where something of a plateau is reached.

Figure 18: Flaw Introduction by Age of Applications



Breaking Out Flaw Introduction by CWE Category and Age

Figures 19 and 20 shows newly introduced flaws broken out by CWE category. This is not a persistent or cumulative view, though we will get to that view soon. As in the consolidated timeline view introduced above in Figure 18, where we talked about the honeymoon, you can see that following the initial dip in flaw introduction the application goes through a period where flaw introduction slowly creeps up at a rate that does not match the 40% growth track from Figure 17. Application growth is fairly constant throughout the lifecycle, whereas growth in flaw introduction is delayed. Something is clearly different than the first year and a half. The fuzzy plot lines in Figure 18 (and following plots) indicate that our certainty range opens due to applications of that age declining in number.

As in the previous section where we focused on languages, you can see that not all flaw categories are equal. For the analysis in this section we have consolidated languages, grouped by the overall percentage of how often flaws are introduced (when they are introduced.) The scale on the side of each box is the percentage of applications with the timeline at the bottom.

Figure 19: Flaw Introduction Over the First 10 Years

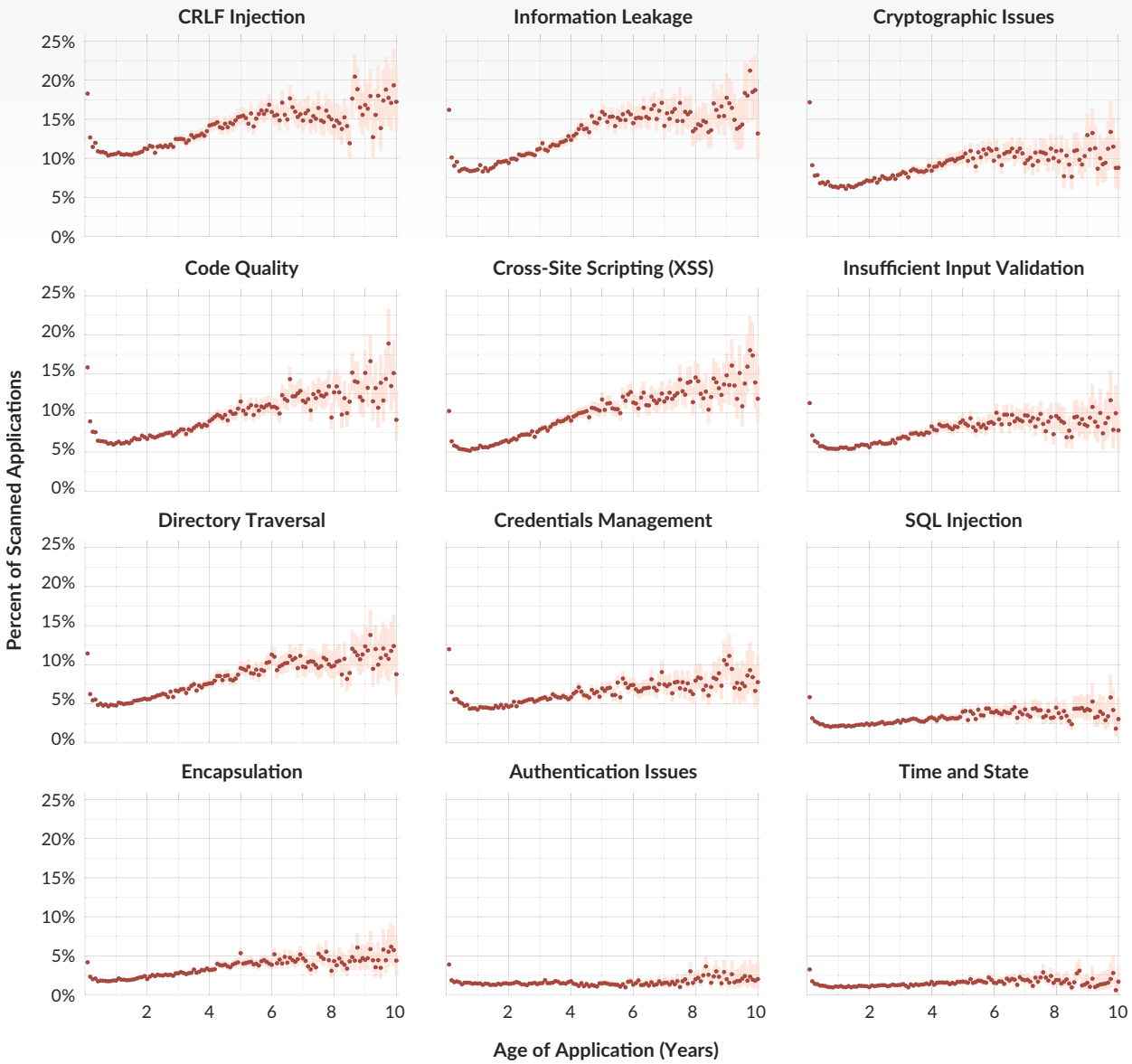
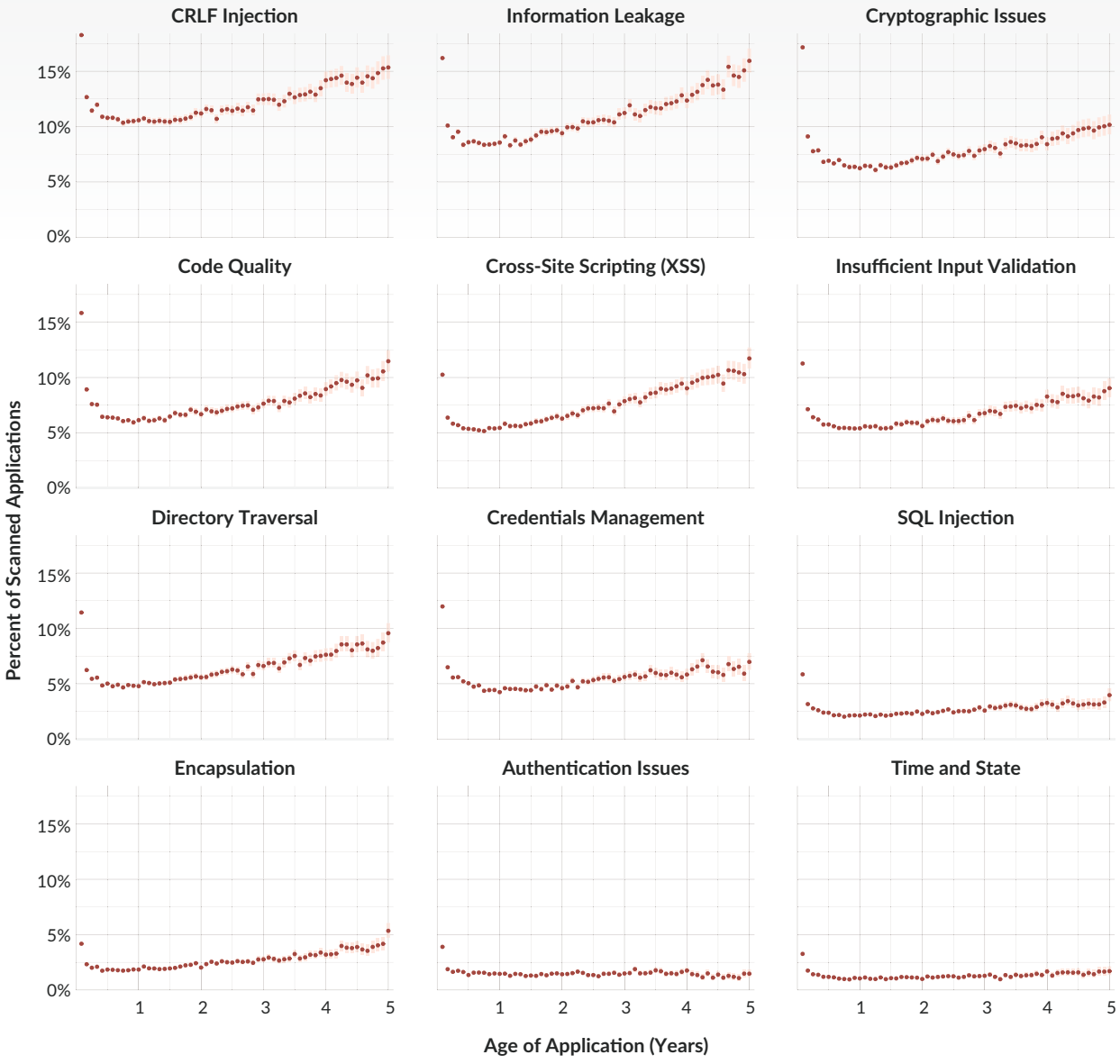
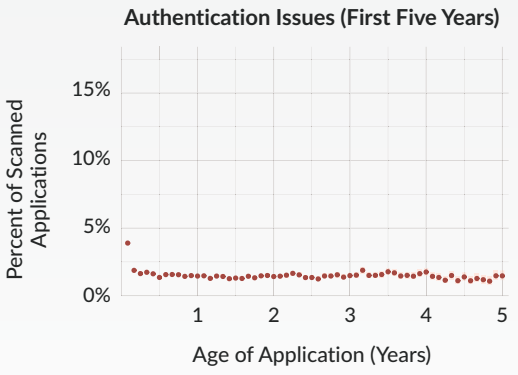
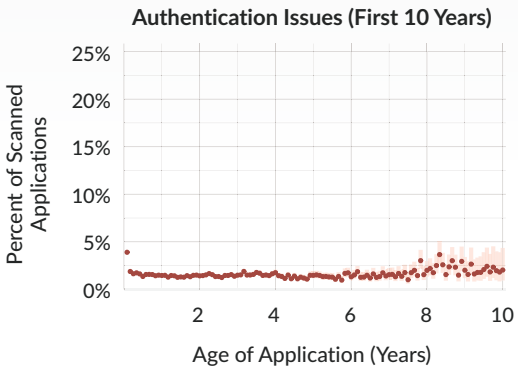


