
O'REILLY®
Compliments of
snyk
Securing Open Source Libraries
2019 Update
Managing Vulnerabilities in Open Source Code Packages
Guy Podjarny
REPORT

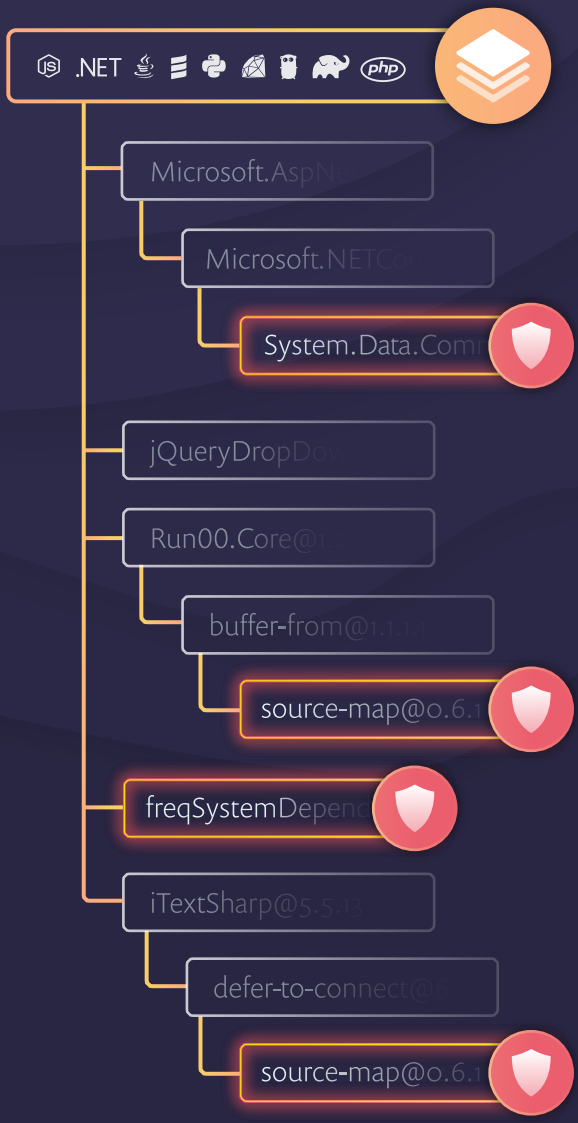
snyk

Securing Open Source Libraries **with Snyk**

Join more than 200,000 developers using Snyk to automatically find and fix vulnerabilities in in open source code packages.

Snyk is the leading developer- first security solution that continuously monitors your application's dependencies and helps you quickly respond when new vulnerabilities are disclosed.

Create a free account at Snyk.io



Customers protected by **snyk**

Microsoft · New Relic · skyscanner · Google · The Guardian · Adobe · salesforce · mongoDB

Securing Open Source Libraries

Managing Vulnerabilities in Open Source Code Packages

Guy Podjarny

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Securing Open Source Libraries
by Guy Podjarny

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Courtney Allen
Production Editor: Nicholas Adams
Copyeditor: Jasmine Kwityn
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2017: First Edition

Revision History for the First Edition

2017-11-14: First Release
2019-06-24: Second Release

This work is part of a collaboration between O'Reilly and Snyk. See our statement of editorial independence.

978-1-491-99697-3

[LSI]

Table of Contents

Introduction

Open source software—the code of which is publicly available to scrutinize and typically free to use—is awesome. As consumers, it spares us the need to reinvent the wheel, letting us focus on our core functionality and dramatically boosting our productivity. As authors, it lets us share our work, gaining community love, building up a reputation, and at times having an actual impact on the way software works.

Because it's so amazing, open source usage has skyrocketed. Practically every organization out there, from mom-and-pop shops to banks and governments, relies on open source to operate their technical stacks—and their businesses. Tools and best practices have evolved to make such consumption increasingly easier, pulling down vast amounts of functionality with a single code or terminal line.

Unfortunately, using open source also carries substantial risks. We're relying on this crowdsourced code, written by strangers, to operate mission-critical systems. More often than not, we do so with little or no scrutiny, barely aware of what we're using and completely blind to its pedigree.

Every library you use carries multiple potential pitfalls. Does the library have software vulnerabilities that an attacker can exploit? Does it use a viral license that puts our intellectual property at risk? Did a malicious contributor hide malware amidst the good code?

Unlike commercial software, Free Open Source Software (FOSS) rarely offers any guarantees or warranties. As a consumer of open source software, it is your responsibility to understand and mitigate these risks.

This risk materialized in full force with the Equifax data breach announced in September 2017. The hack, which exposed extremely personal information of *143 million* individuals, was possible due to a severe vulnerability in the open source Apache Struts library. This vulnerability was disclosed in March 2017, the Equifax system in question was not patched until late July, only *after* the breach was discovered. Equifax was fully capable of identifying and fixing this issue earlier, preventing the massive leak, and many claim not doing so is negligence on the company's part. The Equifax breach is certain to become a poster child for the importance of securing data and using open source responsibly.

Book Purpose and Target Audience

This book will help you address the risk of vulnerable open source libraries, the very thing that tripped up Equifax. As I'll discuss throughout this book, such vulnerable dependencies are the most likely to be exploited by attackers, and you'll need good practices and tools to protect your applications at scale.

Because the responsibility for securing applications and their libraries is shared between development (including DevOps) and application security, this book is aimed at architects and practitioners in both of these departments.

With that in mind, the next few sections further explain what is in and out of scope for this book. The remaining topics will hopefully be covered in a broader future book.

Tools Versus Libraries

Open source projects come in many shapes and forms. One somewhat oversimplified way to categorize them is to divide them into tools and libraries.

Tools are standalone entities, which can be used or run without writing an application of your own. Tools can be big or small, ranging from tiny Linux utilities, such as cat and cURL, to full and complex platforms such as CloudFoundry or Hadoop.

Libraries hold functionality meant to be consumed inside an application. Examples include Node.js's Express web server, Java's OkHttp HTTP client, or the native *OpenSSL* TLS library. Like projects, libraries vary greatly in size, complexity, and breadth of use.

This book focuses exclusively on libraries. While some open source projects can be consumed as both a tool and a library (for instance, in the form of a container), this book only considers the library aspect.

Application Versus Operating System Dependencies

Open source software (OSS) projects can be downloaded directly from their website or GitHub repository, but are primarily consumed through registries, which hold packaged and versioned project snapshots.

One class of registries holds operating system dependencies. For instance, Debian and Ubuntu systems use the apt registry to download utilities, Fedora and RedHat users leverage yum, and many Mac users use HomeBrew to install tools on their machines. These are often referred to as server dependencies, and updating them is typically called "patching your servers".

Another type of registry holds software libraries primarily meant to be consumed by applications. These registries are largely language specific—for example, pip holds Python libraries, npm holds Node.js and frontend JavaScript code, and Maven serves the Java and adjacent communities.

Securing server dependencies primarily boils down to updating your dependencies by running commands such as `apt-get upgrade` frequently. While real-world problems are never quite this simple, securing server dependencies is far better understood than securing application dependencies. Therefore, while much of its logic applies to libraries of all types, this book focuses exclusively on application dependencies.

Containers introduce some new complexity into handling OS dependencies. Containers are a part of the application, but they contain the OS, including its dependencies and their potential vulnerabilities. The guidance therefore works quite well for securing containers, too, but it focuses primarily on application dependencies.

To learn more about securing your servers, including their dependencies, check out Lee Brotherston and Amanda Berlin's *Defensive Security Handbook* (O'Reilly, 2017).

Known Vulnerabilities Versus Other Risks

There are multiple types of risks associated with consuming open source libraries, ranging from legal concerns with library license, through reliability concern in stale or poorly managed projects, to libraries with malicious or compromised contributors.

However, in my opinion, the most immediate security risk lies in known vulnerabilities in open source libraries. As I'll explain in the next chapter, these known vulnerabilities are the easiest path for attackers to exploit, and are poorly understood and handled by most organizations.

This book focuses on continuously finding, fixing, and preventing known vulnerabilities in open source libraries. Its aim is to help you understand this risk and the steps you need to take to combat it.

Comparing Tools

Tools that help address vulnerable libraries are often referred to as Software Composition Analysis (SCA) tools. This acronym doesn't represent the entire spectrum of risk (notably, it does not capture the remediation that follows the analysis), but as it's the term used by analysts, I will use it throughout this book.

