STATE OF SOFTWARE SECURITY

Open Source Edition

VERACODE

VERACODE

Contents

Introduction + Key Findings 02

Chapter 1 05 **Open Source Library Usage**
07 Open source library usage is very skewed
08 Core libraries are almost always included
10 Let's talk about JavaScript
13 Libraries — very numerous — many versions — wow
14 Dependency types

Chapter 2 16 **Flaws in Open Source Libraries**
18 Flaw prevalence in libraries by language
20 Are certain types of flaws more prevalent than others?
22 Prevalence of the OWASP top flaws by language
24 Libraries with public proof-of-concept exploits
26 OWASP and exploitability

Chapter 3 27 **Implications of Library Flaws on Applications**
29 Applications with flaws in open source libraries
30 Do more libraries inevitability mean more problems?
32 The most concerning flaws are a rare breed
34 Relative prevalence of flaws by OWASP category

Chapter 4 35 **Options for Managing Library Security Flaws**
37 Most fixes are minor
38 Fixes are present for the scariest OWASP flaws
39 A silver lining

Conclusion + Recommendations 40

Introduction + Key Findings

Application security is one of the great frontiers in information security.

> Our long-running *State of Software Security* series has mapped the uncharted areas in several directions, to the point that it's clear there is no simply saying "write better code, developers!"

Apart from the code that is authored by developers, virtually no modern application can avoid including open source libraries that provide functionality that would be extremely tedious to write from scratch.

Whether we're looking at a relatively common library with a rich feature set, such as OpenSSL, or a four line JavaScript library that provides backward compatibility (yes, we're looking at you, isarray), all of this imported code represents functionality that your developers did not author, but becomes code you have to manage. That free puppy[1] that you adopt still needs to be fed, walked, and taken to the vet.

WHAT FOLLOWS IS AN EXAMINATION OF THAT PROVERBIAL FREE PUPPY:

- **How are open source libraries actually getting used?**
- **What type of flaws are lurking under those appealing software licenses?**
- **Do developers pick safe libraries with few security flaws or are they looking for features?**
- **And finally, what can developers do to maximize their access to this functionality without burdening themselves — and their users — with security debt?**

1 Credit to Kymberlee Price for introducing us to this analogy, applied to risk management at SIRAcon 2018.

TO ANSWER THESE QUESTIONS:

We've turned to the Veracode scanning platform database of over 85,000 applications. Each of these applications has been reviewed for its component libraries, accounting for over 351,000 unique external libraries. We then sliced and diced this data by language, flaw type, dependency, and even whether there are known exploits for their flaws.

85,000 APPLICATIONS

351,000 UNIQUE EXTERNAL LIBRARIES

HERE ARE SOME OF THE THINGS WE FOUND:

TRANSITIVE DEPENDENCIES

Some language ecosystems tend to pull in many more transitive dependencies than others.

Languages that have the largest amount of attack surface introduced from transitively included dependencies:

- JavaScript
- Ruby
- PHP

FLAWED PHP LIBRARIES

Including any given PHP library has a greater than 50 percent chance of bringing a security flaw along with it.

COMMON LIBRARIES

There are a small number of libraries that are almost always found in applications.

The most commonly included libraries are present in over 75 percent of applications for each language!

MINOR UPDATES

Fixing most library-introduced flaws in most applications can be accomplished with only a minor version update.

Major library upgrades are not usually required!

ACCESS CONTROL

Among the OWASP Top Ten[2] flaws, weaknesses around access control are the most common.

This weakness represents over 25 percent of all flaws.

TOP FLAW CATEGORIES

These top four categories of flaws found in libraries represent 75 percent of all flaws:

- Access control
- Cross-Site Scripting
- Sensitive data exposure
- Injection

2 For more information on the extremely popular OWASP Top Ten project, see owasp.org/www-project-top-ten/.

1

Chapter 1

Open Source Library Usage

07 Open source library usage is very skewed
08 Core libraries are almost always included
10 Let’s talk about JavaScript
13 Libraries — very numerous — many versions — wow
14 Dependency types

> **Some of them want to use you Some of them want to get used by you.**
>
> Eurythmics, “Sweet Dreams (Are Made of This)”

We wouldn’t go so far as to say that libraries have their own wants and dreams, but it is true that application developers and library developers come together with their own desires. These sometimes overlap and sometimes differ. It’s where the incentives for both developer communities fail to align that users may bear the price of insecure software.

Let’s examine some questions, focusing on the specific quantity, types, and versions of libraries applications are using. This will give us a baseline to understand when certain findings stand out later. We can’t hit the vocal notes of Annie Lennox, but we do plan on giving some sweet data visualizations!

Open source library usage is very skewed

The number of external libraries found in any given application varies quite a bit depending on the language in which the application is being developed. This segregation by language is a theme we'll see throughout this report, so let's spend some time getting up to speed on this right away.

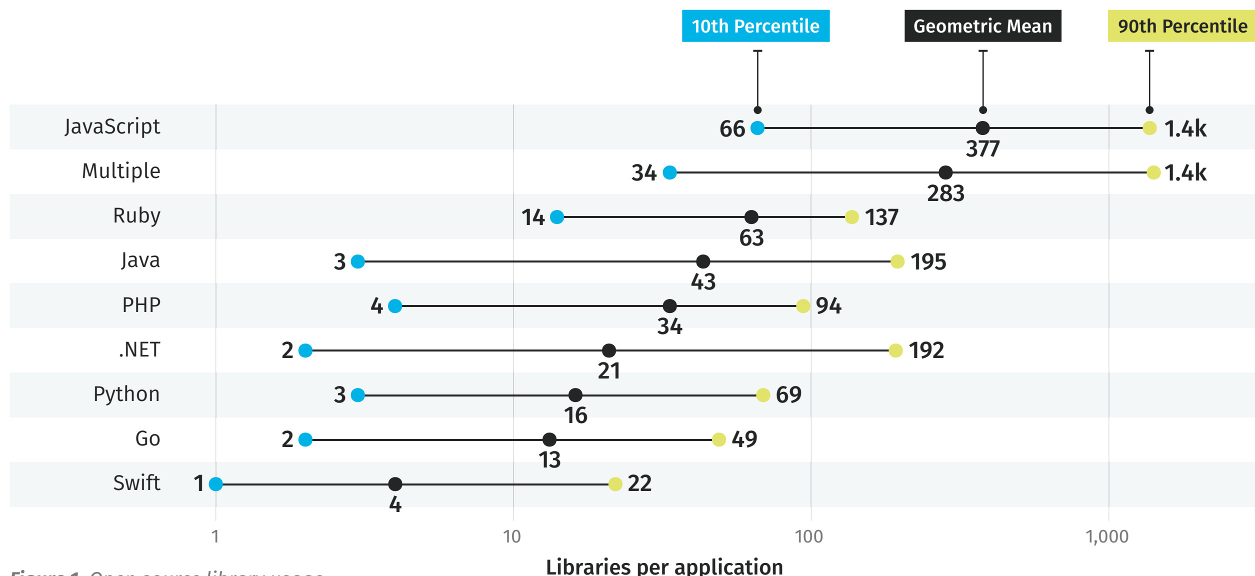


Figure 1 Open source library usage

JAVASCRIPT

Most of the JavaScript applications in our data set have hundreds of dependencies, with the dependency count reaching over 1,000 different libraries in some applications. While that number may shock you, keep in mind that the JavaScript community has a propensity to package up very small units of functionality — a point on which we'll have more to say later. This means that JavaScript has a vast number of very tiny libraries.

GO AND SWIFT

In contrast, Go and Swift tend to include only a few dozen libraries at most, which may be a function of their less developed and smaller ecosystems.

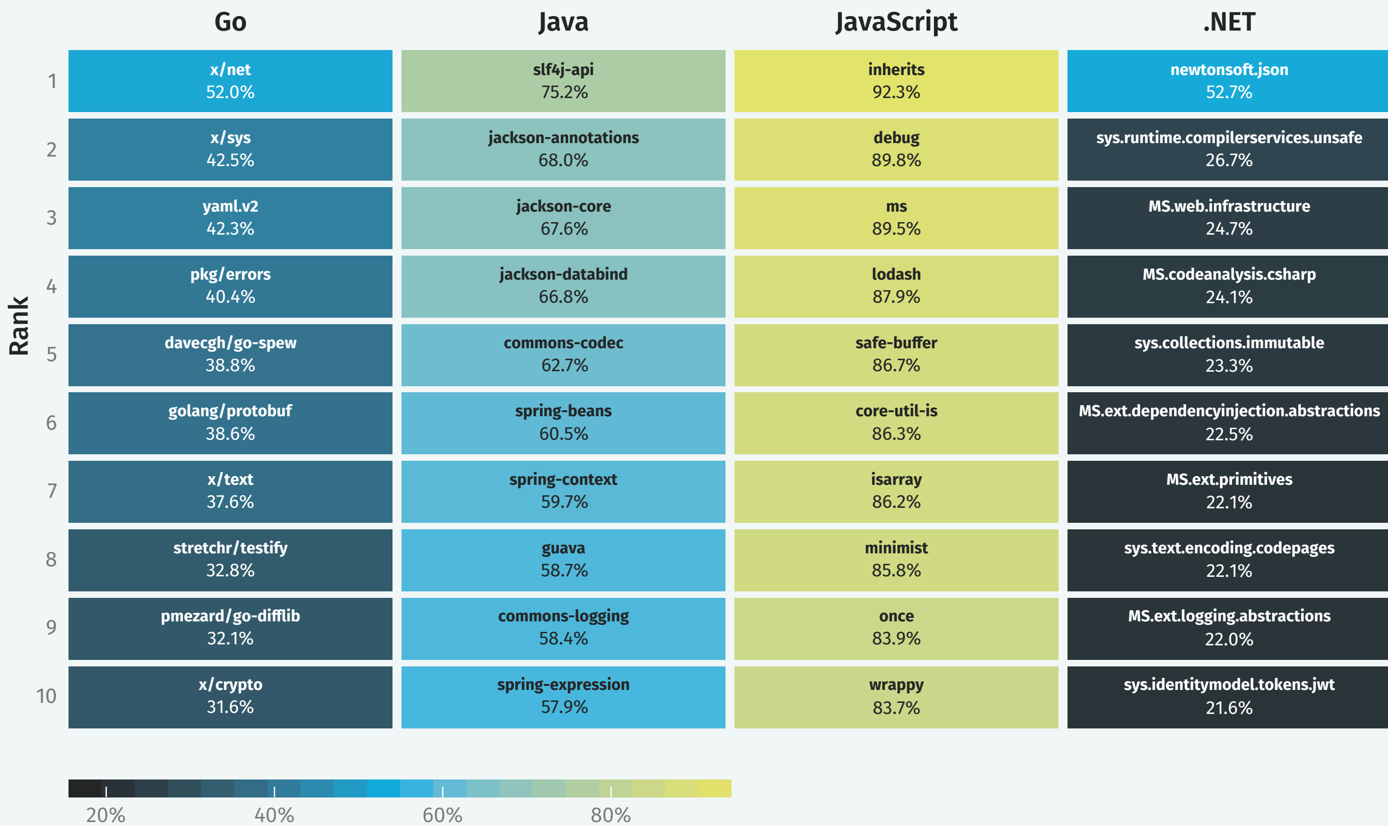
PYTHON

Python's ranking near the bottom in the number of included libraries surprised us, given our personal experience as Python developers. While the most popular Python package manager PyPi has roughly 1/5th of the number of packages found in the most popular JavaScript repository npm (221k vs 1.2M), most Python applications have only 1/100th of the number of libraries found in a typical JavaScript application.

Core libraries are almost always included

> With a rich ecosystem, it's natural that some libraries win out in the marketplace of ideas and become more popular than others.

These superstars of the open source world represent the breakthroughs that become fundamental to the way applications are typically built in their language ecosystems. Wonder who is on these all-star lists? Look no further than Figure 2, where we examine the 10 most commonly included libraries for each language along with the percentage of the applications in that language in which these whales[3] can be found.