Figure 20: Flaw Introduction Over the First Five Years



We can only speculate what causes this increase in flaw introduction to occur over time, and why certain types of flaws seem more prevalent. Perhaps the fundamentals such as "Authentication Issues," often handled up front during the design of the application, are less likely to appear, whereas techniques that require knowledge of inner workings of the architectural design may be touched or updated by people who were not familiar with how it was originally done. We have a lot of theories as to why this creep occurs but can't say for sure. We would love to hear some of your ideas on this.



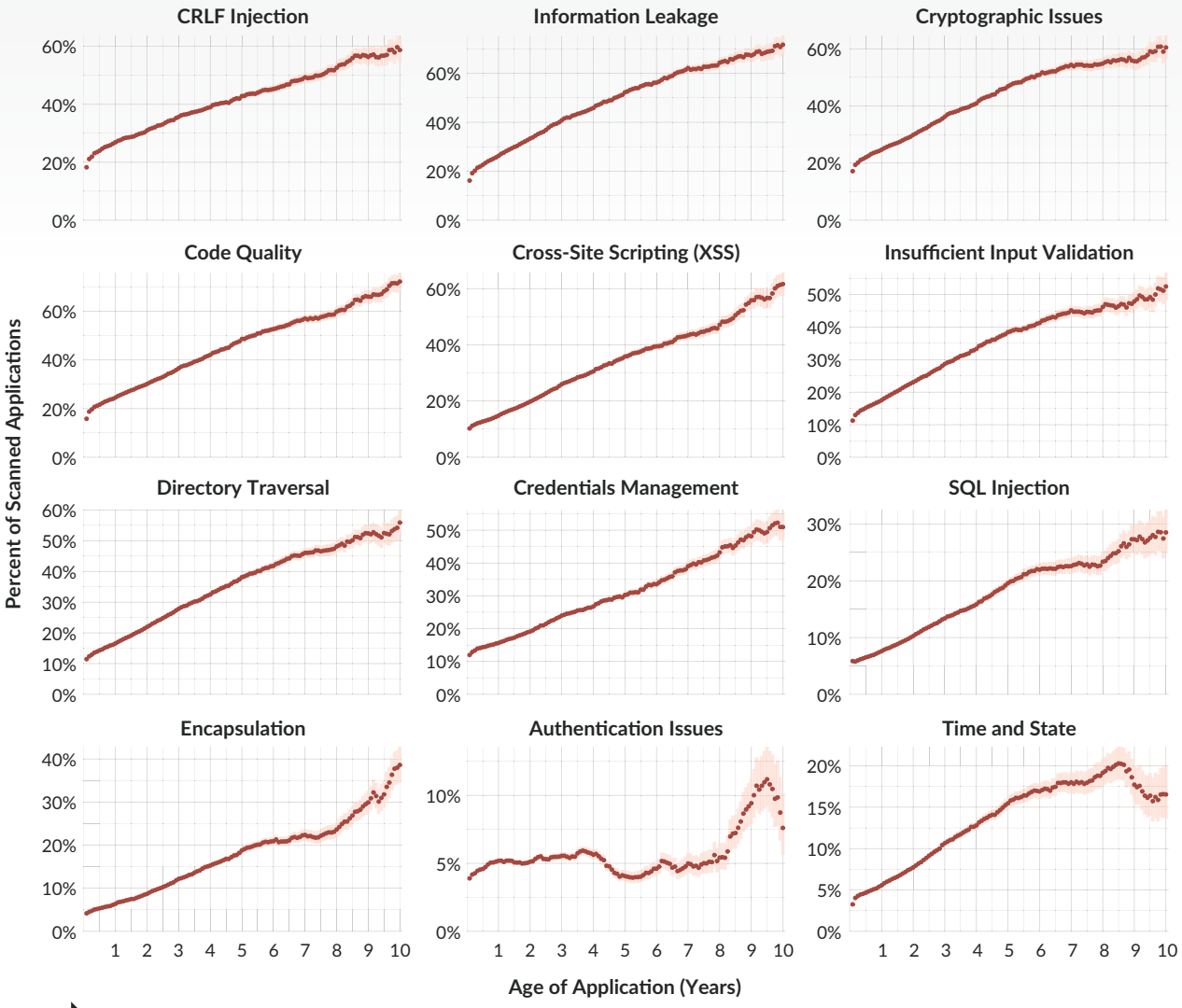


In the language-specific section above, starting with the remediation timeline through our examination of Java, .NET, and JavaScript, we mentioned that a slight increase in the rate of flaw introduction had an impact over time. Figure 21 shows that cumulative build-up. While this is an all-languages view, any application that cannot buck the trend of post-honeymoon hangover piles on flaws just like the rest of them. Some of these cumulative climbs are sadly rather dramatic over the years and create the foundation (and justification) for defining planned obsolescence.

Clearly something other than ad hoc application retirement seems to be needed, as we can see in Figures 21 and 22. The reason we are talking about planned obsolescence is that after the five-year mark, flaw categories that were fairly tame suddenly wake up and begin a period of introduction as seen in Figure 19, and then accumulation as shown in Figure 21.

The rate of flaw introduction is not shockingly different in years five to ten despite the fact that we know application growth rate slows down after year five. But that accumulation view tells a story. This is the second period where age of an application and flaw introduction are decoupled, but with vastly different results than the honeymoon period.