Because the tooling landscape is evolving rapidly, I will mostly avoid referencing the capabilities of specific tools, except when the tool is tightly tied to the capability in question. When naming tools, I'll focus on ones that are either free or freemium, allowing you to vet them before use. Chapter 6 takes a higher level perspective to evaluating tools, offering a more opinionated view of which aspects matter most when choosing the solution.

Book Outline

Now that you understand the subject matter of this book, let's quickly review the flow:

- Chapter 1 defines and discusses known vulnerabilities and why it's important to keep abreast of them.
- Chapters 2 through 5 explain the four logical steps in addressing known vulnerabilities in open source libraries: finding vulnerabilities, fixing them, preventing the addition of new

vulnerable libraries, and responding to newly disclosed vulnerabilities.

- Chapter 6, as mentioned earlier, advances from explaining the differences between SCA tools to highlighting what I believe to be the most important attributes to focus on.
- Finally, Chapter 7 summarizes what we've learned, and briefly touches on topics that were not covered at length.

This book assumes that you are already familiar with the basics of using open source registries such as npm, Maven, or RubyGems. If you're not, it's worth reading up on one or two such ecosystems before starting on this book, to make the most of it.

CHAPTER 1

Known Vulnerabilities in Open Source Packages

A "known vulnerability" sounds like a pretty self-explanatory term. As its name implies, it is a security flaw that is publicly reported. However, due to the number and importance of these flaws, a full ecosystem evolved around this type of risk, including widely used standards and both commercial and governmental players.

This chapter attempts to better define what a known vulnerability is, and explain the key industry terms you need to understand as you establish your approach for addressing them.

Vulnerabilities in Reusable Products

Known vulnerabilities only apply to reusable products with multiple deployments, also referred to as third-party components. These products could be software or hardware, free or commercial, but they always have multiple instances deployed. If a vulnerability exists only within one system, there's no value in inventorying it and making other people aware of it (except, perhaps, among attackers).

Therefore, when we speak of known vulnerabilities we're only referring to reusable products. Because most known vulnerabilities deal with commercial products (the world of open source known vulnerabilities is a bit more nascent), the entity in charge of the product is often referred to as a *vendor*. In this book, because it deals with open

source packages and not commercial software, I'll refer to that entity as the *owner* or *author* of the package.

Vulnerability Databases

At the most basic level, a vulnerability is deemed *known* as soon as it's publicly posted in a reasonably easy to find location. Once a vulnerability is broadly disclosed, defenders can learn about it and protect their applications, but attackers—including automated or less sophisticated ones—also get the opportunity to easily find and exploit it.

That said, the internet is a big place, and holds a lot of software. New vulnerabilities are disclosed regularly, sometimes dozens in a single day. For defenders to be able to keep up, these vulnerabilities need to be stored in a central and easy to find location. For that purpose, quite a few structured databases, both commercial and open, have been created to compile these vulnerabilities and information about them, allowing individuals and tools to query test their systems against the vulnerabilities they hold.

In addition, there are several databases that focus on vulnerabilities in open source packages, such as the Node Foundation third-party vulnerability database, Rubysec, Snyk's database, and Victims DB. However, before digging into those, let's review the broader and more standardized foundations of the known vulnerability database world: CVE, CWE, CPE, and CVSS.

Known Vulnerabilities Versus Zero Days

Vulnerabilities can also be known to certain parties but not be publicly posted. For instance, bad actors often find vulnerabilities in popular libraries and sell them on the black market (often called the "dark web"). These vulnerabilities are often referred to as zero-day vulnerabilities, implying zero days have passed since its disclosure.

Common Vulnerabilities and Exposures (CVE)

The most well-known body of vulnerability information is Common Vulnerability and Exposures (CVE). CVE is a free dictionary of vulnerabilities, created and sponsored by the US government, main-

tained by the MITRE non-profit organization. While backed by the US government, CVEs are used globally as a classification system.

When a new vulnerability is disclosed, it can be reported to MITRE (or one of the other CVE Numbering Authorities), which can confirm the issue is real and assign it a CVE number. From that point on, the CVE number can be used as a cross-system identifier of the this flaw, allowing for easy correlation between security tools. In fact, most vulnerability databases will hold and share the CVE even when maintaining their own ID for the given vulnerability.

It's important to note that CVE is not itself a database, but rather a dictionary of IDs. To help automated systems access all CVEs, the US government also backs the National Vulnerability Database (NVD). NVD *is* a database, and exposes the vulnerability information through the standardized Security Content Automation Protocol (SCAP).

CVE is a pretty messy list of vulnerabilities, as it applies to systems of an extremely wide variety. To help keep it consistent and usable for its consumers, MITRE established various guidelines and policies around content and classification. The three most noteworthy ones, at least for the world of OSS libraries, are CPE, CWE, and CVSS.

Common Platform Enumeration (CPE)

In addition to the vast number of vulnerabilities they represent, CVEs also indicate flaws in extremely different products. In an attempt to make it easier for you to discover if a given CVE applies to your products, NVD can amend each CVE with one or more Common Product Enumeration (CPE) fields. A CPE is a relatively lax data structure that describes the product name and version ranges (and perhaps other data) this CVE applies to. Note that CPEs are not a part of CVE, but rather a part of NVD. This means a known vulnerability will not have product info unless it makes it to NVD, which doesn't always happen (more on that later on).

CPEs are a powerful idea, and can enable great automated discovery of vulnerabilities. However, defining a product in a generic way is extremely hard, and the content quality of many CVEs is lacking. As a result, in practice CPEs are very often inaccurate, partial, or simply not automation-friendly enough to be practical. Products that rely heavily on CPEs, such as the OWASP Dependency Check, need to

use fuzzy logic to understand CPEs and fail when content is lacking, leading to a large number of false positives and false negatives.

To address this gap, most commercial vulnerability scanners and databases in the OSS libraries space only use CPEs as a starting point, but maintain their own mappings of a CVE to the related products.

Common Weakness Enumeration (CWE)

While every vulnerability is its own unique snowflake, at the end of the day most vulnerabilities fall into a much more finite list of vulnerability types. MITRE classifies those types into the Common Weakness Enumeration (CWE) list, and provides information about each weakness type. While CWE items can get pretty specific (e.g., CWE-608 represents use of a non-private field in Struts' ActionForm class), its broader categories are more widely used. For instance, CWE-285 describes Improper Authorization, and CWE-20 represents the many variants of Improper Input Validation. CWEs are also hierarchal, allowing a broader scope CWE to contain multiple narrower scope ones.

The smaller number of CWEs makes it more feasible to provide rich detail and remediation advice for each item, or define policies based on a vulnerability type. Each CVE is classified with one or more CWEs, helping its consumers focus on the CWEs they're most interested in and removing the need to repeat CWE-level information for each associated CVE.

Common Vulnerability Scoring System (CVSS)

Vulnerabilities are disclosed on a regular basis, and at a rather alarming pace, but not all of them require dropping everything and taking action. For instance, leaking information to an attacker is not as bad as allowing them to remotely execute commands on a server. In addition, if a vulnerability can be exploited with a simple HTTP request, it's more urgent to fix than one requiring an attacker to modify backend files.

That said, classifying vulnerabilities isn't easy, as there are many parameters and it's hard to judge the weight each one should get. Is access to the DB more or less severe than remote command execution? If this execution is done as a low-privileged user, by how much should that reduce its severity score compared to an exploit per-

formed as root? And how to judge an exploit requiring a long sequence of requests, but that can be accomplished with a tool downloaded from the web?

On top of all of those, the severity of an issue also depends on the system on which it exists. For example, an information disclosure vulnerability is more severe on a bank's website than it is on a static news site, and a vulnerability that requires physical machine access matters more to an appliance than to a cloud service.

To help tackle all of these, MITRE created a Common Vulnerability Scoring System (CVSS). This system is currently at its third iteration, so you're likely to see references to CVSSv3, which is the version discussed here. CVSSv3 breaks up the scoring into three different scores, each of which are split into several smaller scores:

Base
: Immutable details about a vulnerability, including the attack vector, exploit complexity, and impact it could have.

Temporal
: Time-sensitive information, like the maturity of exploit tools out there or ease of remediation.

Environmental
: Context information for the vulnerable system, such as how sensitive it is or how is it accessible.

Figure 1-1 shows the variables and calculation of a CVSSv3 score for a sample vulnerability (the calculation was generated using FIRST.org's online tool).