Rank	Go	Java	JavaScript	.NET
1	x/net 52.0%	slf4j-api 75.2%	inherits 92.3%	newtonsoft.json 52.7%
2	x/sys 42.5%	jackson-annotations 68.0%	debug 89.8%	sys.runtime.compilerservices.unsafe 26.7%
3	yaml.v2 42.3%	jackson-core 67.6%	ms 89.5%	MS.web.infrastructure 24.7%
4	pkg/errors 40.4%	jackson-databind 66.8%	lodash 87.9%	MS.codeanalysis.csharp 24.1%
5	davecgh/go-spew 38.8%	commons-codec 62.7%	safe-buffer 86.7%	sys.collections.immutable 23.3%
6	golang/protobuf 38.6%	spring-beans 60.5%	core-util-is 86.3%	MS.ext.dependencyinjection.abstractions 22.5%
7	x/text 37.6%	spring-context 59.7%	isarray 86.2%	MS.ext.primitives 22.1%
8	stretchr/testify 32.8%	guava 58.7%	minimist 85.8%	sys.text.encoding.codepages 22.1%
9	pmezard/go-difflib 32.1%	commons-logging 58.4%	once 83.9%	MS.ext.logging.abstractions 22.0%
10	x/crypto 31.6%	spring-expression 57.9%	wrappy 83.7%	sys.identitymodel.tokens.jwt 21.6%

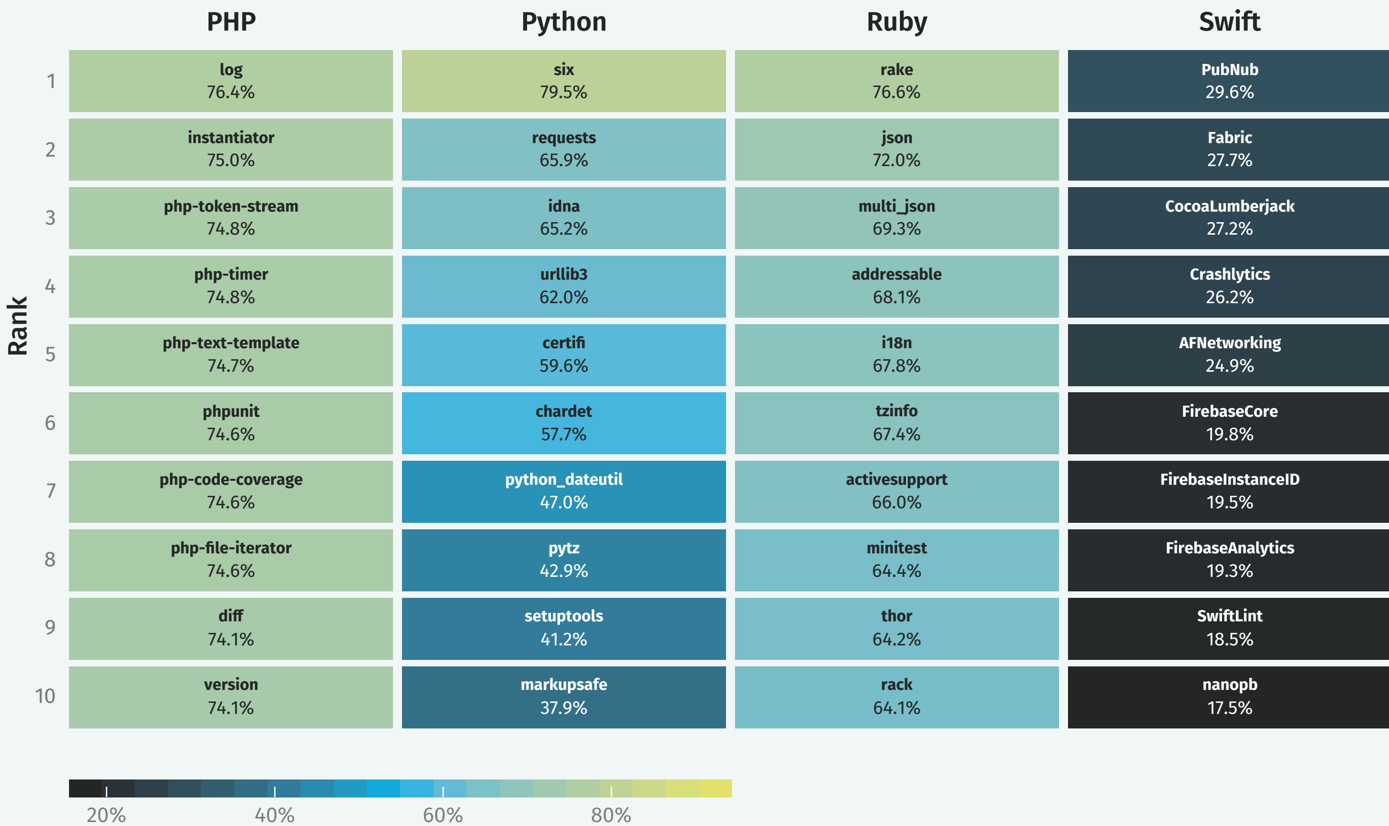
Figure 2 *Most popular libraries by language*

3 Our use of a term from gambling (en.wikipedia.org/wiki/High_roller) is deliberate here. Through use of data, you can beat the odds and have a more rational approach to information security.

Here, we see that even with the huge number of JavaScript libraries both available and in use, JavaScript's top libraries are present in more applications. In fact, all languages apart from Swift have at least one library that is included in over half of the applications scanned.

THE TAKEAWAY

Many languages have libraries that are almost a given for inclusion in an application. JavaScript and Python, in particular, have several core libraries that are likely to be in use for any given application.



Let's talk about JavaScript

> JavaScript as a language has some characteristics that set it apart several times in this study.

Rank	Go	Java	JavaScript	.NET
1	x/net 52.0%	slf4j-api 75.2%	inherits 92.3%	newtonsoft.json 52.7%
2	x/sys 42.5%	jackson-annotations 68.0%	debug 89.8%	sys.runtime.compilerservices.unsafe 26.7%
3	yaml.v2 42.3%	jackson-core 67.6%	ms 89.5%	MS.web.infrastructure 24.7%
4	pkg/errors 40.4%	jackson-databind 66.8%	lodash 87.9%	MS.codeanalysis.csharp 24.1%
5	davecgh/go-spew 38.8%	commons-codec 62.7%	safe-buffer 86.7%	sys.collections.immutable 23.3%
6	golang/protobuf 38.6%	spring-beans 60.5%	core-util-is 86.3%	MS.ext.dependencyinjection.abstractions 22.5%
7	x/text 37.6%	spring-context 59.7%	isarray 86.2%	MS.ext.primitives 22.1%
8	stretchr/testify 32.8%	guava 58.7%	minimist 85.8%	sys.text.encoding.codepages 22.1%
9	pmezard/go-difflib 32.1%	commons-logging 58.4%	once 83.9%	MS.ext.logging.abstractions 22.0%
10	x/crypto 31.6%	spring-expression 57.9%	wrappy 83.7%	sys.identitymodel.tokens.jwt 21.6%

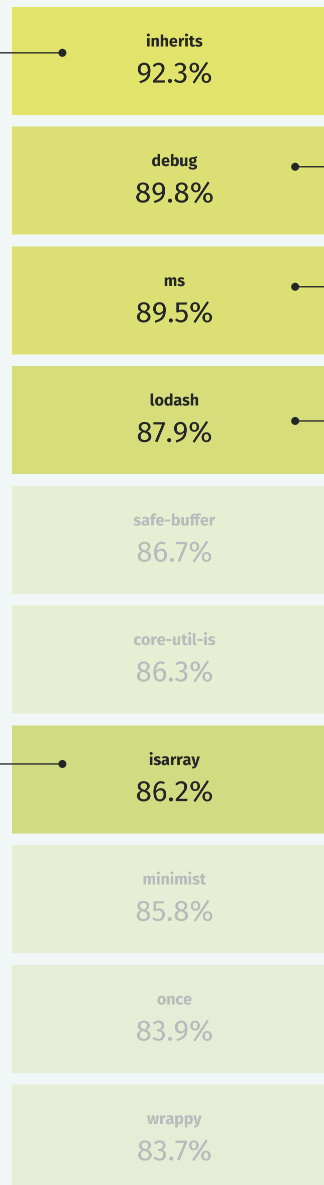
More than any of the languages we've looked at, JavaScript encourages the creation and use of very, very, small libraries that do one task. Looking at the top 10 libraries, this becomes painfully obvious.

RANK #1

inherits

36 LINES OF CODE

The most commonly used JavaScript library is inherits, located in npm. This package is 36 lines of code, which provides a thin wrapper around the inherits functionality in the utils submodule of node.js when available and provides its own version when that native module is not available. It is thankfully free from flaws based on current analysis.



RANK #2

debug

790 LINES OF CODE

RANK #3

ms

162 LINES OF CODE

The next two packages in this list — debug and ms — with 790 and 162 lines of code respectively, both have published CVEs related to denial of service flaws. So even the smallest packages, implementing trivial functionality, can have flaws, and may exist deep in a dependency tree.

RANK #7

isarray

4 LINES OF CODE

isarray is a scant four lines, doing exactly what you might think it does (it enables older versions of JavaScript to check if an object is an array), while still being the seventh most popular JavaScript package. We are forced to ask if pulling all these libraries that have to be managed by their development teams is a decision that's been consciously thought through by JavaScript developers.

RANK #4

lodash

35,000+ LINES OF CODE

Among the top 10, only one — lodash — is more than a few dozen KB in size, while the rest are less (and usually much less) than 1,000 lines of code.