Figure 21: Flaw Accumulation Over the First 10 Years

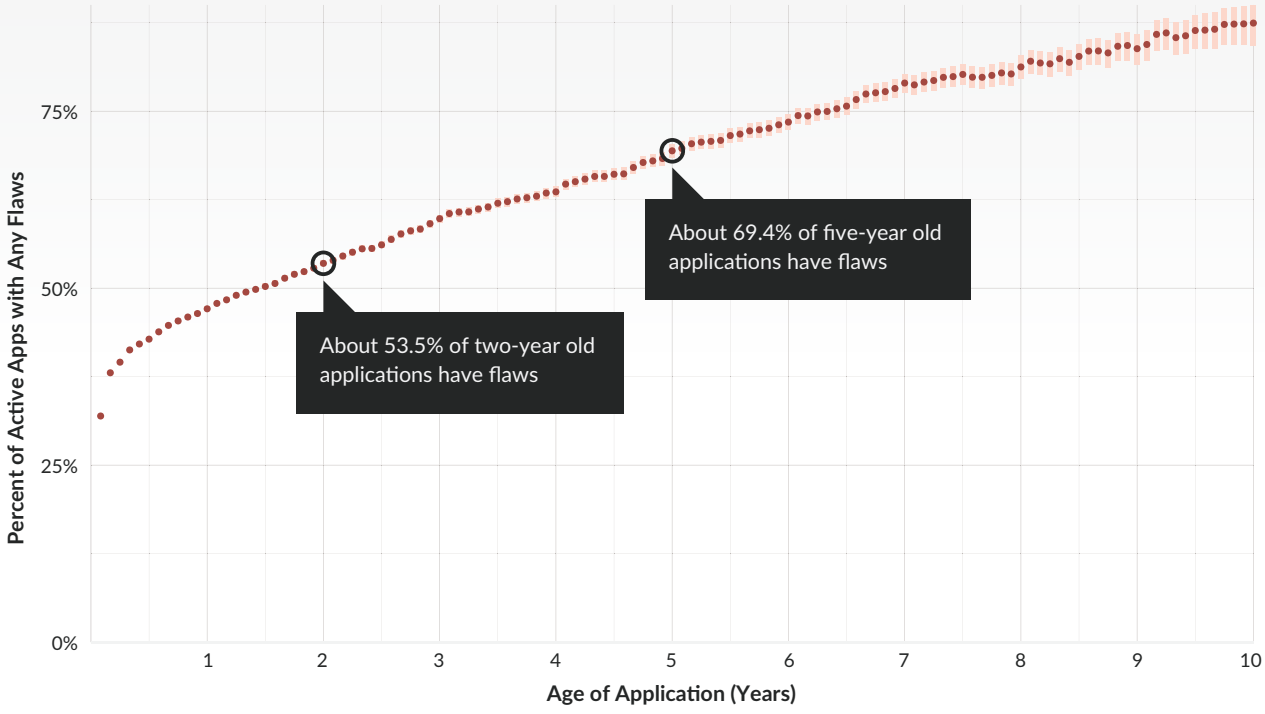


Note: The scales are very different for each CWE category top to bottom in Figure 21 since the flaws are present in different percentages of applications.

To round out our look at the introduction of flaws into applications, we wanted to have a top down view of how applications fare over the years. To get this view, we took the last scan per application per month and looked for the percentage of applications with any flaws. We can see when we take a CWE and CWE category neutral approach that the percentage of applications with flaws climbs as applications age, and the percentage climbs differently than the previous graphs, except for the CWE category "cumulative view" we just looked at in Figure 21. The percentages are different, because we've zoomed out in Figure 22 to include any flaws, giving more of a chance for inclusion.

Out of all the apps younger than one year old, fewer than half carry any flaws. We can see that by the time an application is two years old there is a 54% chance it is carrying at least one flaw and maybe many more. As those same applications mature and evolve, at least two out of three are carrying at least one flaw by the time they are four or five years old. This trend continues up to the ten-year mark where nine in ten applications have at least one flaw. A quick look back at the percentages in Figure 21 allows us to assume that based on this cumulative view there are *probably many more*.

Figure 22: Flaw Accumulation by Age of Applications



Section Five

Modeling Factors That Influence Flaw Introduction

In SoSS v11 and v12 we talked about the benefits of developer enablement, and we've talked a bit about flaw awareness so far this year. We wanted to explore other things that might help teams hit the objective of introducing fewer new flaws, so we began digging around in the data.

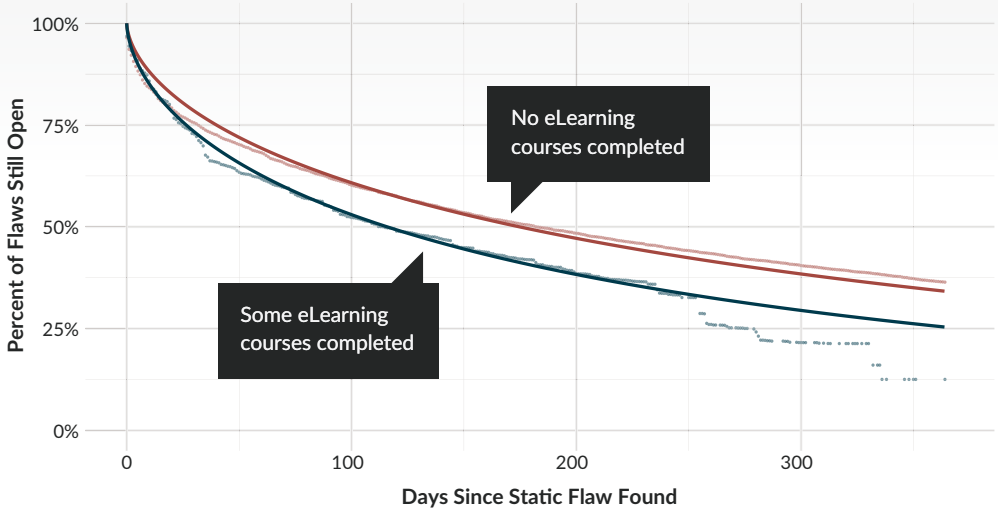
A look back at SoSS v12

Training and Flaw Remediation

In Figure 23, taken from SoSS v12, we demonstrated how completing training influences remediation time. This year we wanted to have a look at a similar group of factors to see whether we can reduce the likelihood of new flaws being introduced at all.

If we do introduce flaws, then what influences the volume of flaws being introduced?

Figure 23: Time spent learning in Veracode Security Labs



To tackle a macro trend like this, we wanted to take the largest data set possible and look for markers that positively and negatively correlate to flaw introduction. We wound up with over one-and-a-half million application scan months. Then we examined different cuts of the data on different timelines to determine whether the analysis was consistent. Each scan month represents one unique application scan per month. The application in question had to have size information available and had to have been scanned more than once to avoid random noise that would result in a single point with no trending.

We needed to account for a few observations:

1. Many applications had no flaws introduced at all, but this could be a result of not having been scanned (or no growth).
2. Since no flaws could be just no code written at all or nothing changed, size data helps determine whether there has been a change.

Other things we wanted to answer:

1. What is the chance that flaws are introduced at all?
2. Once a flaw is introduced, how widespread is it?
3. Something separates no flaws from some flaws (or a lot of them) so what is it? Age? Inactivity? Skill? Training? CI/CD automation? Scan cadence? Mercury in retrograde?

In our first examination, we found that there was about a 27% chance that an application will introduce one or more new flaws every month. Remember this number (27%) as we move through the narrative.

We know from other analysis that those applications that introduce flaws tend to introduce a lot of them. Figure 24 below shows how that 27% chance of introducing a flaw can be influenced. We wanted to understand how these factors can shift the probability a flaw will be introduced up or down and whether there was a compound effect.

The reduction or increase here is noted as a percent change in newly introduced flaws per month, Remember that the starting baseline is a 27% chance of one or more flaws being introduced. The next section all refers to Figure 24.