While all three scores matter, public databases typically show CVSS scores based only on the Base component. The constant change in the Temporal score makes it costly to maintain, and the Environmental score is, by definition, specific to each vulnerability instance. Still, users of these databases can fill in those two scores, adjust the weights as they see fit, and get a final score.

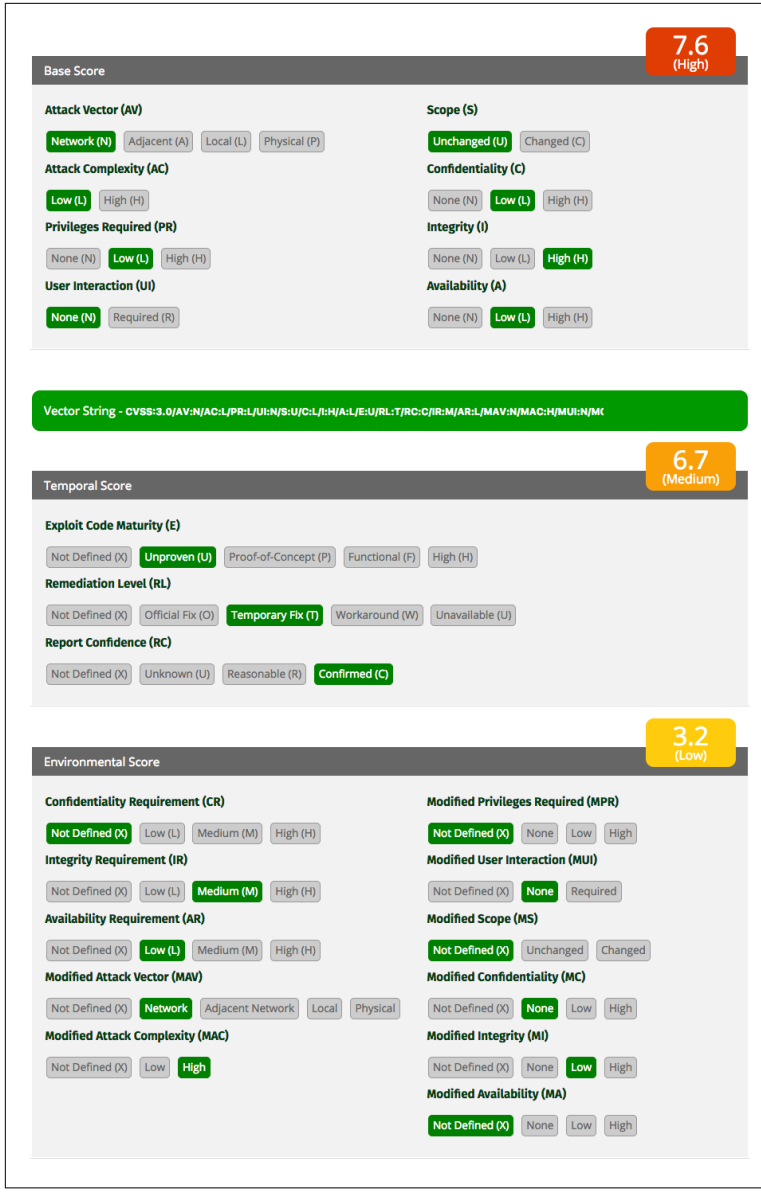


Figure 1-1. A completed CVSSv3 score

Known Vulnerabilities Outside CVE and NVD

Having a CVE is helpful, but it's also a hassle. Receiving a CVE number requires the author or reporter to know about it in the first place, and then go through a certain amount of paperwork and filing effort. Finally, CVEs need to be approved by specific CVE Numbering Authorities (CNA), which takes time. As a result, while CVEs are the norm for vulnerabilities in certain types of systems (e.g., network appliances), they're not that common in other worlds.

CVEs are in especially bad shape when it comes to known vulnerabilities in OSS packages. For instance, an analysis of one database shows only 67% of Ruby gem vulnerabilities have a CVE assigned. This stat drops to 45% for PHP (Composer) vulnerabilities, and a meager 27% for npm package vulnerabilities (both stats based on 2019 numbers up to May).

One reason for this gap is the fact many library vulnerabilities are reported by developers, not security researchers, and communicated as bugs. Such issues are often fixed quickly, but once fixed the author and reporter rarely go through the CVE process. Even when they do, the assignment of the CVE lags far behind, while attackers may be exploiting the vulnerability now made known.

Yet another lag takes place between the assignment of a CVE and posting it to NVD, providing more detail and perhaps CPEs. Such delays are expected when a new vulnerability reserves a number while it's going through a responsible disclosure process, but the lag often happens after the vulnerability is known.

This lag is very evident when looking at vulnerable Maven packages: 37% of Maven package vulnerabilities that have a CVE were public before being added to NVD, and 20% of those were public for 40 weeks or more before being added.

Known vulnerabilities that are not on NVD are still known, but harder to detect. Library vulnerabilities are likely to either have no CVE, not be listed on NVD (and thus have no advisory or CPE), or have poor-quality CPEs. Each of those would prevent their detection by tools that rely exclusively on these public data sources, notably OWASP Dependency Check.

Using CWE and CVSS Without CVE

While CVE, CWE, and CVSS are all MITRE standards, they can be used independently of one another. CWE and CVSS are often used for vulnerabilities that have no CVE, offering standardized classification and severity even if there is no industry-wide ID.

Unknown Versus Known Vulnerabilities

Every known vulnerability was at some point unknown. This may seem obvious, but it's an important point to understand—a new *known* vulnerability is not a new vulnerability but rather a newly *disclosed* one. The vulnerability itself was already there before it was discovered and reported. However, while disclosing a vulnerability doesn't create it, it does change how it should be handled, and how urgently it should be fixed.

For attackers, finding and exploiting an unknown vulnerability is hard. There are endless potential attack variants, which need to be invoked quickly while avoiding detection. Determining if an attack succeeded isn't always easy either, implying a submitted payload might have successfully gone through while the attacker remains none the wiser.

Once a vulnerability is disclosed, exploiting it becomes far easier. The attacker has the full detail of the vulnerability and how it can be invoked, and only needs to identify the running software (a process called fingerprinting) and get the malicious payload to it. This process is made even easier through automated exploit tools, which enumerate known vulnerabilities and their exploits. This automation also lowers the barrier to entry, allowing less sophisticated attackers to attempt penetration.

Known vulnerabilities are largely considered to be the primary cause for successful exploits in the wild. To quote two sample sources, Verizon stated "Most attacks exploit known vulnerabilities that have never been patched despite patches being available for months, or even years", and Symantec predicts that "Through 2020, 99% of vulnerabilities exploited will continue to be ones known by security and IT professionals for at least one year". On the application side, analyst firms such as Gartner and RedMonk have repeatedly stated the critical importance of dealing with known vulnerabilities in your open source libraries.

Known vulnerabilities should therefore be handled *urgently*. Even though it's the same vulnerability, its disclosure makes it much more likely attackers would use it to access your system. The disclosure of a vulnerability triggers a race, seeing whether a defender can seal the hole before an attacker can go through it. Figure 1-2 shows attackers response to the disclosure of the severe Struts2 vulnerability mentioned earlier, ramping from zero attacks to over a thousand observed attacks per day in a couple of days.

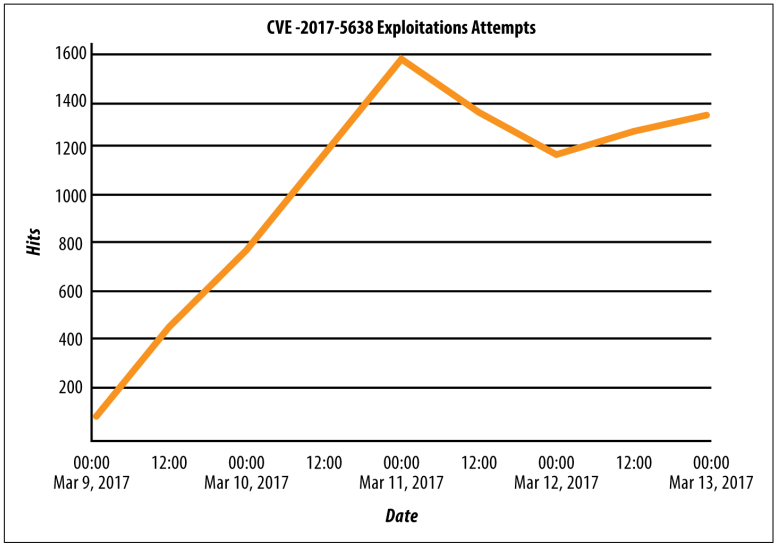


Figure 1-2. Daily exploitation attempts of the Struts vulnerability disclosed on March 9th kicked in immediately

The good news is that known vulnerabilities are also easier to defend against than unknown ones. Typically a known vulnerability also has a known solution, often in the form of a software upgrade or patch. Even if no software solution exists, you should at least have a better understanding of how to detect attacks and keep them from getting through with your security controls. As I'll discuss in Chapter 5, it's important to invest in systems that let you learn about such disclosures quickly and to act on them faster than the bad actors will.