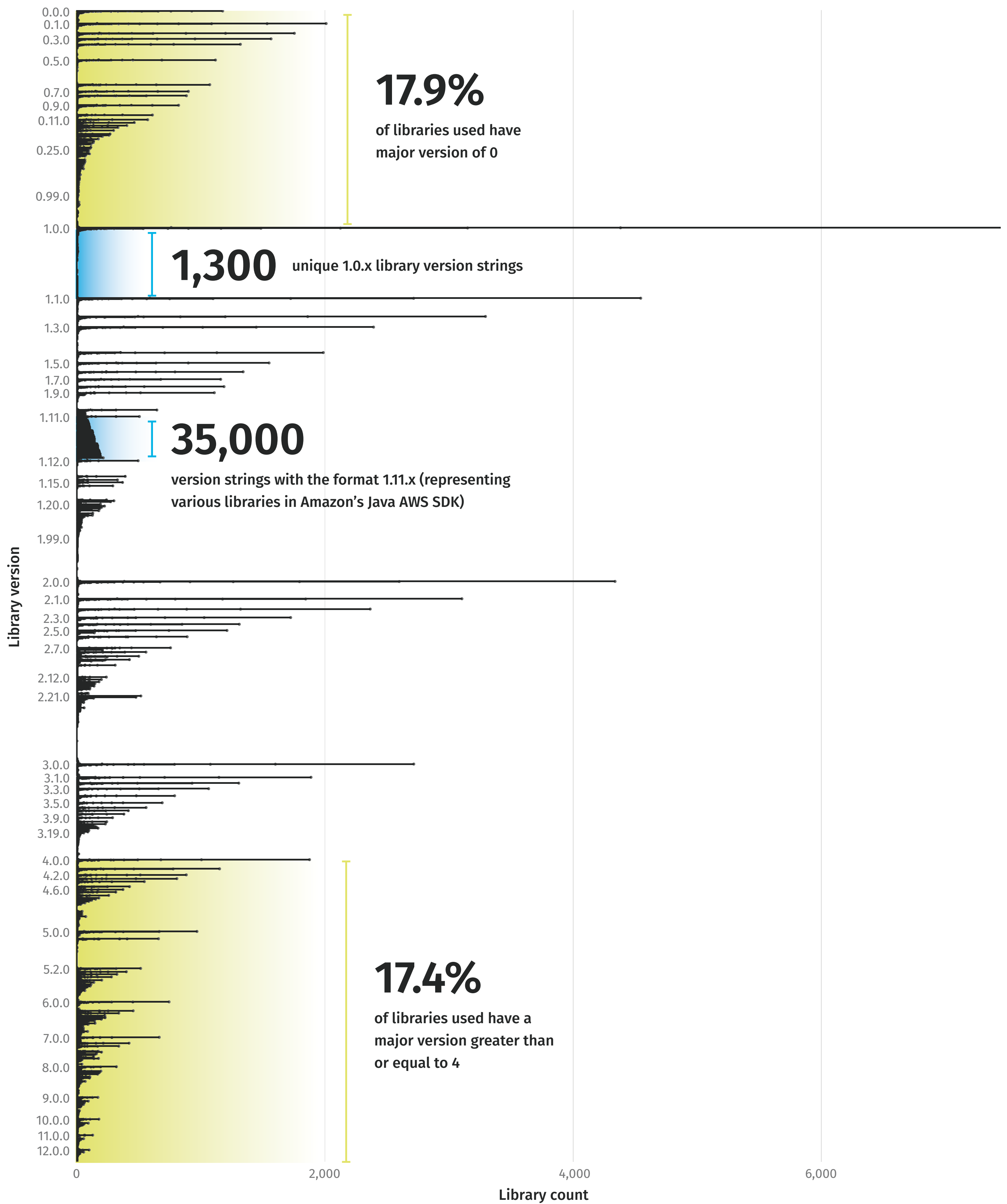


Figure 3 Library versions

Libraries — very numerous — many versions — wow

> Sure, there are those super popular libraries. However, we expect many of our readers may be asking about the versions of these and other libraries. This natural question unfortunately doesn't have a simple answer.

Most libraries use some form of semantic versioning,[4] using x.y.z formatting to denote major(x) and minor(y) versions. Things get a little strange after that, with some libraries using the third value (z) as a patch number, while others use it as a sequential build number. Some libraries eschew major and minor versions altogether, just using sequential build numbers from start to finish. And of course, in an effort to make data scientists like us insane,[5] many use some combination of custom version numbers.

We examine the landscape of these versions in Figure 3. The vertical axis includes all library versions that could be parsed as an x.y.z version string. Each spike represents the number of libraries with any particular version string.

MAJOR VERSION SPIKES

This figure presents an interesting overall pattern. Major releases (those in the x.0.0 format) represent the largest spikes, with an exponential[6] decay through successive minor versions. The major version spikes themselves also decay slowly, with only about 17 percent of libraries having a major version number greater than 4.

PRE-RELEASE LIBRARIES

Also of interest is the number of libraries that have pre-release version numbers (that is, they have a major version number of zero, i.e., 0.y.z). Nearly 18 percent of all scanned libraries are marked as pre-releases, indicating that they may have reduced testing, documentation, and overall rigor compared with major releases.

4 For more on semantic versioning (SemVer), see semver.org

5 We kept thinking of this gem from XKCD (xkcd.com/927/) when wrestling with this problem.

6 A fancy way of saying "rapid."

Dependency types

An application's attack surface is not limited to its code and the code of packages developers explicitly include. Libraries themselves have their own dependencies.[7]

These libraries that are included indirectly are transitive dependencies, and they can cascade into much more code being included in an application than a developer anticipated. Understanding how many libraries "come along for the ride" can be important in identifying where flaws might enter an application. Since they're not explicitly included by developers, a large proportion of transitive dependencies can represent attack surface that is below the surface visibility of maintainers. This hidden dependency debt represents an additional, and perhaps unexpected, workload for the ongoing vetting and maintenance of an application.

In Figure 4, we look at where applications commonly pick up their dependencies, by language. For each language, we look at each application and determine how the library came to be included.

DIRECT

If the application has most (more than 66 percent) of its dependencies from explicit calls, we count that in the direct category.

TRANSITIVE

If the application has relatively few (less than 33 percent) explicitly linked libraries and instead picks up most of its library baggage from those transitive calls, we place it in the more transitive category.

BALANCED

And what if the application is relatively split between direct and transitive sources? You guessed it — the application is counted as balanced.

7 And those second-order dependencies can have their own dependencies. It's turtles all the way down...

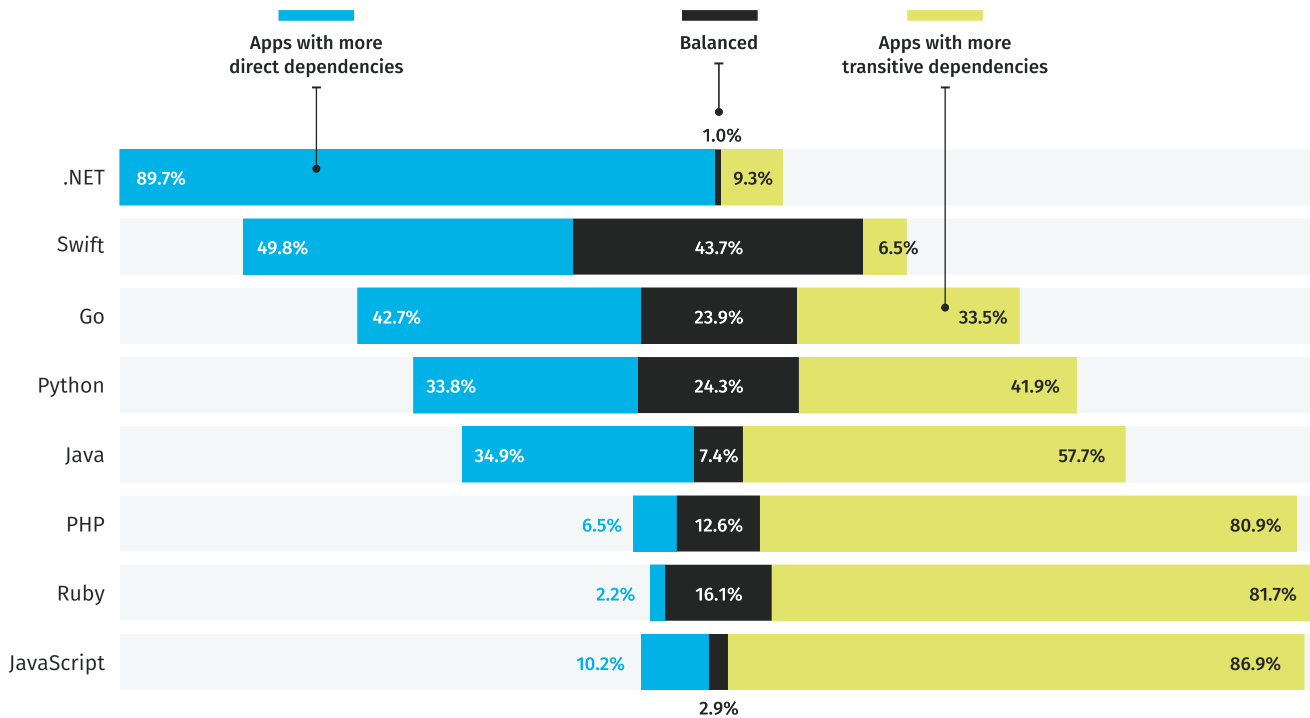


Figure 4 *Library dependency types by language*

It's important to keep in mind that being more transitive as opposed to direct isn't necessarily a bad code smell[8] by itself. Our goal is to help show which languages may have unintended consequences for maintainers. An application that picks up most of its dependencies via second, third, or even greater degrees of separation from a developer's explicit instruction increases the difficulty of managing those dependencies.

The dominance of transitive applications in the JavaScript segment is an effect of the large number of interdependencies between libraries in that ecosystem, and the preponderance of tiny, single function, libraries. While a few applications have purely direct dependencies, most have a large percentage of secondary (and tertiary and more!) dependencies. Languages like Go, Java, and Python have more even distributions among applications, while .NET is a standout with most applications having few transitive dependencies.

THE TAKEAWAY

JavaScript, Ruby, PHP, and Java have most of their attack surface from transitive inclusions that developers need to ensure they are managing.

8 For more on code smells, see Martin Fowler's Refactoring: Improving the Design of Existing Code

Flaws in Open Source Libraries

18 Flaw prevalence in libraries by language
20 Are certain types of flaws more important than others?
22 Prevalence of the OWASP top flaws by language
24 Libraries with public proof-of-concept exploits
26 OWASP and exploitability

“

Just like every cowboy sings a sad, sad song Every rose has its thorn.

Poison, “Every Rose Has Its Thorn”

Given the open source library usage we’ve seen previously, what effect does this have on the security posture of our applications? What are the nature and variety of flaws (thorns) in the garden of our libraries (roses)? To answer this question, we first need to understand how likely any particular library is to have a flaw.

0101010101

Flaw prevalence in libraries by language

In Figure 5, we provide a breakdown — by language – of the average number of flaws found in flawed libraries, compared with the percentage of libraries in that language that contain a flaw.

Keep in mind that if a library has multiple versions that are in common use, that library can be counted multiple times. Given that most libraries are rarely at their best at all times and many applications don't use bleeding-edge libraries, we feel this is an accurate representation of the world as most developers experience it.

As the percentage of libraries containing a flaw increases, it becomes more important to be aware of — and to be able to manage — flawed libraries. For instance, if you pick any random PHP library, it more than likely has a flaw. That's especially important with PHP as it's such a common application for server-side web applications and, therefore, frequently exposed to a large threat community.

We've highlighted four languages as having particularly illustrative values.

#1 — **Swift**

#2 — **PHP**

#3 — **.NET**

#4 — **Go**

But not all flaws are equal. Some security issues are relatively exotic or difficult to exploit while others may be much more significant to their application. It's this sorting of the zebras from the horses to which we now turn.

LANGUAGE #1

Swift

Swift, with its specialized use in the Apple ecosystem, has the highest density of flaws, but it has an overall low percentage of flawed libraries.

LANGUAGE #2

PHP

Compared with Go, PHP has an even higher rate of flawed libraries and over double the density of flaws in a given library.