Note: To deal with the negative space we needed to adjust the way we were looking at this data set. We chose a hurdle model to capture when something crosses a threshold (in this case a flaw being introduced) and when it does, we examine its attributes.

Figure 24: Factors Influencing the Probability of Flaw Introduction

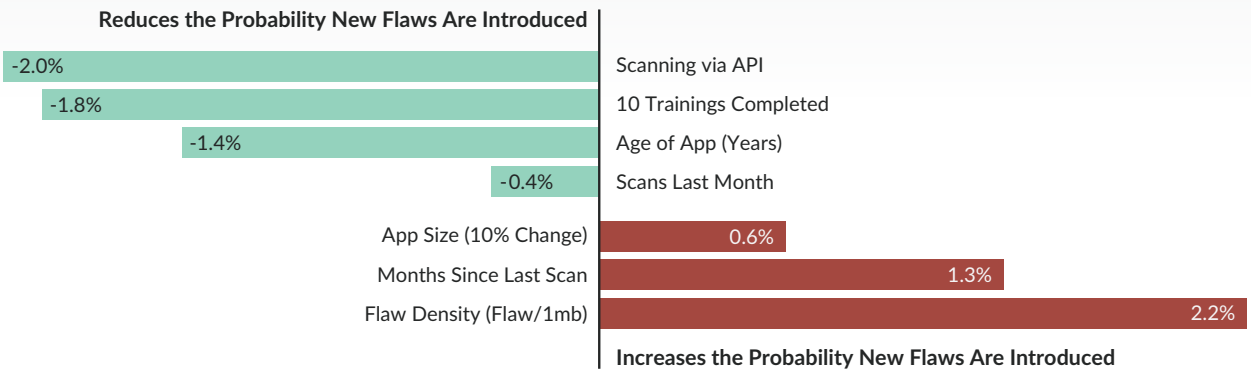


Figure 24 represents factors that influence *if* flaws are introduced. Figure 25 examines the factors that influence how many flaws are introduced *when* they are introduced. These factors can pay off or penalize twice.

Scanning Via API

When we see applications integrate code scanning into their pipeline via API scanning, we see the probability that flaws are introduced drop by 2% on average. This is the largest apparent jump, but before you get underwhelmed by a 2% drop, remember that it affects our base 27% chance and reduces to 25% (in other words, this represents a 7.4% drop from 27% to 25%). The API scan doesn't itself make things more secure, but it is an indicator of maturity. If an organization is using automation that abstracts human interaction, then we can assume it has other things in place, such as access control to the pipeline (but that is an assumption). So, we have reduced our base chance from 27% to 25%.

Training

The next stop is training, and that has a rather obvious connection to flaw introduction. Security Labs trainings exist to educate developers on the different types of flaws and how to avoid introducing them into their code. We had done some basic analysis in SoSS v12 on the impact of Security Labs and wanted to revisit it, confirm the benefit, and complete some further analysis. Figure 24 draws a line at 10 trainings (causing a 1.8% drop in flaw introduction per month on average), but 10 isn't a magic number — when developers complete any number of Security Labs, flaws are less likely to be introduced, and the more training the better!

Age of the Application

Age of the application in this model actually represents years on the Veracode Platform. For every year the application is scanned on the Veracode platform (proxy for "age"), we can expect an average drop of 1.3% in the probability of one or more flaws being introduced. Again, this is a continuous variable, so six months is half the effect on average, and so on. The drop associated with age is only possible if all else remains constant. Relatively constant flaw density and very slow growth would be required to see a net reduction. We'll touch this point again when discussing application size, where we'll see to what degree those factors cancel out any benefits from age.

Scans Last Month

Not a surprise, but if an application scanned more the month before, it was slightly less likely to introduce one or more flaws this month. We talked about this in previous years as scan cadence and decided to go with scans last month because of those results. We also divided the cadence metric so cadence can give or take. You'll see the other side of the coin in a bit.

Application Size

An increase of the application size by 10% (which is a rather large shift in size) is 0.6% more likely to introduce one or more new flaws. This number was slightly surprising, given our other analysis in which age and flaw introduction seemed decoupled in several phases of the application life cycle.

As mentioned, growth and age are not always tied, but as applications grow, they do become more complex. Given our yearly growth rate of 40% as seen in Figure 17, you see that over time the chance of flaws associated with growth (40% growth yields 2.4%) cancels out the reduction in chance that age brings with it. This explains the views we had where flaw introduction in applications creeps upwards after the first couple of years.

Months Since Last Scan

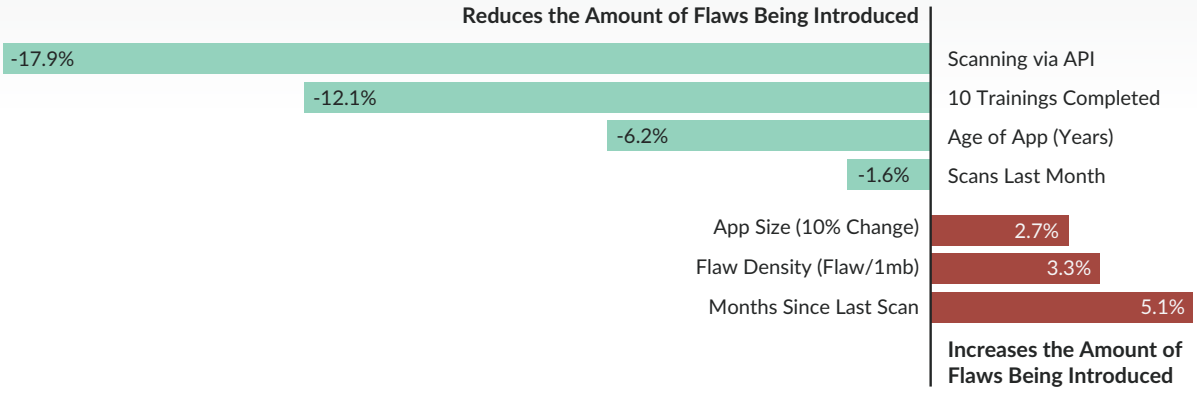
When we look at "months since last scan," the results are just common sense: the longer one waits to scan an application, the more likely it is to discover one or more flaws when the app is scanned. For every month delay in scanning, we'd expect an average increase of 1.3% in the likelihood of flaw introduction. This is the other side of the coin of regular cadence: if it's been a while, flaws quickly pile up (6 months is +7.8%).

Flaw Density

Applications with higher security debt (as measured by flaw density of one flaw per one mb of code) are more likely to introduce flaws moving forward. Going back to Figures 18, 21, and 22, you can see that applications with higher flaw density tend to be at one end or the other of their lifecycle.

Note: Figure 25 measures the flaw density prior to the application-scan-month, so the newly discovered flaws are not included in the metric.

Figure 25: Factors Influencing the Number of Flaws Introduced



If we consider the 27% probability per month that flaws will be introduced, about 73% of months in which applications have been scanned have zero flaws, which is rather significant. When flaws were introduced, the fascinating thing that happens is that the same factors that reduce the chance of introducing flaws in the first place also reduced the volume of flaws introduced. Sort of a double dip moment. In Figure 25, we can see a familiar view of the factors that reduce and increase the number of flaws if and when flaws are introduced. It is interesting that they line up neatly on the same sides as they did in Figure 24.

We return to our assumption that Scanning via API is related to programs that employ automation and limit human interaction with the pipeline. It seems that organizations that build in automation so that scans are launched via API perform better, reducing the chances of introducing flaws at all (per month). Then according to Figure 25, they also enjoy a reduction of 18% in the number of flaws introduced when they do.

Remember in Figure 24 in which we showed that Security Labs training had an increasing benefit based on the number of courses completed? When we consider Figure 25, once again 10 trainings completed isn't a waterfall type of number. Benefits in reducing the number of flaws introduced are seen even with a single training, and we again find the data is letting us know that the more training completed the better.