Responsible Disclosure

So far I talked about a disclosure as a single point in time, when in fact it shouldn't be so binary. The right way to disclose a vulnerability, dubbed *responsible disclosure*, involves several steps that aim to give defenders a head start in the mentioned race.

Understanding responsible disclosure is not critical for open source consumers, but it is very important for open source authors. To learn more about responsible disclosure, you can read Tim Kadlec's excellent blog post, or look at Snyk's responsible disclosure as a template.

Summary

Known vulnerabilities are not as simple as they initially appear. The definition of what's considered *known* and the management of the vulnerability metadata are very hard to do well.

CVE and NVD work well for curating vulnerabilities in commercial products, but do not scale to the volume and ownership model of open source projects. Over time, these standards may evolve to meet this demand, but right now their coverage and detail level are not enough to protect your libraries.

CHAPTER 2

Finding Vulnerable Packages

Now that you understand what a known vulnerability is, let's start going through the four steps needed to address them: find, fix, prevent, and respond.

The first step in solving any problem is acknowledging you have one! And so, with vulnerable packages, your first act should be to look for vulnerable packages your application is consuming. This chapter discusses how to test your application, when and how you should run such a test, and the nuances in running the test and interpreting the results.

Taxonomy

Before you start testing, let's first discuss what you should anticipate seeing in the results.

Known Vulnerability Versus Vulnerable Path

Consider an app that uses two packages, A and B, wherein A uses B (with the same version) as well, resulting in the following dependency tree:

```
App
 ├─┬ A@1.0.0
 │ └── B@1.0.0
 └── B@1.0.0
```

Now, let's assume `B@1.0.0` has a known denial-of-service (DoS) vulnerability. When testing the app, how many vulnerabilities should

we report? On one hand, the app only has one known vulnerability: DoS in `B@1.0.0`. On the other hand, there are two instances of this vulnerability. What should we report: one or two?

The answer, unfortunately, is both, as each number is right in its own way. Different tools may count the two differently, but for the purpose of this book I'll use the terms *known vulnerability* and *vulnerable path* to separate the two.

A known vulnerability represents a unique ID of a vulnerability in your tree (in this example, one). For each known vulnerability, you'll want to inspect its severity and determine how important it is for your systems. For instance, a DoS vulnerability typically matters more to a public-facing site than one on the intranet. This inspection is only needed once per known vulnerability, regardless of how many times it repeats in your dependency tree.

Once you have determined that a vulnerability is sufficiently concerning, you'll want to inspect its vulnerable path for exploitability. Each vulnerable path implies a different potential way for an attacker to get a malicious payload to the package in question. In this example, an attacker can potentially get a malicious payload to `B` through `App`'s direct use of it, or through `A`, requiring you assess both the `App->B` and `App->A->B` paths to see if they allow that to happen. Both paths also need to be fixed independently, which I'll discuss in the next chapter.

It's worth noting that vulnerable paths are counted on the logical dependency tree, not the number of library copies. For instance, if our sample app was a Ruby or Maven app, which use global dependencies, there would only be one instance of `B@1.0.0` on disk, but there would still be two paths to assess and remediate. In fact, ecosystems which use version ranges and global dependencies, such as Ruby, Java, and Python, often have a large number of vulnerable paths for each known vulnerability.

Figure 2-1 shows a dependency tree where the same vulnerable *negotiator* library is used (indirectly) by two direct dependencies, *errorhandler* and *express*. The application therefore has one known vulnerability, but two vulnerable paths. Attackers may be able to exploit the vulnerability through the app's use of either *errorhandler* or *express*, and each of those would need to be independently upgraded to trigger a downstream upgrade to *negotiator* and fix the vulnerability.

Figure 2-1. In negotiator, goof has one known vulnerability, but two vulnerable paths

Testing Source Versus Built Apps

Roughly speaking, there are two ways to test for vulnerable application dependencies: before or during a build, when you have only your source code, or after a build, when you have the resulting application and have fetched its dependencies. While logically both tests should yield a list of vulnerable dependencies, they have some important differences.

Testing source code

When testing source code, the testing tool needs to anticipate which dependencies your application will pull in when built. This means fetching the original manifest files, such as a *package.json* or *pom.xml*, extracting the specified dependencies, and building the dependency graph. This graph is then intersected with the vulnerability database, resulting in a list of vulnerable dependencies.

Extrapolating the tree has a fair bit of complexity to it. The Software Composition Analysis (SCA) tool has to mimic the logic of the package manager, along with the instructions in the manifest. For instance, it has to understand and interact with the package manager to resolve version ranges (e.g., what's the version this npm range will yield `qs@^0.5.3`?), factor in multiple manifest and lock files (e.g., handle parent *pom.xml* files in Maven, or combine *Gemfile*

and *Gemfile.lock* in Ruby), and separate between dev-time dependencies and production libraries.

While source code scanning is harder for the tool, it's actually easier for the user to consume. Building the graph is much faster than building the application, making tests dramatically faster than deployed code tests. This speed is good all around, but is especially handy when testing code changes for vulnerabilities—for instance, built into a GitHub pull request or running in the IDE. More on that in Chapter 4.

In addition, source code scanning ensures an accurate understanding of how a package got into your system, as the testing tool is the one constructing the dependency graph. This results in a more accurate report of *vulnerable paths*, and better filtering of dev dependencies and other package manager subtleties. More importantly, this complete understanding of the tree is critical for automatically fixing issues, as we'll see in the next chapter.

The weakness of source code scanning is that build processes that aren't fully standardized may throw it off—for instance, by manually pulling in a library not specified in a manifest file or using different library sets in different contexts. In such cases, source code scanning may remain ignorant of the fact this library is being used, and won't report on its vulnerabilities.

Fundamentally, the dependency graph built is only an *approximation* of what the real graph would be. Good SCA tools approximate well in the vast majority of cases, but the more complex and unique your build process is, the more likely they will miss or inaccurately report a library in your tree.

It's worth noting that in environments that use dependency lock files, such as Ruby and Yarn (or newer versions of npm), the list of libraries is explicitly defined and enforced, ensuring the approximation is accurate. In such environments, source code scanning is far more accurate, but tools still need to understand how to build a graph to provide the deep dependency perspective we just discussed.

Testing built apps

The other way to determine which dependencies are used is to look on disk. When running the scanner, it reads the relevant folders and environment variables and sees which libraries are installed. This list

is intersected with the vulnerability DB to determine which vulnerabilities are present.

The key advantage of testing built apps is that it anchors on the app that was actually deployed. As long as the tool can properly identify the app's dependencies, the list it finds is the actual list of dependencies installed, regardless of how they got there. This makes the security report inherently more accurate—there's no guessing or approximation, just stating the facts (assuming the tool's detection is accurate, of course).

The main disadvantage of this approach is that it cannot report on vulnerable paths. Detecting which vulnerable components are installed does not tell you which code parts use them, crippling your ability to assess exploitability. This is somewhat similar to testing for operating system dependencies—you can tell you're using a vulnerable OpenSSL, but cannot tell which apps on your machine are using this library. Not mapping the vulnerable paths may flag issues in libraries you're not really using, and eliminates the ability to calculate an automated fix, as we'll see later on.

A more minor disadvantage is the fact it requires building the app. Depending on where you run the test, building the app may be hard or impossible, and is often slow. This limits the integration points in which you can run such a test.

Combining code and built app testing

Because neither approach is perfect, the best solution is to use both!

To combine the two, certain tools test *built* apps, but also look for package manifest files, build a logical tree, and try to fit the libraries found on top of the tree. As a result, you can be certain you get the built-app testing coverage, with (to the extent possible) the source code–driven depth of understanding.