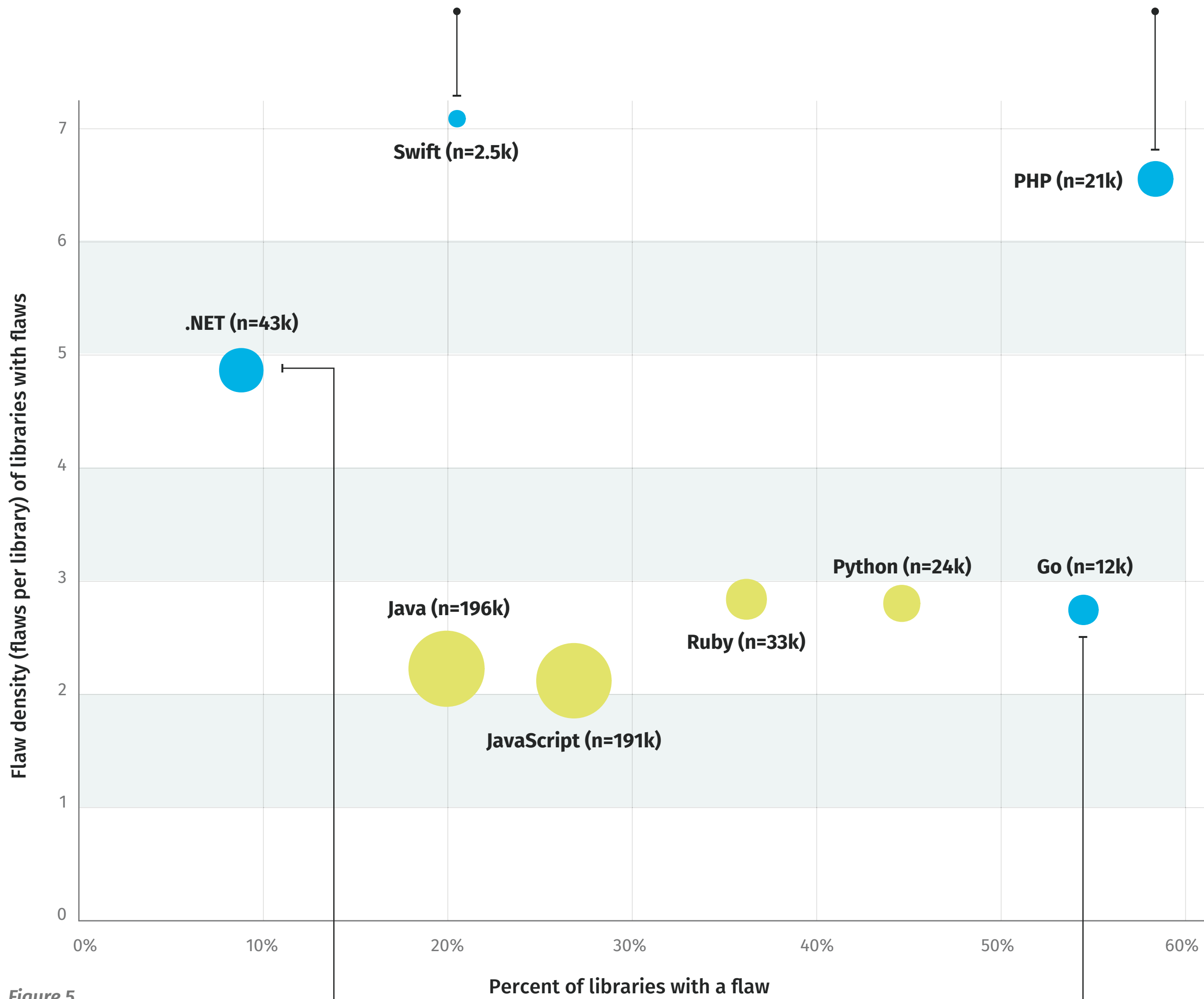


Figure 5
Flaw prevalence in libraries by language

LANGUAGE #3

.NET

Contrast Swift with .NET, which manages an impressively low percentage of flawed libraries on a population that is over 17 times larger than Swift.

LANGUAGE #4

Go

Go has a high percentage of libraries with flaws, but an overall low number of flaws per individual library.

Are certain types of flaws more prevalent than others?

> Software weaknesses may be categorized according to their Mitre Common Weakness Enumeration[9] (CWE). CWEs provide a comprehensive hierarchy of every way software can be wrong and, as a whole, is daunting. Thankfully, a more focused categorization exists — the Open Web Application Security Project (OWASP) Top 10 flaws.[10]

We'll use the OWASP Top 10 as a common lens into the nature of vulnerabilities we've detected, with two caveats:

1. The logging category is one we see very little of at the library level.

2. The known vulnerability category is self-referential.

Figure 6 examines the categories of all discovered flaws across all libraries. We see that Cross-Site Scripting leads the pack, with the insecure deserialization and broken access control categories also making up a substantial portion of all flaws. Security misconfiguration represents a tiny fraction of flaws, which is unsurprising as most libraries don't expose direct configuration — but instead rely upon the calling application code to handle configuration and deployment.

While Cross-Site Scripting and access-control issues are ones that many developers are familiar with treating, the insecure deserialization category — coming in at number two — is worth talking about in more depth. This category is an interesting member of our vulnerability fashion show.

9 For more information on CWE, see cwe.mitre.org

10 While originally designed for web applications, we can map between CWEs and these categories for a more condensed view of flaw types, see cwe.mitre.org/data/definitions/1026.html.

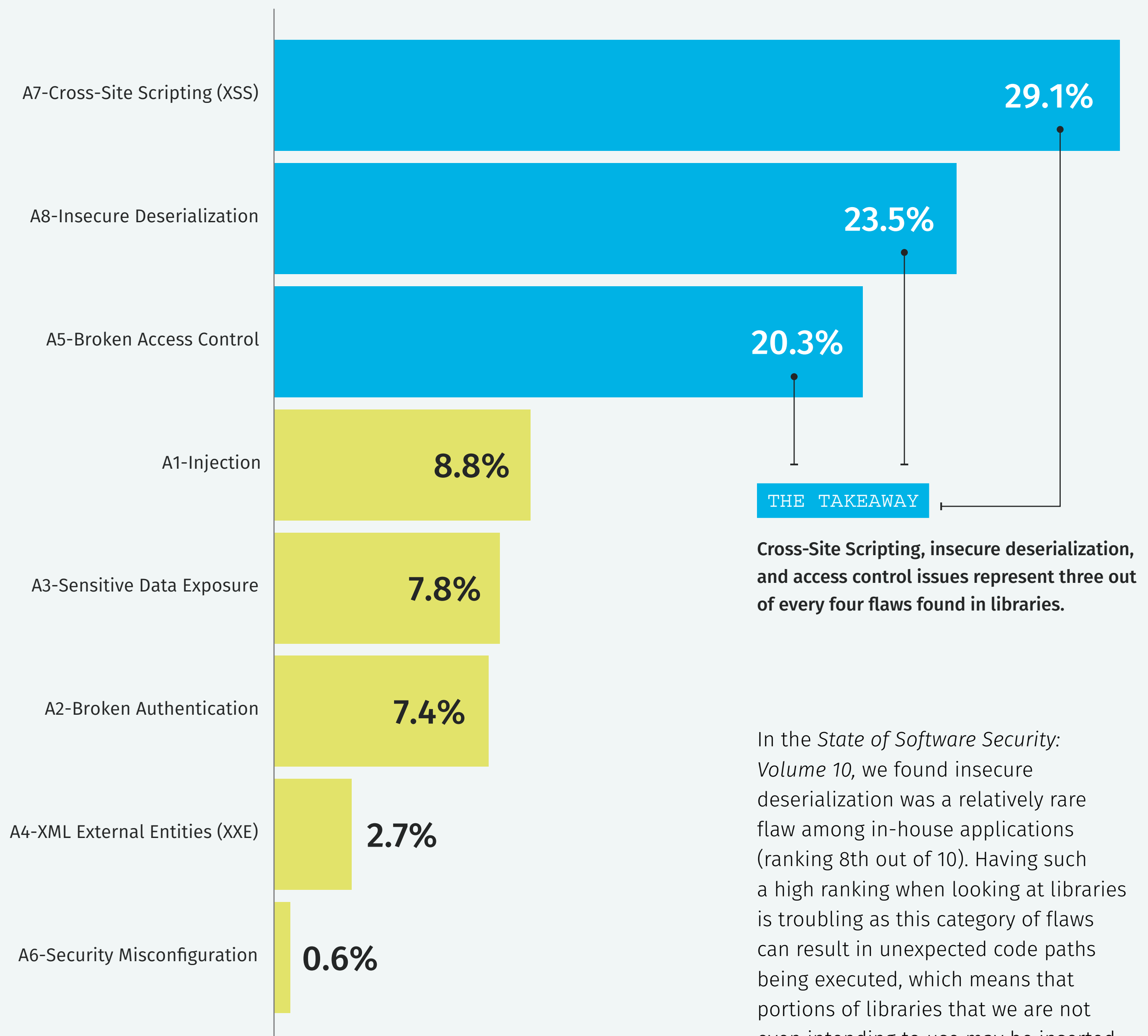


Figure 6 *Categories of discovered flaws across libraries*

THE TAKEAWAY

Cross-Site Scripting, insecure deserialization, and access control issues represent three out of every four flaws found in libraries.

In the *State of Software Security: Volume 10,* we found insecure deserialization was a relatively rare flaw among in-house applications (ranking 8th out of 10). Having such a high ranking when looking at libraries is troubling as this category of flaws can result in unexpected code paths being executed, which means that portions of libraries that we are not even intending to use may be inserted into the execution path of their hosting applications through use of this flaw.

Our next step is a more fine-grained view of these categories across different languages.

Prevalence of the OWASP top flaws by language

We know that there's quite a bit of variation among languages in the numbers and locations of the flaws in their libraries, so perhaps you can anticipate what we see below with the differences in the prevalence of the OWASP top flaw categories.

In Figure 7, we're now looking at how often these OWASP top flaws categories appear across various languages. The percentages in each tile reflect how many of the libraries in each language's ecosystem have a flaw from a given category. We suggest looking at bands of relatively heavy shading both horizontally — showing categories that are common across languages — and vertically — showing languages that have relatively high levels of OWASP top flaws overall.

SCANNING ACROSS LANGUAGES

Scanning across languages, PHP unfortunately stands out starkly, with over 40 percent of libraries in this popular language having Cross-Site Scripting issues. Broken access control and authentication — the number two and three categories for PHP — are also more prevalent here than in any other language.

SCANNING ACROSS FLAW TYPES

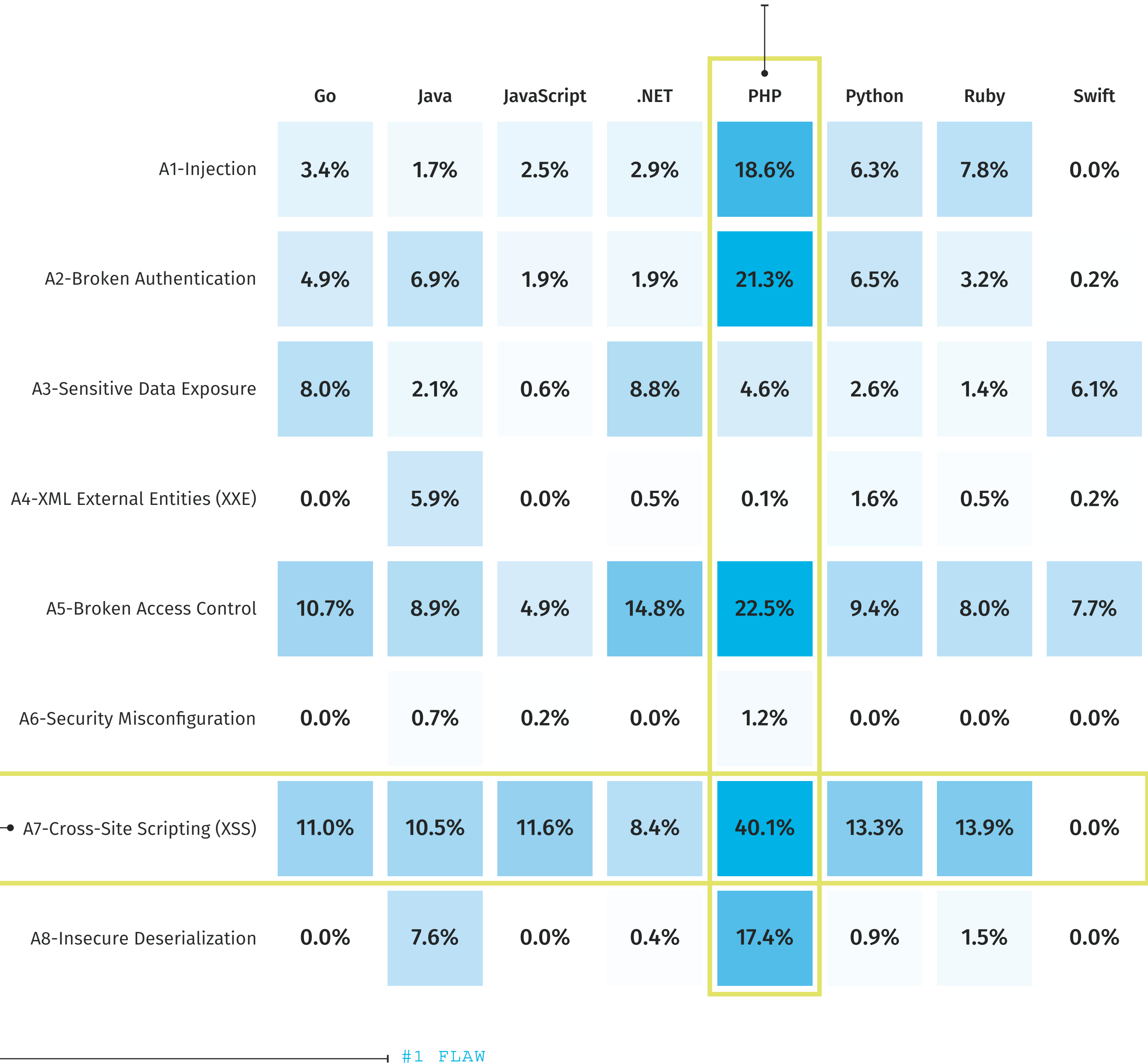
Switching to a horizontal look for common flaw types, Cross-Site Scripting is common across all languages, while our number two category from Figure 6, insecure deserialization, is found commonly only in PHP and Java.[11] Interestingly, broken access control nudges out Cross-Site Scripting for the top area of concern for users of .NET and Go's libraries.