When it comes to the number of flaws introduced, the age of the application is an interesting factor to examine. In the discussions about Figure 24, we mentioned that growth and flaw density tend to cancel out the benefits, and we are left with a slowly ascending line that we have seen in many of the plots so far. Age is in itself a factor that reduces flaw count if everything else remains constant, but not everything does remain constant, does it? If code is written at a usual pace to yield a 40% growth rate, then the 2.7% from 10% growth in Figure 25 becomes 10.8% (+4.6% net effect when combining age and growth).

One curious difference between Figure 24 (chances of introducing) and Figure 25 (count of flaws introduced) is that months since the last scan and flaw density have flipped positions for least helpful. As we mentioned before, skipping months seems to simply increase the chances of finding something when a scan is run. The issue with a broken or extended cadence is how things add up, potentially leading to something that would yield a less optimal remediation curve. Since we are viewing how many flaws are introduced in Figure 25, the data indicates that this percentage impact is on the number of flaws introduced and it compounds at a 5.1% rate per month. Ouch.

Section Six

Fragility of Open Source

Given some of the recent focus on the Software Bill of Materials (SBOM), we thought this would be a good year to examine other factors that can introduce flaws. To a large degree this is the great unknown of open source.

Developers build their applications using libraries completely outside their control, establishing dependencies for basic functions that an application needs. Some of these dependencies then introduce further dependencies and things move fast out there. This follows along on our top three discussion about flaw introduction, tech debt accumulation, and lifecycle management. For this report, we took some first steps to analyze and profile open-source repositories. Not reinventing the wheel has obvious rewards, but open source is not free. It cedes control and introduces external dependencies.

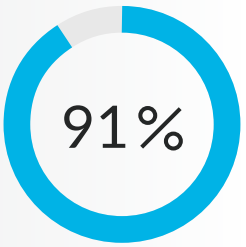
For each publicly disclosed vulnerability, we can only speculate how many undisclosed and undiscovered vulnerabilities there really are waiting to hit the news and launch us all into the next panic. Aside from scattergun technical controls and herculean response tactics, what steps can organizations take to reduce their exposure and improve their response if they are affected?

Log4j[4] was a wakeup call, or at least it should have been. Certainly it tested the response and readiness capabilities for many teams. The nature of the response depended on the level of visibility an organization had into the application composition and exposure. Whatever the end of 2021 was, history tells us we will get to do it again.

How do you determine what is safe or not safe to use? Is it by repository? Repository owner type? Contribution cadence? Some mystical reputation score calculated through some opaque process? The answer is not as simple as you may think, and anyone who thinks it is will hopefully see why soon enough.

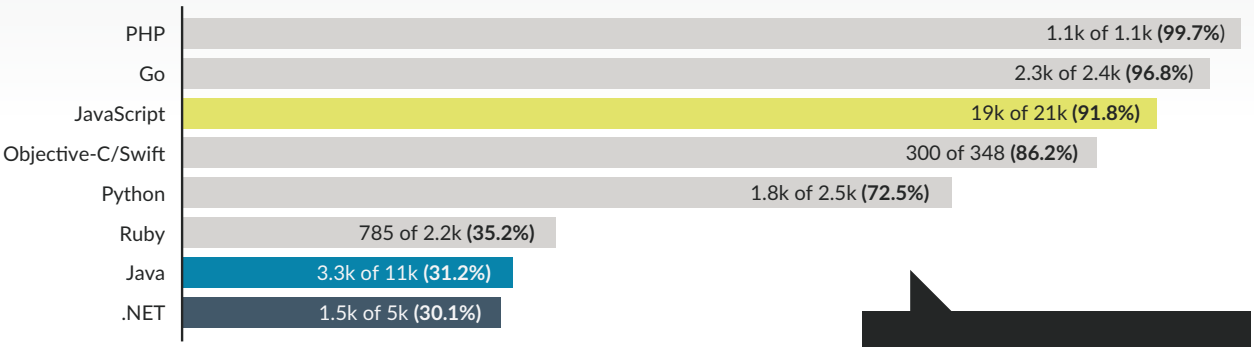
[4] For Veracode Log4j resources have a look here: www.veracode.com/log4j-vulnerability-resources

We completed an analysis of those GitHub repositories scanned with composition analysis to begin to peer into what is out there. Out of millions of repositories hosted publicly on GitHub, we identified 29,783 that were actively used in production code by Veracode customers. This means this is not just a dragnet analysis looking for what we might arbitrarily think is malicious or filtering using stars or some other method for inclusion/exclusion. The repos in this analysis are actually included in production code. What we are measuring is the fragility of legitimate packages.



of third-party libraries are hosted in GitHub

Figure 26: Percent of Unique Libraries in Use with a GitHub Repository (by Language)



Percent of Distinct Libraries with Known Repositories

Figure 26 breaks down the proportion of identified libraries in use that we have correlated to a public repository on GitHub. For example, JavaScript applications tend to include quite a few third-party libraries, and out of the 21,000 unique libraries we found in use in actual applications. Out of the libraries in use in JavaScript applications, we were able to correlate 91.8% of those (about 19,000) to a public GitHub Repository. Connecting to the repository enables us to collect more information about each library, including licenses in use, how long the library has been around, how actively it's maintained and how many developers are contributing to the code base.

Note: 51.8% of .NET libraries have a repository URL just listed as "dot.net," and 7.5% have just "asp.net" listed, making it difficult to talk about .NET libraries.

To do any analysis we would have to look at the specific details of contributors to figure out which ones are MSFT employees and which ones are community. We decided to stick to GitHub for this year.

Now that we have established which libraries in use by applications have a valid and public repository, we wanted to look at the demographics. Repositories can either be owned by "user" or "organization" in GitHub. Within GitHub, repositories with "organization" as the owner have other functionality and are generally used on more serious/mature projects; this gave us the overall account demographics within GitHub prior to looking at popularity.

Of these, we then looked at the 100 most-used packages to understand the account demographics and the plot above Figure 27 gave way to Figure 28. In Figure 28 we see about 75% of the 100 most-used packages are owned by organizations, and a smaller but still substantial number are indicated as owned by "user" accounts.

Figure 27:
Owner Types Across
All Repositories

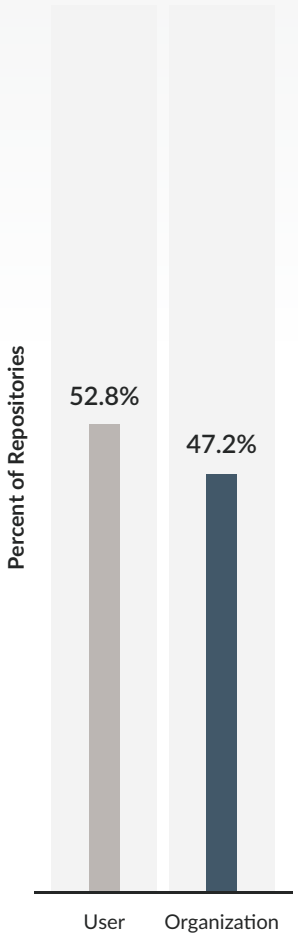
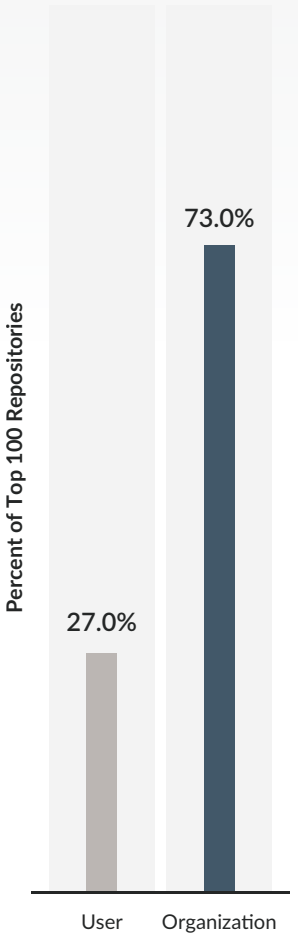