Alternatively, you can use both approaches at different stages of your development. For instance, you can use source code scanning as part of your GitHub/GitLab/BitBucket workflow, while using a built app test in your build process or as a deployment gate. In the majority of cases, the two approaches would yield the same results and act consistently. When they don't, the built app tests will ensure you don't deploy a vulnerability.

Testing indirect dependencies

Whenever testing for vulnerable dependencies, it's critical you don't just test your direct dependencies, but also the indirect ones. In other words, if you're using package A, and it uses package B, you should make sure you're testing B for flaws as well.

This may seem obvious, but it's actually overlooked by *many* tools, especially when testing source code. Parsing manifest files is relatively easy, while expanding those files to their dependency trees is hard, and so many tools only do the former but still claim to protect your dependencies.

Let's look at some data. A scan of over a million snapshot projects and has discovered that vulnerabilities in indirect dependencies account for 78% of overall vulnerabilities. This further amplifies a critical need for clear insight into the dependency tree and the need to be able to correctly highlight nuances of a vulnerable path in order to address these vulnerabilities.

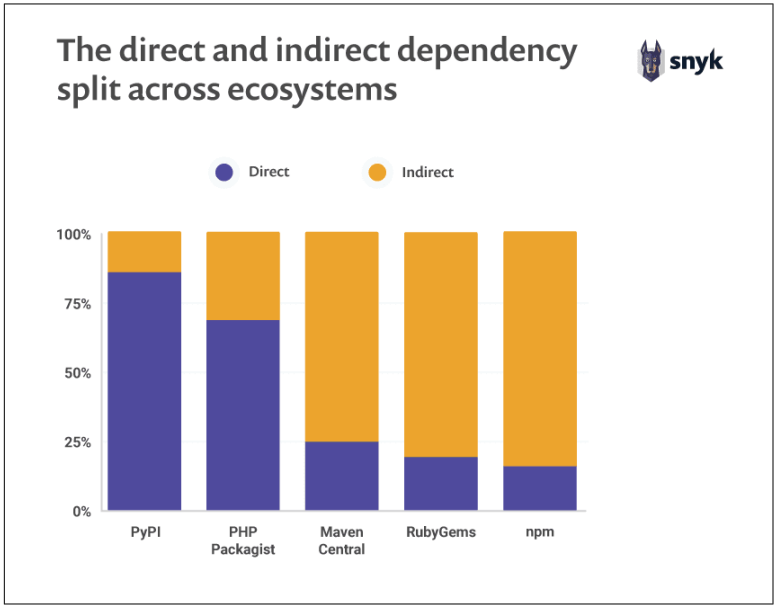


Figure 2-2. Share of direct vs indirect dependencies by ecosystem

Only testing direct dependencies is extremely flawed, and in most ecosystems the amount of risk reduction it offers is so low it is practically worthless. Throughout this book I will assume testing is done

on both direct and indirect dependencies, and I encourage you to *make sure your tool does the same*.

Finding Vulnerabilities Using the Command Line

Finally, let's find some vulnerable libraries!

Vulnerable Demo Apps

For brevity, this book doesn't include detailed execution of the tools, but I'd encourage you to try out the mentioned tools on your own applications. You can also use sample vulnerable applications such as goof for npm or java-goof for Java, to ensure you get results.

The lowest common denominator for OSS vulnerability scanning is the command-line interface. Most tools support a CLI (many are CLI only), including the OWASP Dependency Check, RubySec, and more. With the exception of a few tools, running a CLI test is often a quick and extremely easy action. The tool is often installed from the package manager itself (e.g., `npm install -g sca-tool`), and execution is typically a simple *test* command (e.g., `sca-tool test`).

Figure 2-3 shows a truncated example of running a CLI test on `goof`.

Figure 2-3. A partial output of CLI test

The output varies by tool, but most will list either the known vulnerabilities or vulnerable paths, and provide information for each of them. This typically includes a title, description, severity score, and more. Severity is the easiest indicator of how urgently you should address the issue, but the remaining details can help you understand the issue and how it relates to your code.

Because the terminal is a limited presentation layer, the detailed advisory is usually in a web page linked from the output, as are more details about the dependency tree. If you want to generate your own reports, look for a machine-friendly format option (e.g., `--json`) which lets you filter, manipulate, or reformat the results. Figure 2-4 shows an example of a detailed web-based advisory linked from the more limited command line interface

When using a CLI, make sure you filter out dependencies that don't actually make it to production. CLI tests typically run in a build or dev environment, which contain dev dependencies (e.g., test libraries). These libraries don't actually get deployed, and so security flaws in them are of far lesser importance. Ensure your tool or your process filter out such dependencies to spare yourself unncessary work.

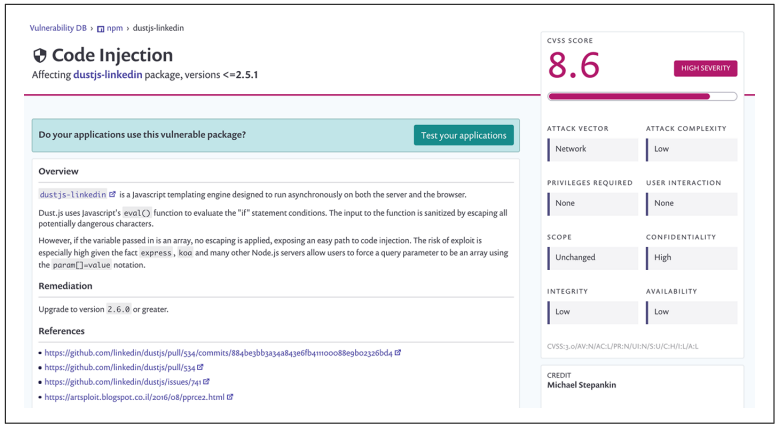


Figure 2-4. Advisory about a code injection vulnerability in LinkedIn's Dust.js templating library

The command-line interface is also a very popular way to integrate vulnerable dependency testing into your continuous integration/continuous deployment (CI/CD). More on that in Chapter 4.

Finding Vulnerabilities in SCM (GitHub, BitBucket, GitLab)

The CLI tools are great and flexible, but running them on every single app—let alone integrating them continuously—takes some effort. To make testing (and fixing) even easier, some SCA tools can

also integrate testing directly into source code management (SCM) platforms such as GitHub, BitBucket, and GitLab.

Testing SCM repos is almost always a matter of a few clicks via a web interface. It should include the following steps:

1. A user asking to test their repos.
2. Granting the service access to the repos, so it can extract the manifest files.
3. Choosing which repositories to test or protect.
4. Viewing a report.

Using an SCM integration offers a few advantages over the CLI:

- Easier to test or browse multiple apps at once
- The web interface is naturally richer than the terminal, often allowing better usability
- It simplifies continuous testing and fixing, as we'll see later on

That said, testing your repos isn't always better than a CLI. For starters, many of us prefer the CLI to the web interface at times. Second, the CLI is far more flexible, while an SCM integration tends to be more opinionated. Finally, testing via SCM naturally means using source code testing, as opposed to the CLI's built apps testing, which has the pros and cons discussed earlier.

Granting Source Code Access

If you've been reading carefully, you may have noticed that I glossed over a security aspect of the SCM integration: granting source code access. Testing directly against source code repos requires access, both at the network level and API tokens. Network access is an issue for on-premises source code management tools, such as GitHub Enterprise or BitBucket Server, and granting token access is additional exposure not everybody is comfortable with.

The good news is that scanning source code *repositories* for vulnerable dependencies doesn't really require source code access. The trees are extrapolated from manifest files, such as *package.json* and *pom.xml*, and the tools can typically work well without having access to your actual source code - even if granted access to the repositories. Some commercial SCA tools support installing por-

tions of the service on-prem, enabling `git` integration while keeping the service's access to a minimum.

Finding Vulnerabilities in Serverless and PaaS

So far I've only discussed testing as part of your dev or build processes. However, newer infrastructure paradigms such as Function-as-a-Service (FaaS, a.k.a Serverless) and Platform-as-a-Service (PaaS) open up another possibility.

Both FaaS and PaaS abstract the infrastructure from the app developer, and seamlessly handle operating system dependencies. Some platforms, such as Heroku, CloudFoundry, and Google Cloud Functions, go on to build the apps for you, fetching the dependencies stated in your package manifest. However, no platform goes as far as managing those dependencies and informing you when they went out of date or vulnerable. Over time, these libraries inevitably grow stale and insecure, opening a path for attackers to walk in.