11 The large number of libraries in these two languages is what brings this category so high in our overall rankings.

#1 LANGUAGE

PHP

With its high overall rate of flaws, PHP shows up with Cross-Site Scripting, access control, and authentication flaws.



	Go	Java	JavaScript	.NET	PHP	Python	Ruby	Swift
A1-Injection	3.4%	1.7%	2.5%	2.9%	18.6%	6.3%	7.8%	0.0%
A2-Broken Authentication	4.9%	6.9%	1.9%	1.9%	21.3%	6.5%	3.2%	0.2%
A3-Sensitive Data Exposure	8.0%	2.1%	0.6%	8.8%	4.6%	2.6%	1.4%	6.1%
A4-XML External Entities (XXE)	0.0%	5.9%	0.0%	0.5%	0.1%	1.6%	0.5%	0.2%
A5-Broken Access Control	10.7%	8.9%	4.9%	14.8%	22.5%	9.4%	8.0%	7.7%
A6-Security Misconfiguration	0.0%	0.7%	0.2%	0.0%	1.2%	0.0%	0.0%	0.0%
A7-Cross-Site Scripting (XSS)	11.0%	10.5%	11.6%	8.4%	40.1%	13.3%	13.9%	0.0%
A8-Insecure Deserialization	0.0%	7.6%	0.0%	0.4%	17.4%	0.9%	1.5%	0.0%

#1 FLAW

Cross-Site Scripting

Cross-Site Scripting is the most common type of flaw across almost every language.

Figure 7
Prevalence of OWASP top flaws in libraries by language

Libraries with public proof-of-concept exploits

Many organizations prioritize treating flaws based upon the availability of public proof-of-concept (PoC) exploits.

As code becomes available to demonstrate in practical terms that a flaw can be leveraged to exploit a codebase, the probability of that flaw being used for harm is generally agreed to be higher.[12] With our understanding of how prevalent flaws are across languages and in what quantity, we now look at how frequently those flaws have PoC exploits. For this data, we partnered with Kenna Security to obtain numbers on both the availability of PoC and, as we'll see later, the detection of those PoCs being used in the wild.

In Figure 8 we see that just over one-fifth of all libraries have a publicly published PoC exploit. There's quite a bit of variability by language. The figure demonstrates the percent of libraries with a flaw that also have a PoC exploit published. Here we see some different messages emerging.

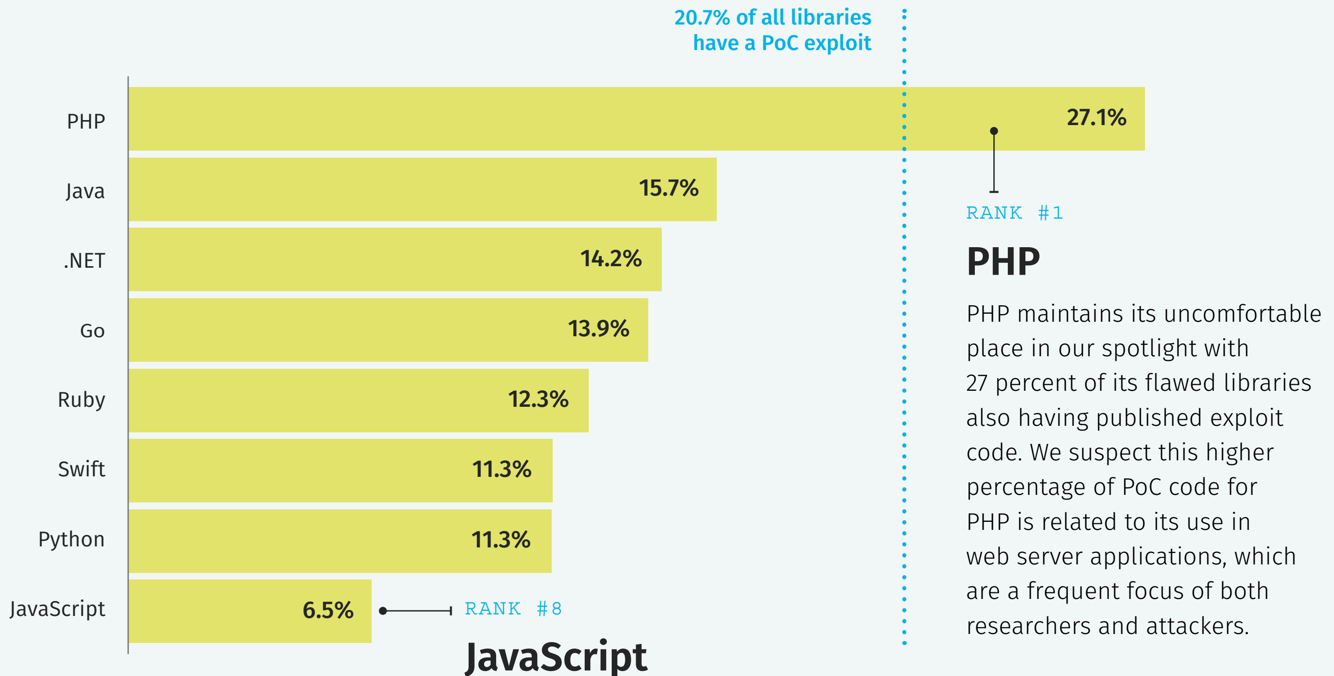


Figure 8
Percent of flawed libraries with a PoC exploit

12 The availability of PoC code is a component in both the CVSS (nvd.nist.gov/vuln-metrics/cvss/v2-calculator) and EPSS (www.kennaresearch.com/tools/epss-calculator/) vulnerability scoring systems. Kenna also cultivates proof of concept information from a number of public data sources.

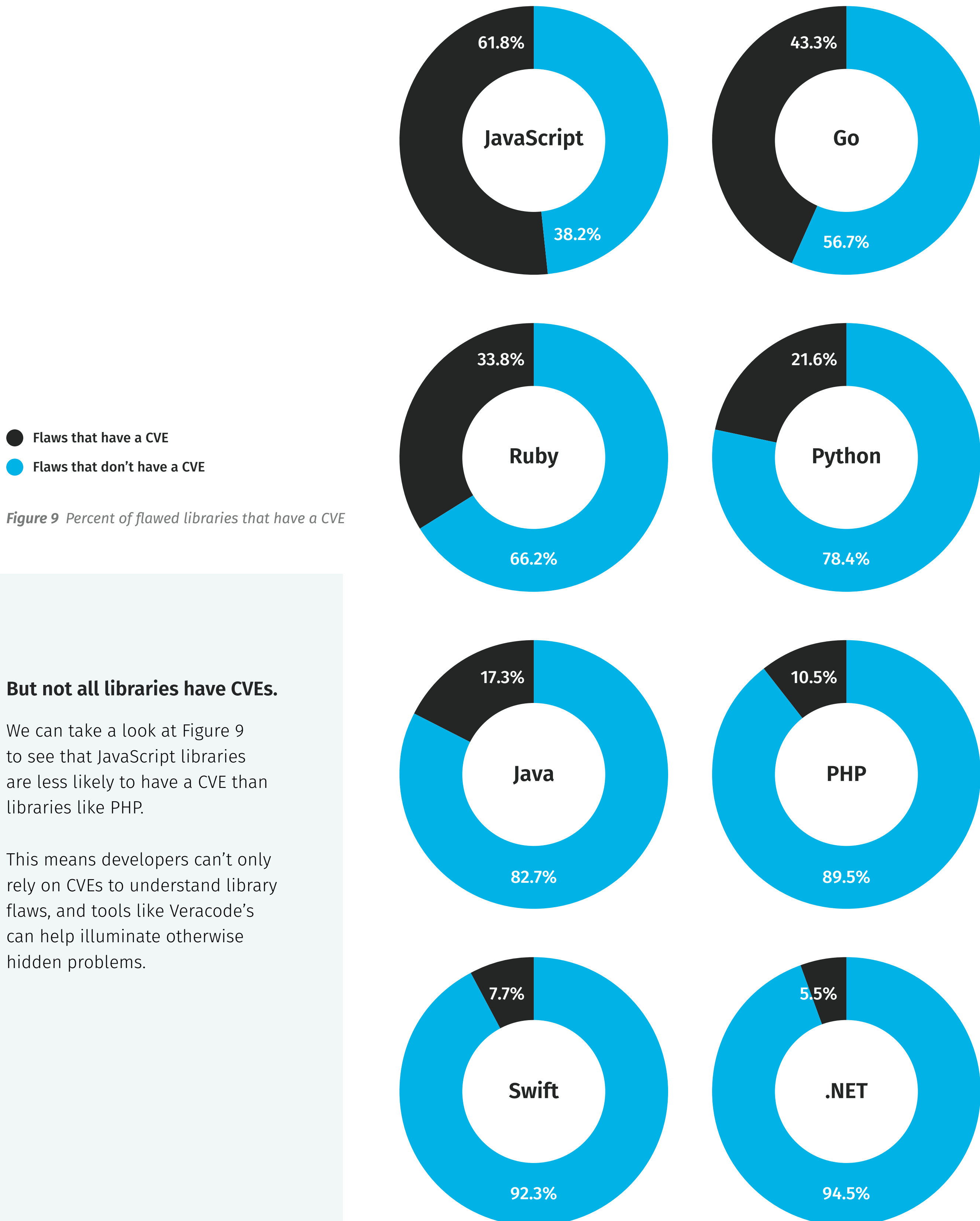


Figure 9 Percent of flawed libraries that have a CVE

But not all libraries have CVEs.

We can take a look at Figure 9 to see that JavaScript libraries are less likely to have a CVE than libraries like PHP.

This means developers can't only rely on CVEs to understand library flaws, and tools like Veracode's can help illuminate otherwise hidden problems.

OWASP and exploitability

> We can further refine our focus by looking at those flaws from the OWASP Top Flaws categories that also have public proof-of-concept code.

To set the story for this part of our discussion, we present Figure 10, which plots the percentage of all libraries with an OWASP Top Flaw against the percentage of libraries with a PoC for those flaws.

Here, we've highlighted three categories of the OWASP Top Flaws for special attention. While most of the top flaw categories are clustered in the single digits for both prevalence in all libraries and availability of proof-of-concept code, these three pull away from their peers quite starkly.

RANK #1 + #2

Insecure Deserialization + Broken Access Control

These two categories have almost three times the rate of public PoC. Remember that insecure deserialization is almost entirely dominated by PHP and Java, and you can see how this information can be used to focus remediation and management efforts.