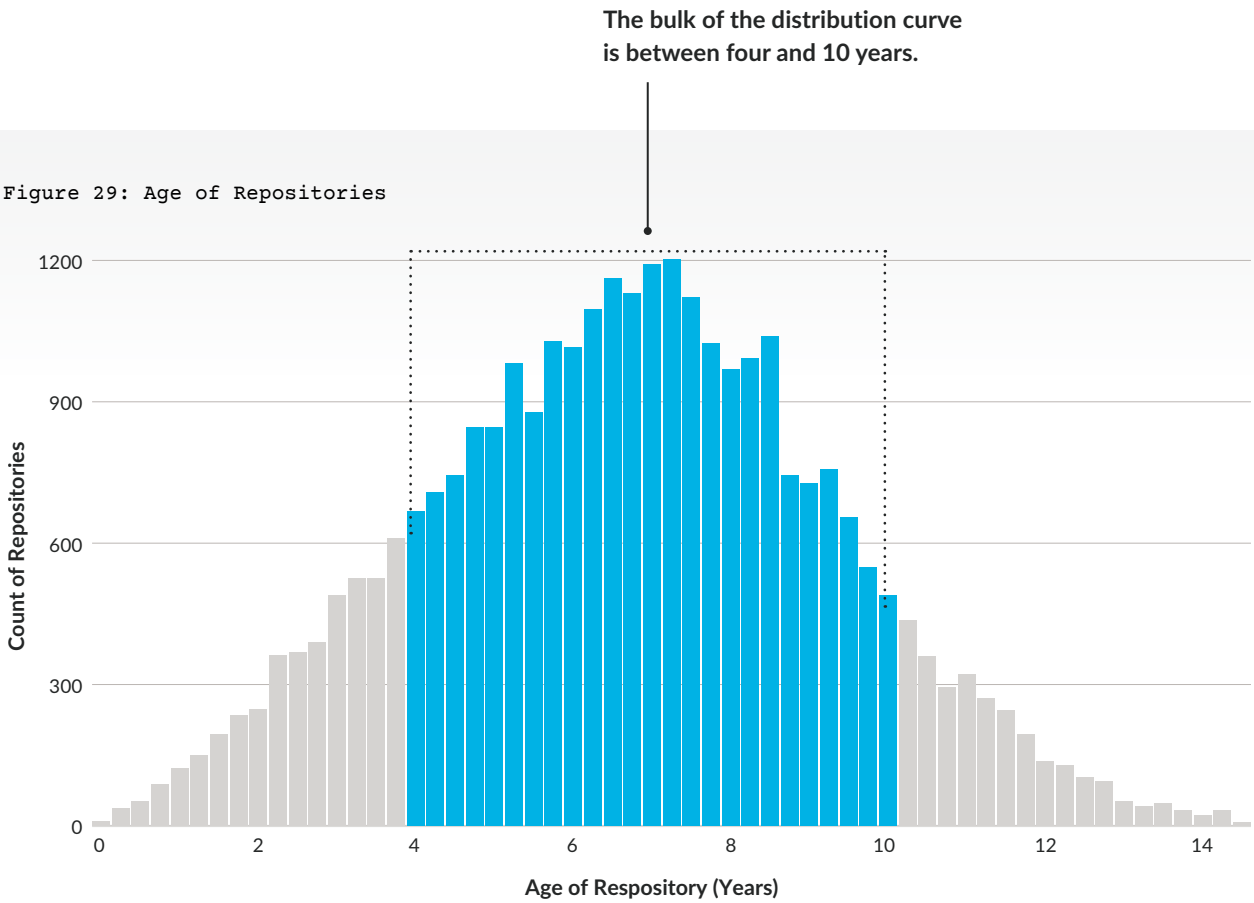
Figure 28:
Owner Types Across
Top 100 Repositories



Examination of the dates in the base information for each repository gives us a distribution that shows the age of our sample by creation date in Figure 29 below. The bulk of the distribution curve in our analysis is between four and 10 years. To be precise: 50% of libraries are between five years and eight years, six months. On the outside edges, 10% are less than three years, four months and 10% are older than 10 years, three months.

We offer this up purely to be informational. Before we collected the data, we weren't exactly sure how common 10-year old libraries were, nor what would be considered a "young" library. But once we created Figure 29, it helped us see the almost bell-shaped curve peaking around seven years. As a reminder, these are the valid repositories and their libraries that have appeared in software composition analysis scans.

Figure 29: Age of Repositories



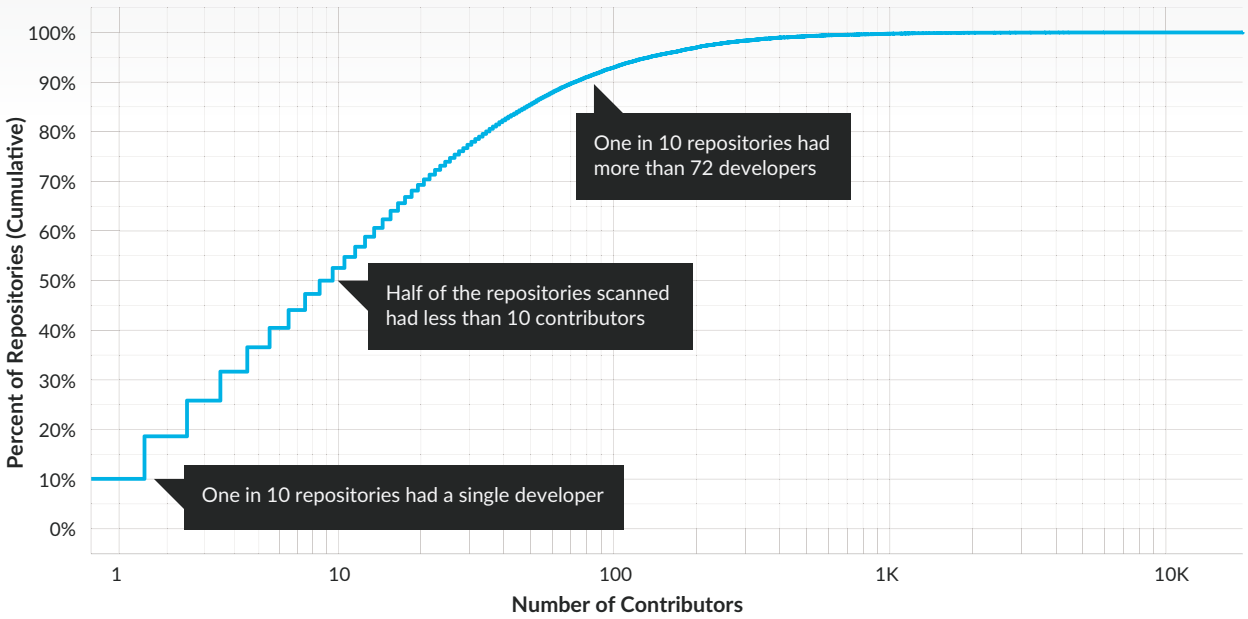
Is there an impact of Open Source on quality?

Now that we have established the demographics for the valid repositories in GitHub, we started examining contribution cadence and project team size (Figures 30 and 31). That's when our research led us into a mostly philosophical area where we are going to have to accept that we don't know the answer (this year).

Are repository demographics and low commit cadence representative of potential problems? Is the last commit indicative of inactivity? Are either of these things cause for concern? We illuminated a few things, and more questions came up. We feel our work here raises some real questions and challenges the conventional wisdom around pre-calculated "confidence scores." This research should spur some healthy debate on what constitutes a healthy or safe library or repository.