Because PaaS and FaaS apps are, in essence, only code, testing these apps is not all that different than what I've discussed so far. And indeed, certain commercial SCA tools added capabilities to directly test PaaS and/or Faas apps for vulnerable dependencies.

Testing FaaS/PaaS apps is somewhat similar to testing repos on an SCM. The service connects to the platform in question (e.g., AWS Lambda, Azure Functions), the user grants access and chooses the apps to test, and a test is run on each app. Unlike SCM, however, the test consists of downloading the *built* app (e.g., Lambda ZIP, Cloud Foundry droplet), extracting it, and running a built-app test on it.

Testing deployed apps doesn't replace the need to scan for vulnerabilities earlier, but it offers a valueable additional layer of protection. Connecting to the deployed platforms and scanning all the deployed apps ensures full coverage of your apps and their dependencies, regardless of when and how they were deployed.

Finding Vulnerabilities in Containers

While PaaS and FaaS still enjoy limited adoption, a far greater number of modern applications are deployed using containers. For these containerized applications, the build output grows to include not only the application, but a full container image—complete with the

operating system and its dependencies. Therefore, scanning such apps for vulnerabilities needs to include this additional scope, capturing the operating system dependencies and intersecting them with an appropriate vulnerability database.

Scanning container images is very similar to scanning applications. Your solution should identify the components used in the image, intersect them with a vulnerability database, and present quality results. Most tools support a CLI-based test of container images, but advanced solutions offer more options.

That said, container image scanning has three important unique perspectives to consider: registry scans, base image handling, and scanning packaged apps.

Scanning Registries

Container images are stored and distributed via container registry, such as Docker Hub, the different cloud platform registries, RedHat's Quay and others. Since the images are stored in and fetched from those registries, they represent a compelling vulnerability scanning target as well.

Some container scanning solutions are able to connect to a registry, show the images stored inside, and scan them for vulnerabilities. Such scans may be easier to deploy as they can be done from the outside, not requiring any changes to your build flow.

Functionally, scanning a container image from a registry is identical to scanning it using a CLI. The delta is purely in your development workflow, allowing you to pick the easiest way to integrate such a scan into your SDLC.

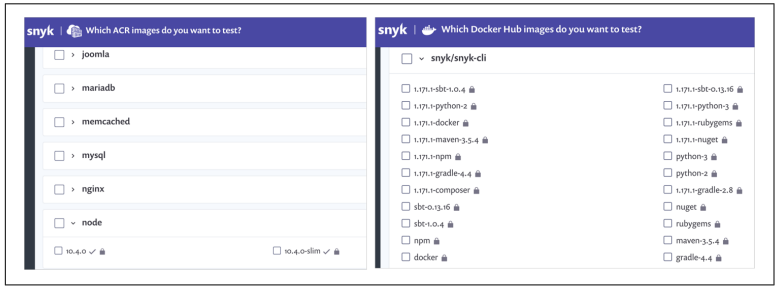


Figure 2-5. Selecting which container images in a registry to scan and monitor for vulnerabilities

Scanning base images

Container images are typically built using a base image, such as *node* or *ubuntu*, and extending it with app specific components. While a container image scan should capture all components with it, most dependencies—and thus most vulnerabilities—will likely arrive as part of the base image. In addition, since the same base images are reused across companies, attackers focus on vulnerabilities within them more than ones specific to the components in your app.

Beyond their importance, base image vulnerabilities are also harder to triage. These are third-party components, and your dev team is unlikely to know why a component in them is installed or how it's used. When finding a vulnerability in a base image, your first action is likely to see if a newer version of the base image can make it go away. If not, brace yourself for a more complicated triage process.

For all these reasons, it's important to use a solution that clearly calls out whether an identified vulnerability comes from the base image or a lower layer. Certain solutions can even flag vulnerabilities in base images in SCM as well, making it even easier to deploy. Separating the handling of base image vulnerabilities will help you prioritize your response, and actually drives different remediation actions, as we'll discuss in the next chapter.

Scanning Packaged Container Apps

While this book deals mostly with securing libraries, not apps, it's important to note that containers make it very easy to consume full apps packaged into a container, such as *mongo* or *nginx*. These packaged apps are also container images, deployed into your Kubernetes cluster just like the images you built yourself.

Scanning packaged container images for vulnerabilities is almost the same as scanning a base image. They are third-party components you're unlikely to understand, are attractive targets for attackers due to their prevalence, and often have a new version that simply makes the security flaws disappear.

When embracing a container vulnerability scanner, I fully recommend using it for such packaged container apps, too.

Finding Vulnerabilities in the Browser

This last example is a bit unique in its approach, and that is testing for vulnerable JavaScript libraries right on a web page itself.

JavaScript is the most widely used language today, and like all languages consumes many open source libraries. Over time, security flaws in these libraries surface just like they do in other platforms, except in this case the vulnerable libraries are public for everyone to see! In fact, a 2017 study found that 77% of the top websites use at least one vulnerable library—a mind-blowing number.

Unfortunately, web pages often consume libraries through simple copy/paste, linking to hosted URLs, or other messy manners, which makes detecting these libraries hard. For a while, the only option for detecting such libraries was using an open source project called *Retire.js*, which offers a Chrome extension (and a few other means) to detects libraries. *Retire.js* continues to be a great tool you should consider using.

More recently, though, Google and Microsoft integrated detection of vulnerable JS libraries into their web development tools, Lighthouse and Sonar. Lighthouse is also embedded in Chrome's and Opera's browser dev tools, further simplifying and raising the visibility of this test. This is a promising new development, and I encourage you to try these tools out.

Vulnerable Component Versus Vulnerable Apps

This book deals with bugs—specifically security bugs—in your dependencies. However, just because a library has a bug, doesn't mean your code triggers it! Many packages offer rich functionality, and your application (or another one of its dependencies) may not invoke the code paths in question, and may never trigger the bug. In other words, the fact that you're using a vulnerable library doesn't mean your app is vulnerable.

This is an important but dangerous distinction. On one hand, it is an entirely accurate statement. In fact, I'd go as far as estimating a typical app is not vulnerable to *most* of the vulnerabilities in its dependencies. Many of these vulnerabilities only happen under cer-

tain configurations, and in many cases you're consuming a full package only to use a thin slice of it.

On the other hand, a vulnerable dependency is a ticking time bomb. Applications are complicated, change frequently, and use other dependencies that are also complex and fast moving. Confidently saying your application *never* uses a library in a certain way, across all vulnerable paths, is hard to do. Moreover, once a dependency is included in your app, it's likely to be used by more and more code paths. If you dismissed a vulnerability in this dependency, another code path may expose this vulnerability to attackers at a later time.

Certain tools are able to highlight the vulnerable libraries that your code actually calls, helping you prioritize remediating them over the rest. This can be be partially done using static analysis, which is convenient but less accurate (due to the complexity of code). Alternatively, it can be achieved by monitoring which libraries are actually loaded in runtime, which requires a bit more investment but offers richer and more accurate insight.

My recommendation is to bias in favor of not allowing vulnerable components in your apps. If possible, fix the vulnerability to remove all doubt and keep your system safe. In cases where you cannot fix, prioritize handling those issues you deem most likely to be exploitabile in your specific app.

Summary

Finding vulnerable libraries is the clear first step in addressing this risk. By taking off your blindfolds, you'll be able to understand the risk and take the appropriate action.

Amidst the many decisions, the main point to keep in mind is comprehensiveness. The solution(s) you choose should find these loopholes across all of your apps, and then match them against a rich vulnerability DB. Without such coverage, you won't have the visibility you need to truly address this risk.

CHAPTER 3

Fixing Vulnerable Packages

Finding out if you're using vulnerable packages is an important step, but it's not the real goal. The real goal is to fix those issues!

This chapter focuses on all you should know about fixing vulnerable packages, including remediation options, tooling, and various nuances. Note that SCA tools traditionally focused on finding or preventing vulnerabilities, and most put little emphasis on fix beyond providing advisory information or logging an issue. Therefore, you may need to implement some of these remediations yourself, at least until more SCA solutions expand to include them.

There are several ways to fix vulnerable packages, but upgrading is the best choice. If that is not possible, patching offers a good alternative. The following sections discuss each of these options, and we will later take a look at what you can do in situations where neither of these solutions is possible.