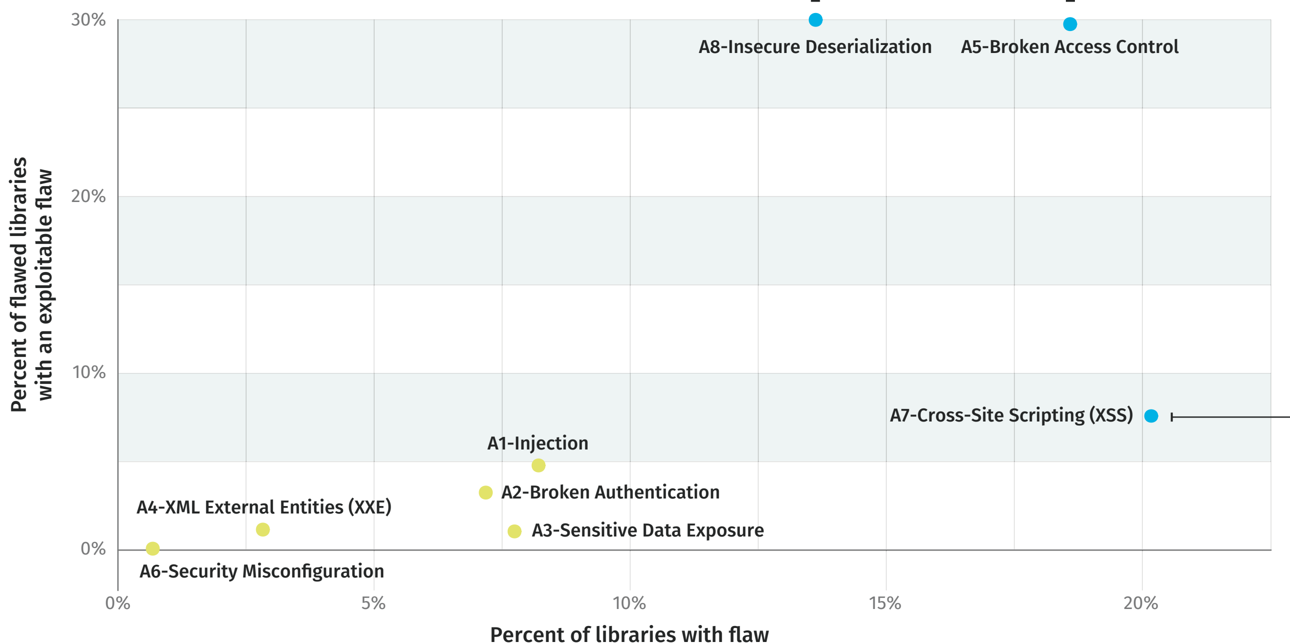


Figure 10
Percent of libraries with an exploitable flaw

RANK #3

Cross-Site Scripting

Our top category for absolute rate of occurrence has a slightly higher percentage of PoC availability, but not dramatically.

3

Chapter 3

Implications of Library Flaws on Applications

29 Applications with flaws in open source libraries
30 Do more libraries inevitability mean more problems?
32 The most concerning flaws are a rare breed
34 Relative prevalence of flaws by OWASP category

“

There is a crack a crack in everything That’s how the light gets in.

Leonard Cohen, “Anthem”

So far we’ve looked at the flaws present in libraries as a whole, but now we can go up the stack even further to look at the applications themselves. After all, it is our applications that we are trying to protect.

0101010101

Applications with flaws in open source libraries

> Examining Figure 11, we are presented with a startling fact — most (71 percent) applications have a flaw in an open source library when they are first scanned.

Those readers who recall the hundreds of libraries included with many applications may find this less startling. While this high rate of flaws upon initial scanning is a little concerning, we'll try to moderate that initial fear with some specific guidance on treatment a little further on.

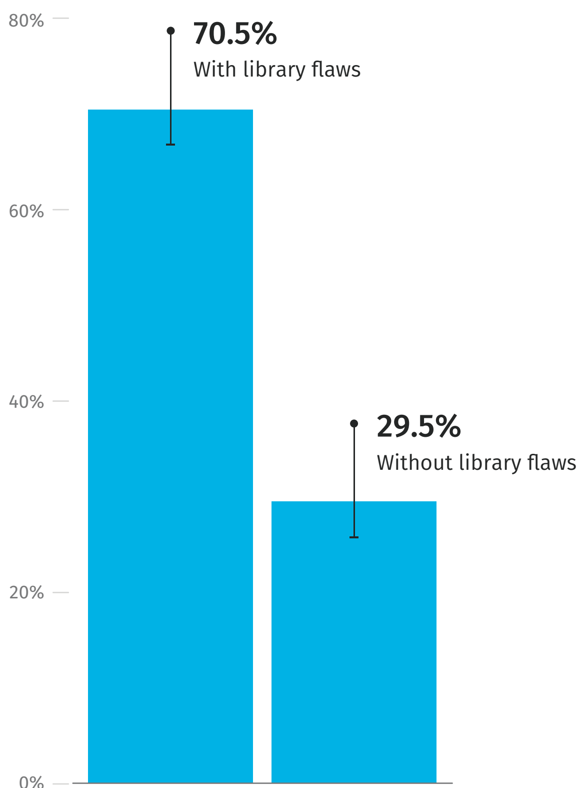


Figure 11
Applications with flaws in an open source library on first scan

> One important consideration is exactly *how* a flaw is being included in applications. As we saw in the first section, this can vary widely among applications. Figure 12 has the answer — flaws are mostly induced from transitive library dependencies.

But surely developers are security conscious and are selecting libraries that have fewer flaws? As data scientists, we don't stand on wishful thinking — no matter how tempting. We performed an extensive review of the frequency that libraries — both with and without security flaws — are found in applications across all of our languages.

We found that flawed libraries don't get used less; in fact, they frequently get used more often. ***While there may be distinct characteristics by which developers choose the libraries they include, it doesn't appear that security vulnerabilities is one of them.*** Will that behavior persist after this report is in the hand of developers? Time will tell!

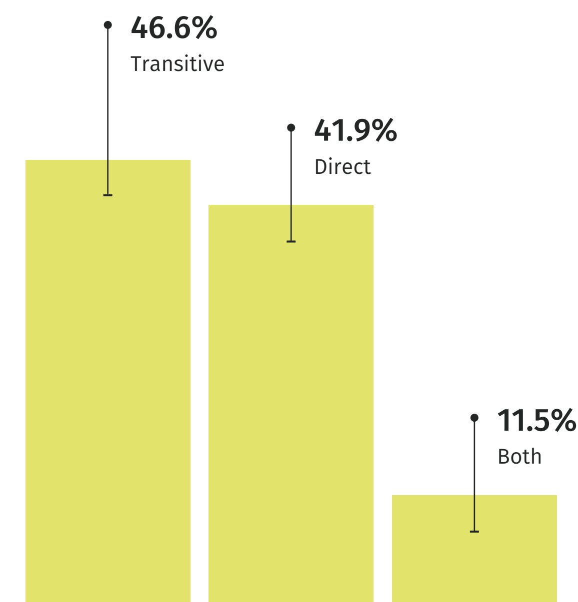


Figure 12
How flaws in open source libraries are included in applications

Do more libraries inevitably mean more problems?

> It's true that the more libraries a given application includes, the more flaws a developer is going to introduce on average.

In Figure 13, we look specifically at this relationship across languages. The relationship between the expanding scope of an application and the number of flawed libraries used is unique for each language. The color indicates the number of applications at a particular point on the chart.

JAVA, JAVASCRIPT, AND PYTHON

In particular, these three languages seem to have a basic relationship that the more libraries an application includes, the more likely flawed libraries will be included. This seems to get exponentially worse as applications use hundreds or thousands of libraries, with a particularly high density of JavaScript applications including many dozens of flawed libraries. Python bears a more direct scaling relationship, with a flawed library being included with roughly every 10 libraries used.

.NET

.NET is unique in the areas of highest density appearing as a line of zero flaws along the bottom of the chart.

RUBY, PHP, GO, AND SWIFT

These four languages have sparser unique combinations, but still show weak correlation among libraries and flawed libraries.

On a positive note, we observe that correlation is not fate.

Across all languages, we see applications (in the lower right of any language in Figure 13) that use hundreds (or, in the case of JavaScript, thousands of libraries), with minimal or no flawed libraries included. Complexity doesn't have to mean sacrificing security.

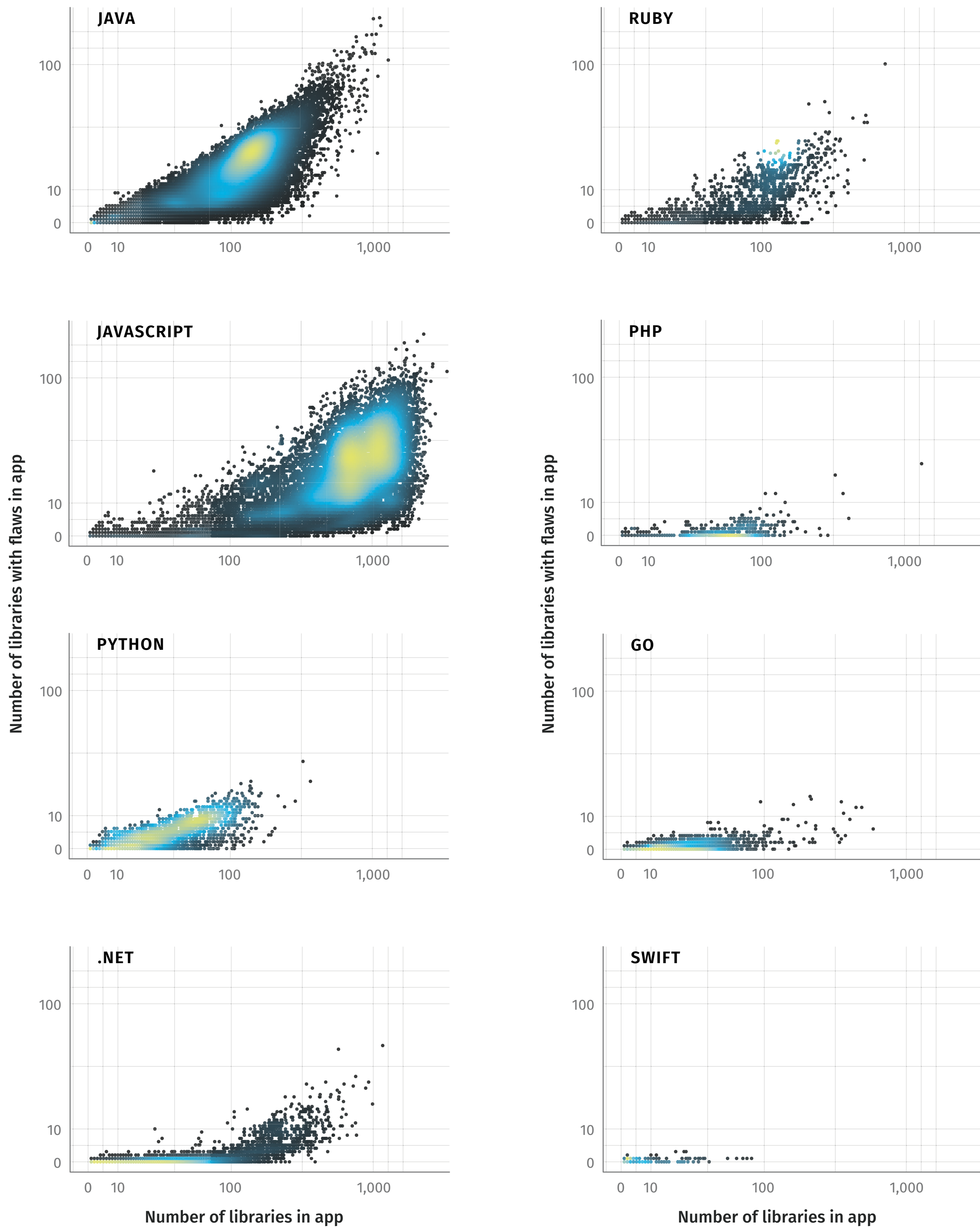


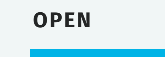
Figure 13 Number of flawed libraries unique to each language

The most concerning flaws are a rare breed

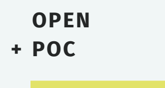
We've talked a lot about various types of flaws, when they occur in libraries, and when those libraries are subsequently used in applications. It's clear there is a lot to worry about out there. But in what order should we arrange our worry?