The first thing we had to accept as imprecise was the number of contributors. This is a bit tricky to count. Why? Because some developers may commit from different email addresses, they will be counted twice. Our estimate is that the following graph (Figure 30) could be over-counting by maybe 10%. In any case, we can see that about half of repositories have 10 or fewer contributors, about one in 10 repositories had a single contributor, and about one in four repositories had more than 25 contributors listed.

Figure 30: Number of Contributors per Repository



Once we identified our distribution of contributor count by repository, we had a look at when the last commit was on each library in Figure 31.

The Bus Test

In terms of business continuity management / disaster recovery (BCM/DR) when risk comes up, the bus test comes out. It goes like this: how many people must be hit by a bus in order to stop a project completely? That's your bus test number. If you find the bus test to be a distastefully morbid image, you can substitute it with other paradigms for the same results. Vacation. Attrition. Promotion. Alien abduction. Pick your project poison. When dealing with the bus test, lower numbers are bad, since lack of fault tolerance becomes the effective risk to the project.

Figure 31: Years Since Last Repository Commit

And the award for most contributors goes to:

1

"DefinitelyTyped"

Analysis through the API showed this repository had 18,948 contributors (their website is slightly out of date and claims 15.5K). This top spot honor is oddly ironic.

github.com/DefinitelyTyped/DefinitelyTyped

2

"Ruby on Rails"

With just over 5,000 contributors through our API analysis.

github.com/rails/rails

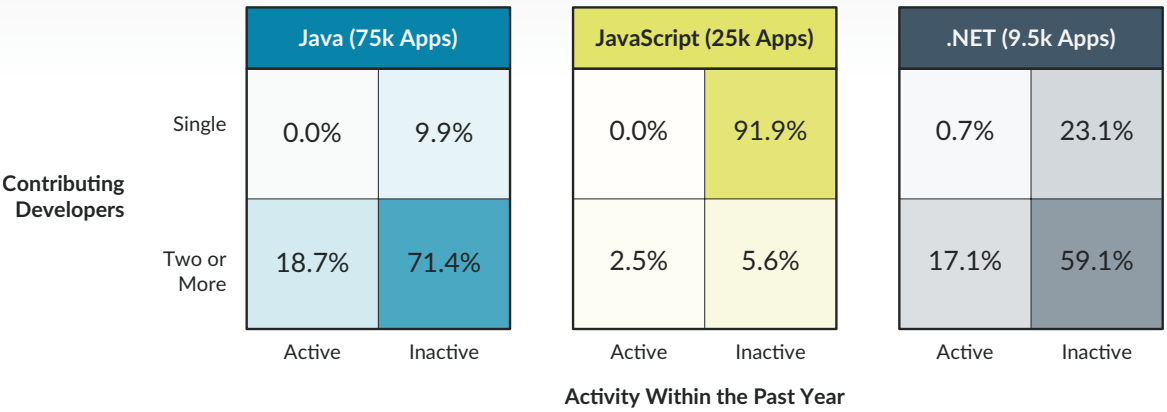
Figure 32: License Types from Libraries in Use with Repositories in GitHub

Note: Not paying attention to licensing can introduce risk to a commercial application. This issue can come up at inopportune moments. Such as when due diligence is performed during a merger or acquisition. No license means no usage rights, and that is why it is important not to ignore this.

Another example is the commercial implications of inclusion of a single GPL library. Under GNU GPL that inclusion means the entire program must be covered under a GPL compatible license. Licensing can be a real blind spot, so it is advisable to get a handle on it.

By looking at each of the applications on which Veracode has done SCA scans, and knowing the third-party libraries in each, we can count applications that depend on at least one library with either a single point of failure (single developer) or that aren't being actively maintained (no code commits for over a year). When the graphic below (Figure 33) appeared as a combination of Figure 30 and 31 we knew we'd hit something, but what?

Figure 33: Percent of Applications by Developers and Activity (by Language)



For starters, given the "lowest of the group" flaw density in JavaScript, what does it mean when 92% of applications use at least one library maintained by a single contributor with zero contributions in the last year? Are these apps more brittle or fragile due to these seemingly absentee landlord dependencies or are they fine?

We have accepted that we don't know what the practical implications of single or infrequent are. Perhaps a single developer wrote some great bomb-proof code years ago, but since it is epic code and does exactly what it says on the tin, that developer moved along. To carry this further, if no flaws have been reported on it, then no updates are needed. Right? Maybe the only thing that needs to be done in the next 16 years is an update before January 2038.

At this point, our theoretical semi-perfect code and some abandoned bug-infested project cannot be distinguished by this method, and they would both be categorized as either infrequent and single developer (or both). To complicate matters, in the last few years supply chain poisoning and repository typosquatters[5] are beginning to use a successful recipe that spammers came up with in the mid 2000s. They employ assembly line tactics with increasing volume to evade signature-based detection and might even at first host code that hashes legit (pending their "update"). This arms race has just begun. Contributions to adjacent typosquatter repositories may very well be within the last year, but rather than an indication that the repository (or the libraries within) is active and good, those are going to be bad.

Taking it further and thinking about application architecture, do we want to know about these types of repositories filled with dormant libraries? What if your application requires a component that cannot be easily substituted? Suddenly this obscure component makes the front-page news and the creator has long since disappeared. This gets worse if you don't understand exactly how or why the third-party code works, and the author of that third-party library is grabbed by a passing flying saucer. That's also a potential licensing problem too, in which suddenly a cease and desist or an audit turns something up and the library can no longer be used (see Figure 28). At what point, then, does the aforementioned "bus test" become relevant, and what steps should you take to mitigate it? These are all pitfalls of modern application architecture when we reuse code created by third parties.

Hopefully by now you can see that this simply cannot be hit with a wide brush. The more we talked about it internally, the more we decided that this is a discussion that needs more research, and we'll share when that's ready. There are clearly two sides to the story here and we look forward to continuing our research.

[5] FDARKReading. Novel npm Timing Attack Allows Corporate Targeting. www.darkreading.com/application-security/novel-npm-timing-attack-allows-corporate-targeting

Bleeping Computer. October 23. Software downloads for private citizens — 200 ongoing attacks (more recent 600 domains) www.bleepingcomputer.com/news/security/typosquat-campaign-mimics-27-brands-to-push-windows-android-malware

Recommendations for Open Source

In the interim, are there any steps you can take to reduce the risk posed by open-source libraries?

Some common-sense suggestions include:

1. **Prioritize your efforts by looking at vulnerable methods analyses and the existence of public exploits.**

   Consider that it might take weeks or months for a vulnerability to appear in the National Vulnerability Database (NVD) and how much advance warning means to your team. Any SCA solution in use should leverage multiple sources for flaws (not just NVD) to give advanced warning to teams. Once a vulnerability is disclosed (even via unofficial channels), it's a race against the clock to when active exploitation begins. It might take weeks to months for a vulnerability to appear in the NVD, and by then, in-the-wild exploits may have already begun.

2. **Set organizational policy around what vulnerabilities you're willing to accept, understanding that different applications will have different risk profiles and risk tolerances.**

   It's more sustainable to enforce policy programmatically than trying to maintain an internal repo of "safe" libraries, which can be too resource intensive for all but the most well-staffed organizations.

3. **Consider ways to reduce your third-party dependencies.**

   Think back to 2016 and the left-pad package[6] that was 11 lines long. For simple "shortcut" code that is included by default, ask why it is included. Especially if it introduces new dependencies that are required in order for your code to work. If developers can write the code easily, and it's low risk to do so, then try to reduce dependencies that can introduce fragility, or worse, increase your attack surface. It is worth calling out that no one should be trying to roll their own crypto!