Upgrading

As I've previously stated, a vulnerability is a type of bug, and the best way to address a bug is to use a newer version where it is fixed. And so, the best way to fix a vulnerable dependency is to upgrade to a newer version. Statistically, most disclosed vulnerabilities are eventually fixed. In npm, 59% of reported vulnerabilities have a fix. In Maven, 90% are remediable, while that portion is 85% in Ruby-

Gems.[1] In other words, more often than not, there is a version of your library where the vulnerability is fixed.

Finding a vulnerable package requires knowledge of which versions are vulnerable. This means that, at the very least, every tool that finds issues can tell which versions are vulnerable, allowing you to look for newer versions of the library and upgrade. Most tools also take the minor extra step of determining the minimal fixed version, and noting it in the advisory.

Upgrading is therefore the best way to make a vulnerability go away. It's technically easy (update a manifest or lock file), and it's something dev teams are very accustomed to doing. That said, upgrading still holds some complexity.

Major Upgrades

While most issues are fixed, very often the fix is only applied to the latest and greatest version of the library. If you're still using an older version of the library, upgrading may mean switching to a new major version. Major upgrades are typically not backward compatible, introducing more risk and requiring more dev effort.

Another reason for fixing an issue only in the next major version is that sometimes fixing a vulnerability means reducing functionality. For instance, fixing a certain XSS vulnerability in a jQuery 2.x codebase requires a change to the way certain selectors are interpreted. The jQuery team determined too many people are relying on this functionality to deem this a non-breaking change, and so only fixed the vulnerability in their 3.x stream.

For these reasons, a major upgrade can often be difficult, but if you can accept it, it's still the best way to fix a vulnerability.

Indirect Dependency Upgrade

If you're consuming a dependency directly, upgrading is relatively straightforward. But what happens when one of your dependencies is the one who pulled in the vulnerable package? Most dependencies are in fact indirect dependencies (a.k.a. transitive dependencies), making upgrades a bit more complex.

1 Stats based on vulnerabilities curated in the Snyk vulnerability DB.

The cleanest way to perform an indirect upgrade is through a direct one. If your app uses `A@1`, which uses a vulnerable `B@1`, it's possible that upgrading to `A@2` will trigger a downstream upgrade to `B@2` and fix the issue. Applying such an upgrade is easy (it's essentially a direct upgrade), but discovering *which* upgrade to do (and whether one even exists) is time consuming. While not common, some SCA tools can determine and advise on the *direct* upgrades you need to make to fix an *indirect* vulnerability. If your tooling doesn't support it, you'll need to do the searching manually.

Old vulnerabilities in indirect libraries can often be fixed with a direct upgrade, but such upgrades are frequently unavailable for new issues. When a new vulnerability is disclosed, even if the offending package releases a fix right away, it takes a while for the dependency chain to catch up. If you can't find a path to an indirect upgrade for a newly disclosed flaw, be sure to recheck frequently as one may show up soon. Once again, some SCA tools will do this monitoring for you and alert you when new remediations are available.

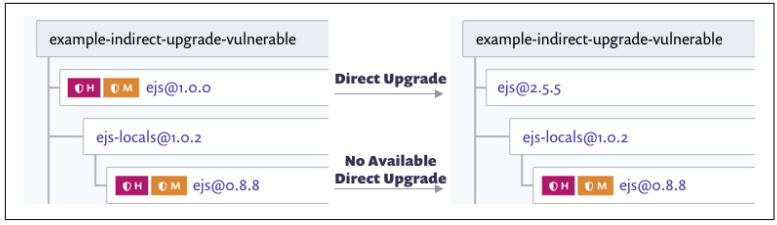


Figure 3-1. The direct vulnerable EJS can be upgraded, but indirect instance cannot currently be upgraded

Conflicts

Another potential obstacle to upgrading is a conflict. Many languages, such as Ruby and Python, require dependencies to be global, and clients such as Ruby's bundler and Python's pip determine the mix of library versions that can co-exist. As a result, upgrading one library may trigger a conflict with another. While developers are adept at handling such conflicts, there are times when such issues simply cannot be resolved.

On the positive side, global dependency managers, such as Ruby's bundler, allow the parent app to add a constraint. For instance, if a downstream `B@1` gem is vulnerable, you can add `B@^2` to your Gem-

file, and have bundler sort out the surrounding impact. Adding such constraints is a safe and legitimate solution, as long as your ecosystem tooling can figure out a conflict-free combination of libraries.

Is a Newer Version Always Safer?

The conversation about upgrading begs a question: can a vulnerability also be fixed by downgrading?

For the most part, the answer is no. Vulnerabilities are bugs, and bugs are typically fixed in a newer version, not an older one. In general, maintaining a good upgrade cadence and keeping your dependencies up to date is a good preventative measure to reduce the risk of vulnerabilities.

However, in certain cases, code changes or (more often) new features are the ones that trigger a vulnerability. In those cases, it's indeed possible that downgrading will fix the discovered flaw. The advisory should give you the information you need about which versions are affected by the vulnerability. That said, note that downgrading a package puts you at higher risk of being exposed to new issues, and can make it harder to upgrade when that happens. I suggest you see downgrading as a temporary and rarely used remediation path.

There Is No Fixed Version

Last on the list of reasons preventing you from upgrading to a safe version is such a version not existing in the first place!

While most vulnerabilities are fixed, many remain unfixed. This is sometimes a temporary situation—for instance, when a vulnerability was made public without waiting for a fix to be released. Other times, it may be a more long-term scenario, as many repositories fall into a poor maintenance state, and don't fix reported issues nor accept community patches.

In the following sections I'll discuss some options for when you cannot upgrade a vulnerability away.

Patching

Despite all the complexity it may involve, upgrading is the best way to fix an issue. However, if you cannot upgrade, patching the vulnerability is the next best option.

Patching means taking a library as is, including its vulnerabilities, and then modifying it to fix a vulnerability it holds. Patching should apply the minimal set of changes to the library, so as to keep its functionality unharmed and only address the issue at hand.

Patching inevitably holds a certain amount of risk. When you use a package downloaded millions of time a month, you have some assurance that bugs in it will be discovered, reported, and often fixed. When you download that package and modify it, your version of the code will not be quite as battle tested.

Patching is therefore an exercise in risk management. What presents a greater risk: having the vulnerability, or applying the patch? For well-managed patches, especially for ones small in scope, I believe it's almost always better to have a patch than a vulnerability.

It's worth noting that patching application dependencies is a relatively new concept, but an old hat in the operating system world. When dealing with operating system dependencies, we're accustomed to consuming a feed of fixes by running `apt-get upgrade` or an equivalent command, often remaining unaware of which issues we fixed. What most don't know is that many of the fixes you pull down are in fact back-ported versions of the original OS author code changes, created and tested by Canonical, RedHat, and the like. A safe registry that feeds you the non-vulnerable variants of your dependencies doesn't exist yet in the application libraries world, but patching is sometimes doable in other ways.

Sourcing Patches

To create a patch, you first need to have a fix for the vulnerability! You could write one yourself, but patches are more often sourced from existing community fixes.

The first place to look for a patch is a new version of the vulnerable package. Most often the vulnerability *was* fixed by the maintainers of the library, but that fix may be in an out-of-reach indirect dependency, or perhaps was only fitted back into the latest major

version. Those fixes can be extracted from the original repo and stored into their own patch file, as well as back-ported into older versions if need be.

Another common source for patches are external pull requests (PRs). Open source maintenance is a complicated topic, and it's not uncommon for repos to go inactive. In such repos, you may find community pull requests that fix a vulnerability, have been commented on and perhaps vetted by others, but are not merged and published into the main stream. Such PRs are a good starting point—if not the full solution—for creating a patch. For instance, an XSS issue in the popular JavaScript Markdown parsing library marked had an open fix PR for nearly a year before it was incorporated into a new release. During this period, you could use the fix PR code to patch the issue in your apps.

Snyk maintains its own set of patches in its vulnerability database. Most of those patches are captures or back-ports of original fixes, a few are packaged pull requests, and even fewer are written by the Snyk security research team.

Depend on GitHub Hash

In very specific cases, you may be able to patch without storing any code changes. This is only possible if the vulnerable dependency is a direct dependency of your app, and the public repo holding the package has a commit that fixes the issue (often a pull request, as mentioned before).

If that's the case, most package managers allow you to change your manifest file to point to the GitHub commit instead of naming your package and version. Git hashes are immutable, so you'll know exactly what you're getting, even if the pull request evolved. However, the commit may be deleted, introducing certain reliability concerns.