Developers are constantly working to close flaws.

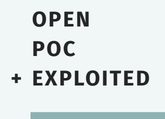
The flaws can be fixed by hand or mitigated another way. The presence of a flaw in a library doesn't mean that the flawed bits will be on the executable path of the application. Moreover, just because a flaw exists, it doesn't mean there is an attacker raring and ready to exploit it. So let's examine these things in order in Figure 14.



Almost all scanned applications have an unfixed flaw in an external library (97.4 percent of applications).



If we prioritize our focus on those flaws that have public proof-of-concept code, the number of libraries we have to contend with drops to just over half. But, wait! We can do even better!



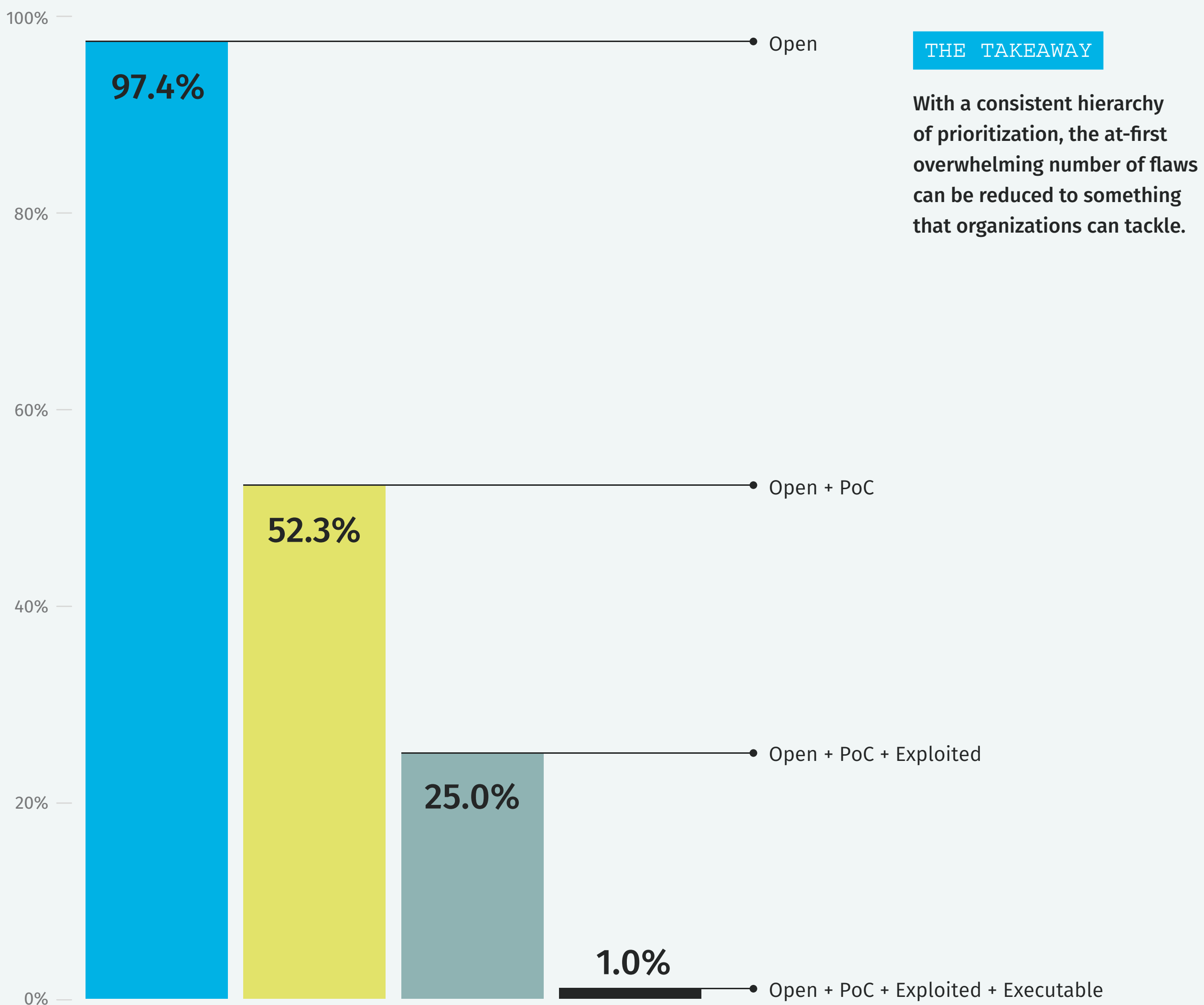
Just because a flaw has a public PoC doesn't mean attackers are using it. Our data on public proof of concepts also notes whether attackers have attempted to exercise that flaw in the wild.[13] Filtering to those vulnerabilities where an attack has been seen in-the-wild brings another 50 percent reduction, with just 25 percent of flaws now in scope for our most critical attention. For most readers, we would suggest this is the minimum viable population of vulnerabilities to address first. But there is one more layer we can explore, however.

Veracode Software Composition Analysis can check whether the flawed parts of the library make it into the application's executable path. With this ultimate check on a flaw with PoC, exploited in the wild, and present in a given application's execution chain, we're now looking at just a small 1 percent of flaws of highest priority. We caution that being on an application's execution chain is a conservative one, only flagging a vulnerability as being on the execution chain if there is a high degree of confidence that this claim is accurate. This reasonable approach to avoid false-positive alerts to developers means that this estimate is likely on the low side.

13 Obviously, we can't see every attack that is attempted. But the data we borrowed from Kenna gathers information from a variety of sources to estimate what vulnerabilities are being exploited in the wild. See the previous references for more information.

Does this mean developers should ignore all those other third-party flaws?

Of course not! They could cause problems down the road, and attackers are always expanding their book of tricks. Organizations and developers need to choose how to focus their resources. We suggest looking at these characteristics and deciding which heuristics work best for your own risk tolerances.



THE TAKEAWAY

With a consistent hierarchy of prioritization, the at-first overwhelming number of flaws can be reduced to something that organizations can tackle.

Figure 14 Qualifying the flaws in open source libraries

Relative prevalence of flaws by OWASP category

> Before we think about the best way to manage flaws in open source libraries, it's instructive to understand that the flaws in libraries are not the same as the flaws that end up in applications and that attacker focus can be altogether different.

Figure 15 makes a comparison across these three categories. We start on the left with a ranking of the most common OWASP categories as they appear in all libraries that have flaws. As we've seen in our previous sections, Cross-Site Scripting is at the top of the charts.

As we narrow our scope to just those applications with flaws that are sourced from a dependency, the relative rankings shift — with insecure deserialization and XML external entities rising markedly.

Becoming even more restrictive in our scope to looking just at applications with flaws from libraries that also have a public proof of concept exploit (those most urgent of flaws), we see a ranking that mirrors our first-hand experience with application security teams, with Cross-Site Scripting falling down the rankings in favor of issues resulting from access-control issues.

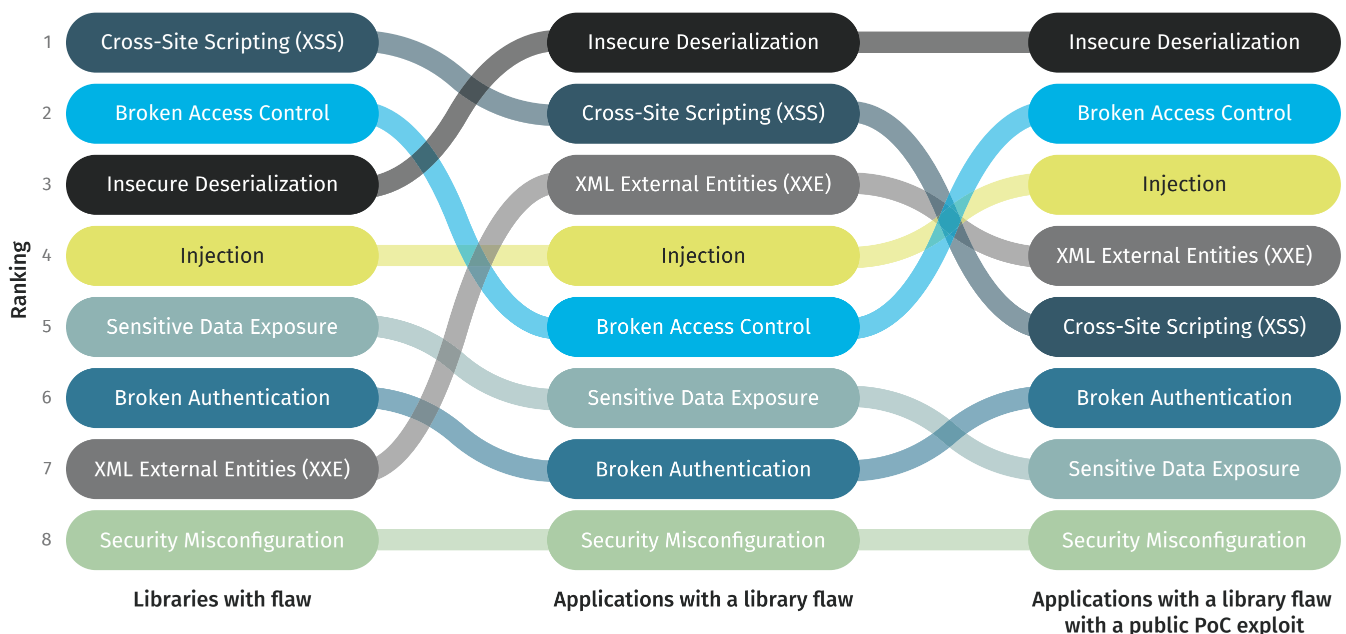


Figure 15
Open source flaw types by exploitability

THE TAKEAWAY

Insecure deserialization stays near the top, while XSS steadily drops as we get closer to flaws more likely to be attacked.

4

Chapter 4

Options for Managing Library Security Flaws

37 Most fixes are minor
38 Fixes are present for the scariest OWASP flaws
39 A silver lining

“

Oh, there’s a lot of opportunities If you know when to take them.

Pet Shop Boys, “Opportunities”

Most fixes are minor

> Now that we understand how libraries are used and in what ways they may contribute to security issues, we're ready to tackle guidance on how to manage this important area.

While a simple call to "update all libraries, everywhere" may sound great (especially to some of our auditor friends), the practical limitations ignored in that simple statement are well-known. But if keeping everything up to date at all times is more of an aim than an achievable goal, how can you prioritize your efforts to still achieve good risk management?

With Figures 16 and 17, we see that most (nearly 75 percent) of the known flaws can be fixed with an update. That's great news — these are bugs with an available solution! That message gets even better when we factor in that most of the security flaw fixing updates are minor revisions or even just patch revisions. These minor and patch updates generally do not change APIs (if semantic versioning rules are being followed) and are some of the least disruptive for application developers to apply.