[6] The owner of left-pad pulled the package after a dispute over naming of another unrelated package called Kik. The results of the take-down caused a major dependency breakdown and service interruptions. www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program

Section Seven

An Ounce of Prevention is Worth a Pound of Cure

Concrete Steps to Improve Your Application Security Program for 2023 and Beyond

When it comes to application security programs, what separates the middle of the pack from the front (or the back) of the maturity curve? Seemingly small percentages that translate into larger differences over time. Factors that can be influenced. It's time to put what we've learned in this report into practice.

Step 1

Steepen the Curve

As a result of this research, our first piece of guidance has its foundation in the Remediation Curve (Figure 7) where we observed the clear differences in flaw profiles between Java, .NET, and JavaScript. Additional focus on flaw remediation seen in JavaScript returns quantifiable improvements over the lifecycle of an application, and its effective security posture.

Quite simply the remediation curve has to fall early and fall faster, since, by the time an application is two years old, we see applications accumulate flaws. It is clear that something happens to the application or to the groups developing them. Whether increasing application complexity from years of steady growth or diminishing focus on production applications over time, this familiar pattern of an upwards slant is clear to see in Figures 18 through 22. We do know that by the time an application is 10 years old there is a 90% chance that it has at least one flaw.

Quite simply the remediation curve must fall early and fall faster. Teams must take steps to reduce the factors that result in accumulation of flaws as our applications go through their lifecycle.

Step 2

Prioritize Automation and Developer Training

In SoSS v11 we examined the set of factors that contribute to remediation, and this time around we broke many of the same factors out and examined how they help prevent flaws from being introduced in the first place. The good news is that things like scan cadence, scanning via API, and developer security training hold up as beneficial for both flaw introduction and remediation.

Developer awareness of which categories of CWE (and even individual CWEs) are introduced is a good starting spot for creating targeted training programs. We have presented this data on a per language basis so that teams can appropriately prioritize to get the most bang for the buck.

Refer to:

Java (Figures 8, 9, and 10)

.NET (Figures 11, 12, and 13)

JavaScript (Figures 14, 15, and 16)

Not introducing flaws in the first place helps in a big way, and we saw this go one step further this year. Those same factors that affect introduction and remediation also impact the number of flaws introduced in those months where flaws are introduced. To visualize the impact of training please refer to Figure 23, and the overall positive and negative influencers in Figures 24 and 25. Automation might be a work in progress for some teams, but training is within reach and should be a priority given its benefits. For those teams that want a quicker return on the time investment, consider targeting the top flaws and CWEs for the languages in use. In short, we've given solid guidance for how to reduce the number of flaws introduced in the first place.

We strongly recommend developer training. We see that it is effective in avoiding introducing flaws and reducing the count when flaws are introduced. In SoSS v12 we saw that companies taking at least one Veracode Security Labs course reduce the time to remediate 50% of flaws by 35%.

Step 3

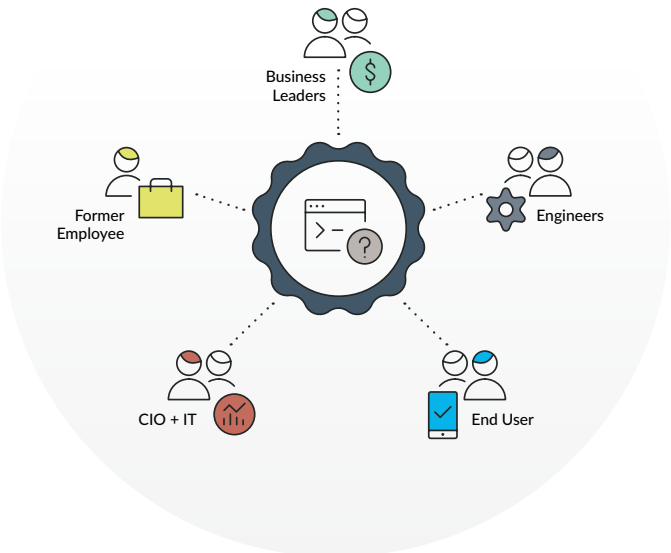
Establish Application Lifecycle Management

It's often an uncomfortable organizational discussion.

Who owns an application?

- The business leaders that feel they are the primary stakeholder?
- The engineering group that develops and maintains the application?
- The end users that the application serves?
- The CIO and IT that deal with the operations, data, and migrations?
- Or the person who is called the application owner who left two years ago?

Oops. Yep.



The full discussion of application lifecycle management is beyond the scope of this document, but the data we have presented on flaw accumulation over time makes it something we think needs to be considered to deliver a future-ready program.

Don't get hung up on the daunting project of creating an exhaustively complete inventory of applications and owners up front. Get going when you have the required information for a few applications, and build your pipeline. If you try to complete an inventory before progressing further with efforts to establish lifecycle controls, you'll never finish. Owners change, developers come and go, business stakeholder priorities change, and that will complicate any nascent efforts to gain insight into the flaw introduction root cause analysis.

Finding the owner or determining the purpose of an application is partially the problem. Deciding that an application is needed is the easy part, but deciding how long that application should be around might be something to which some more attention is given. We can't make these decisions for you, but we can call attention to the upward trend in flaw accumulation that is evident after the first year and a half (Figures 18, 21, and 22) that indicates that something is happening. Many decisions and potentially also inaction come to bear on the application. If they are not accounted for, then should we accept that older applications are inevitably going to introduce flaws when we touch them and then accumulate those flaws?

Investigate what the supportability and quality phases look like in your organization to determine why they occur. Once you do, the discussion of change management, resource allocation, or organizational controls can occur.

Risk appetite or tolerance might also play into what is acceptable given what speed is required, so long as all are aware of what is accumulating. Those pushing for speed need to see Figures 18, 19, 20, 21, and 22 but so do folks that say that resources are scarce. As we mentioned before, we understand that complete rewrites are sometimes unacceptably expensive in terms of resources so examining if an application is still fit for purpose after five years might be a better way of approaching things. Initial discussions could lead to planned obsolescence for some applications and some form of review of the processes and quality control measures involved in continuous product engineering. These ideas to improve supportability over time lead us back to the idea of introducing and maturing the practice of application lifecycle management.

Next Steps:

Now that you've seen the predictable patterns of flaw introduction, realized the fragility of open-source ecosystems and delved with us into the hard data to understand what factors go into flaw introduction, faster remediation, and lower security debt regardless of the language, we hope you are more confident in taking our recommended necessary steps to improve your application security program in 2023 and beyond.

We'd love to continue to help you on your journey.

If you'd like to leverage Veracode's all-in-one modern cloud application security solution, or even just learn how to unify and better train your security and development teams, please reach out to our team or schedule a demo with one of our experts.

Schedule a Demo

Section Eight

Appendix

Methodology

The data represents large and small companies, commercial software suppliers, software outsourcers, and open-source projects.[7] 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 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 levels — predefined security policies based on the NIST definitions of assurance levels).

This research draws from the following:

759,445
applications that used all scan types

1,262,147
dynamic analysis scans

7,522,989
static analysis scans

18,473,203
software composition analysis scans

All those scans produced:

86 million
raw static findings

3.7 million
raw dynamic findings

8.5 million
raw software composition analysis findings

[7] 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 composition analysis presented in the report.

A Note on Mass Closures

While preparing the data for our analysis for a previous volume in this research series, 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, implying developers may be scanning entire filesystems, previous or otherwise invalid branches, and when they would rescan the valid code, every finding not found again would be marked as remediated.

These "ghost-findings" had a large effect: The top one-tenth of one percent of the scans (0.1 percent) accounted for almost a quarter of all the closed findings.

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

VERACODE

Copyright © 2023 Veracode, Inc. All rights reserved. Veracode is a registered trademark of Veracode, Inc. in the United States and may be registered in certain other jurisdictions. All other product names, brands or logos belong to their respective holders. All other trademarks cited herein are property of their respective owners.