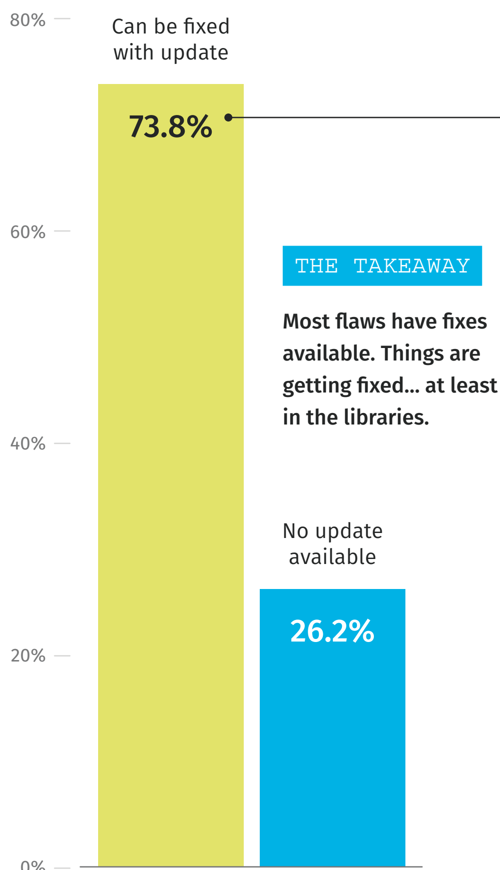


Figure 16 Percent of library flaws with an available update

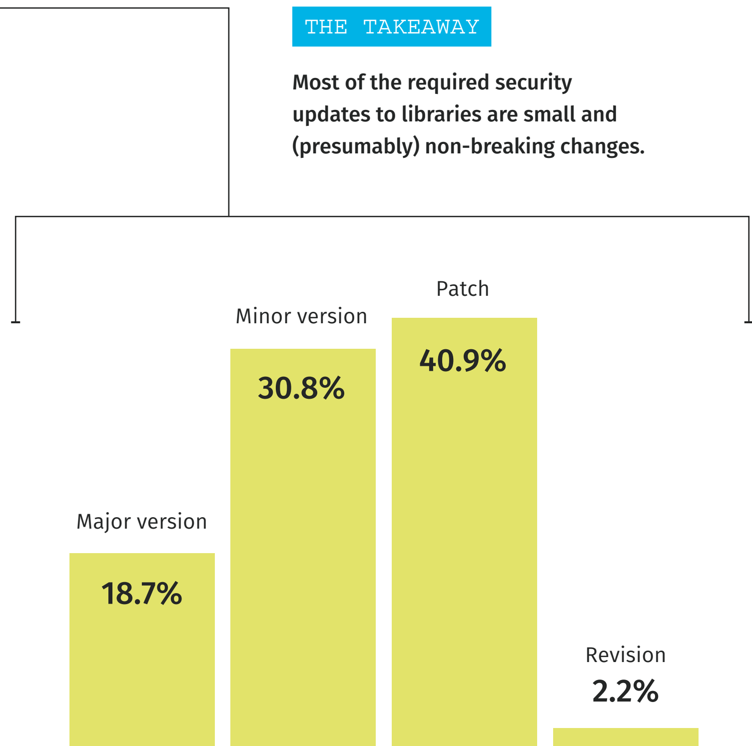


Figure 17 Types of updates available for library flaws

Fixes are present for the scariest OWASP flaws

> This good news gets even better when we look at the high fix rate of library developers for some of the 'scariest' flaws.

In Figure 18 below we show some of those top OWASP categories along with the percentage of those flaws that have a fix in a published version. This story is encouraging, with nearly 90 percent of broken access control flaws able to be corrected with a published update. This is crucial as it is the second most likely flaw to be included in an application where a PoC exploit exists.

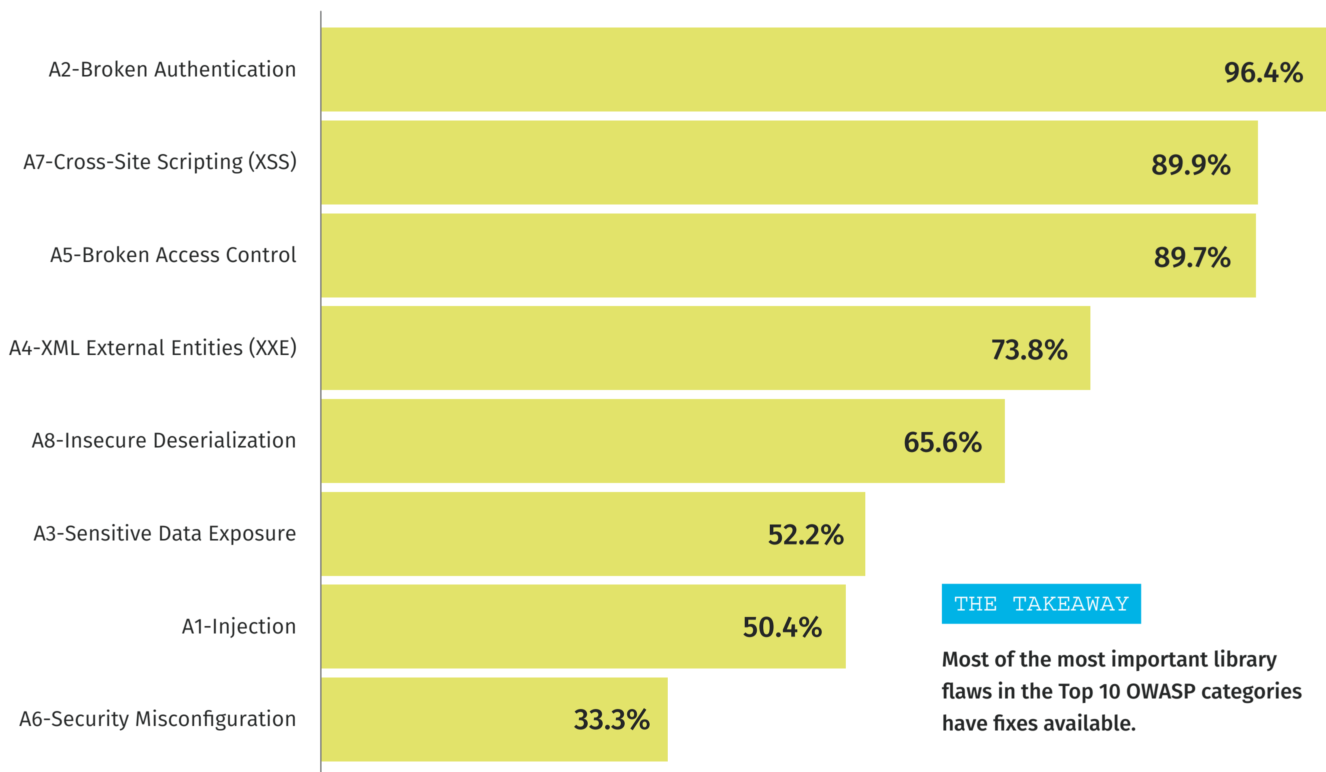


Figure 18 Percent of flaws with available fixes

THE TAKEAWAY

Most of the most important library flaws in the Top 10 OWASP categories have fixes available.

A silver lining

> Let's check back in with those most concerning flaws we talked about previously.

That is those flaws we discussed as having public PoC exploits that have evidence of being used in the wild and that are on an application's executable path.

These are certainly complicated beasts, and it's unlikely they can be fixed with a simple update… right? Figure 19 says otherwise, in fact it shows that those 90 percent of those most concerning 1 percent of flaws can be fixed with an update to the library.

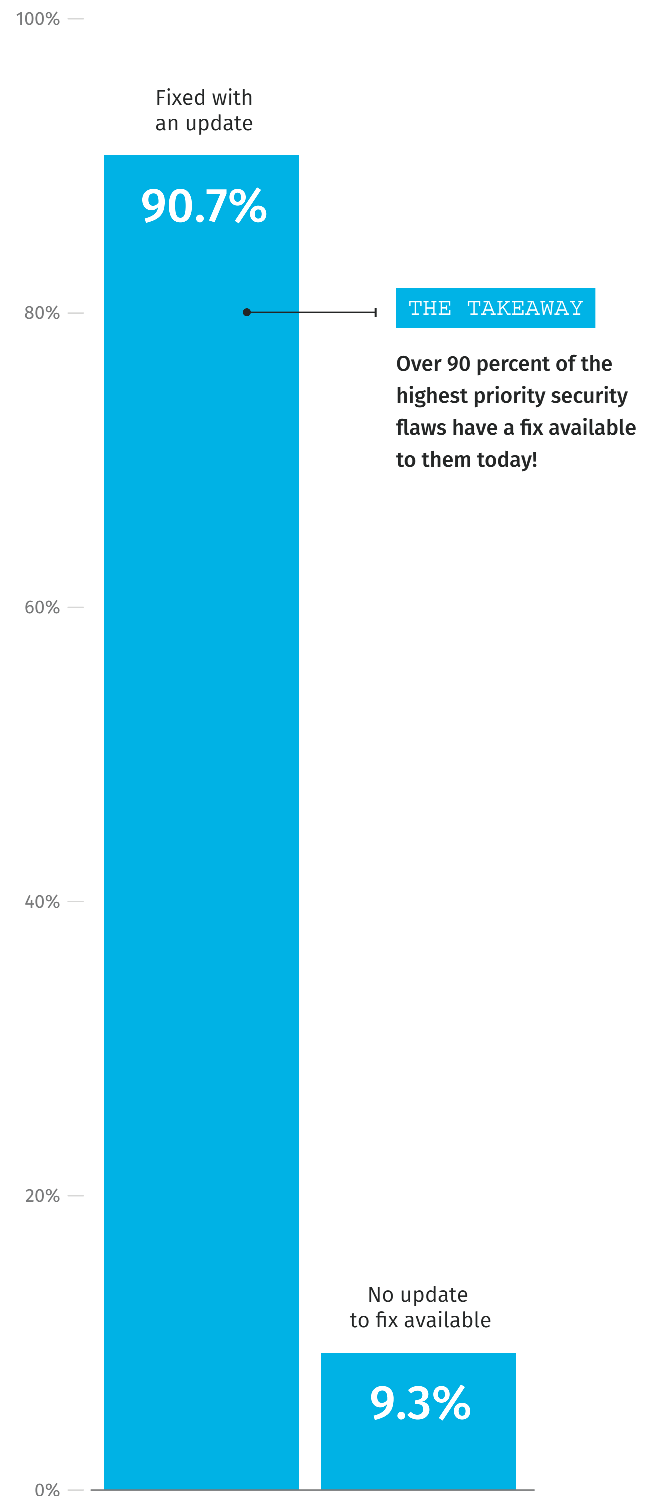


Figure 19
Percent of open flaws on executable path with exploit in the wild

Conclusion + Recommendations

“

When you reach the bottom line The only thing to do is climb.

The Bottom Line, “Big Audio Dynamite”

If software is eating the world,[14] then security flaws in software are perhaps the unpleasant indigestion. Writing software is now a team activity, with collaboration happening across the globe — whether those team members are within a single organization or encompassing the vibrant open source community.

As work patterns have had to adapt to the global economy, so too must our security management practices. Open source software gives companies tremendous advantages, but there’s no free lunch here, and all code must be managed to avoid your own contributions (whether open or closed source in nature) from exposing your users to vulnerabilities.

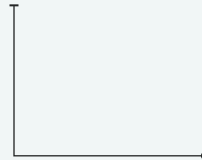
14 Andreessen, Marc. “Why software is eating the world.” *Wall Street Journal* 20.2011 (2011): C2.

So what's the bottom line?

What should our developer readers be acting upon? Our recommendations focus on awareness.

AWARENESS THAT:

Open source software has a surprising, and surprisingly variable, number and type of software flaws.



AWARENESS THAT:

The attack surface of many applications — due to the transitive dependency phenomenon — is much larger than developers may expect.

AWARENESS THAT:

There are fixes for these issues, if developers are aware of them and take the time to apply them.

AWARENESS THAT:

Language selection does make a difference — both in terms of the size of the ecosystem and in the prevalence of flaws in those ecosystems.

AWARENESS THAT:

Most of these fixes are relatively minor in nature, suggesting that this problem is one of discovery and tracking, not huge refactoring of code.

Learn more about managing your open source risk.

VERACODE

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.

LEARN MORE

www.veracode.com

Veracode Blog

Twitter

Copyright © 2020 Veracode, Inc. All rights reserved. All other brand names, product names, or trademarks belong to their respective holders.