Fork and Patch

When patching a vulnerability in a direct dependency, assuming you don't want to depend on an external commit or have none to use, you can create one of your own. Doing so typically means forking the GitHub repository to a user you control, and patching it. Once done, you can modify your manifest to point to your fixed repository.

Forking is a fairly common way of fixing different bugs in dependencies, and also carries some nice reliability advantages, as the code you use is now in your own control. It has the downside of breaking off the normal version stream of the dependency, but it's a decent short-term solution to vulnerabilities in direct dependencies. Unfortunately, forking is not a viable option for patching indirect dependencies.

Static Patching at Build Time

Another opportunity to patch a dependency is during build time. This type of patching is more complicated, as it requires:

1. Storing a patch in a file (often a *.patch* file, or an alternative JAR file with the issue fixed)
2. Installing the dependencies as usual
3. Determining where the dependency you'd like to patch was installed
4. Applying the patch by modifying or swapping out the risky code

These steps are not trivial, but they're also usually doable using package manager commands. If a vulnerability is worth fixing, and there are no easier means to fix it, this approach should be considered.

This is a classic problem for tools to address, as patches can be reused and their application can be repeated. However, at the time of this writing, Snyk is the only SCA tool that maintains patches in its DB and lets you apply them in your pipeline. I predict over time more and more tools will adopt this approach.

Dynamic Patching at Boot Time

In certain programming languages, classes can also be modified at runtime, a technique often referred to as "monkey patching." Monkey patching can be used to fix vulnerabilities, though that practice has not become the norm in any ecosystem. The most prevalent use of monkey patching to fix vulnerabilities is in Ruby on Rails, where the Rails team has often released patches for vulnerabilities in the libraries it maintains.

Other Remediation Paths

So far, I've stated upgrades are the best way to address a vulnerability, and patching the second best. However, what should you do when you cannot (or will not) upgrade nor patch?

In those cases, you have no choice but to dig deeper. You need to understand the vulnerability better, and how it plays into your application. If it indeed puts your application at notable risk, there are a few steps you can take.

Removal

Removing a dependency is a very effective way of fixing its vulnerabilities. Unfortunately, you'll be losing its functionality at the same time.

Dropping a dependency is often hard, as it by definition requires changes to your actual code. That said, such removal may turn out to be easy—for instance, when a dependency was used for convenience and can be rewritten instead, or when a comparable alternative exists in the ecosystem.

Easy or hard, removing a dependency should always be considered an option, and weighed against the risk of keeping it.

External Mitigation

If you can't fix the vulnerable code, you can try to block attacks that attempt to exploit it instead. Introducing a rule in a web app firewall, modifying the parts of your app that accept related user input, or even blocking a port are all potential ways to mitigate a vulnerability.

Whether you can mitigate and how to do so depends on the specific vulnerability and application, and in many cases such protection is impossible or high risk. That said, the most trivially exploited vulnerabilities, such as the March 2017 Struts2 RCE and ImageTragick, are often the ones most easily identified and blocked, so this approach is definitely worth exploring.

Protecting Against Unknown Vulnerabilities

Once you're aware of a known vulnerability, your best move is to fix it, and external mitigation is a last resort. However, security controls that protect against unknown vulnerabilities, ranging from web app firewalls to sandboxed processes to ensuring least privilege, can often protect you from known vulnerabilities as well.

Log Issue

Last but not least, even if you choose not to remediate the issue, the least you can do is create an issue for it. Beyond its risk management advantages, logging the issue will remind you to re-examine the remediation options over time—for instance, looking for newly available upgrades or patches that can help.

If you have a security operations team, make sure to make them aware of vulnerabilities you are not solving right now. This information can prove useful when they triage suspicious behavior on the network, as such behavior may come down to this security hole being exploited.

Remediating Container Vulnerabilities

For the most part, fixing a vulnerability in a container image is very similar to what we've described so far. Vulnerabilities may exist in direct or indirect dependencies, upgrades are the best way to resolve them, and patches are a harder but viable option.

Container vulnerability remediation differs in two areas—the value of rebuilding, and the likelihood of reaching zero vulnerabilities.

Rebuild as a Remediation

Like it or not, container images and OS dependencies rarely "pin" their dependencies. In the vast majority of cases, labels such as `node` or `gradle` are used to represent the most recent version of that image family, and only major versions get their own stream (e.g., Ubuntu's `trusty`). Developers are very welcome to use specific versions, such as `trusty-20190515`, or even specific SHA values, but they rarely do.

Even within the image, OS dependencies rarely state exact version. Commands such as `apt-get install curl` don't specify a version of curl, and, again, only big and breaking changes such as `yum install python3` get their own package repository ID.

This means building container images is not a deterministic action. Building a container image today may create a very different image than the one built yesterday with the exact same source file.

When scanning for vulnerabilities, scanners must capture the specific version of every image and dependency to accurately inform you about vulnerabilities. Similarly, reporting must have clear timestamps, and the time an app was *built* matters greatly.

When fixing vulnerabilities, it means a simple rebuild—without any code changes—is likely to make many vulnerabilities go away! Make sure the solution you use clearly informs you whether fixing a vulnerability requires a simple rebuild or actual code changes, and be ready to perform either quickly.

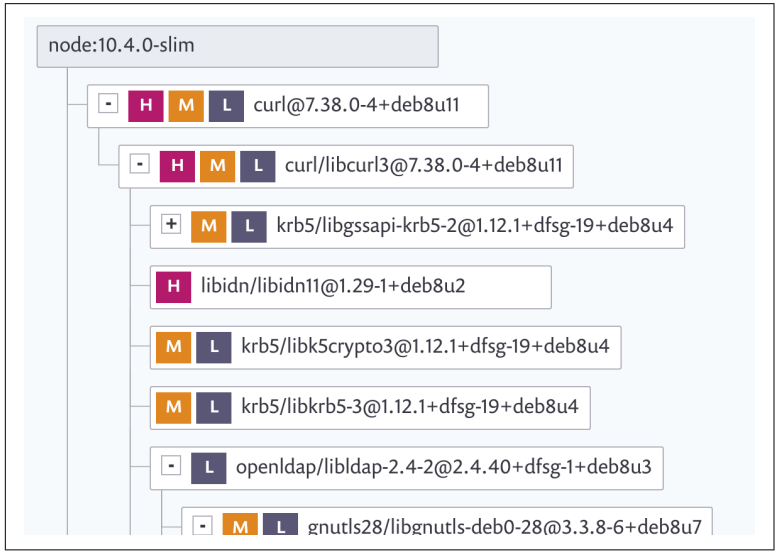


Figure 3-2. Most container vulnerabilities are in indirect dependencies, and many can be fixed with a simple rebuild

Reaching Zero Vulnerabilities

The other sad fact about containers and OS dependencies is that there are many vulnerabilities without an available fix. When scan-

ning a container image, you're very likely to encounter dozens of vulnerabilities for which no fix has been published.

This reality makes it especially important to have good triaging information. Specifically, it's important to have the following:

- Curated CVSS vectors, helping you identify which issues are severe overall but also map those to your surroundings.
- High quality and detailed advisories, especially for high-severity issues, explaining the issue and potential mitigation techniques
- Detailed dependency graphs, showing how a component made it into the image in the first place, helping relate the issue to your app

Runtime monitoring for which components are actually loaded in production, mentioned in Chapter 2, is an another extremely effective way to focus your attention. It isn't a prevalent capability in today's solution, but hopefully one that will become more available soon.

Remediation Process

Beyond the specific techniques, there are few broader guidelines when it comes to remediating issues.

Ignoring Issues

If you choose not to fix an issue, or to fix it through a custom path, you'll need to tell your SCA tool. Otherwise, the tool will continue to indicate this problem.

All OSS security tools support ignoring a vulnerability, but have slightly different capabilities. You should consider the following, and try to note that in your tool of choice:

- Are you ignoring the issue because it doesn't affect you (perhaps you've mitigated it another way) or because you've accepted the risk? This may reflect differently in your top-level reports.
- Do you want to mute the issue indefinitely, or just "snooze" it? Ignoring temporarily is common for low severity issues that don't yet have an upgrade, where you're comfortable taking the risk for a bit and anticipate an upgrade will show up soon.

- Do you want to ignore all instances of this known vulnerability (perhaps it doesn't apply to your system), or only certain vulnerable paths (which, after a careful vetting process, you've determined to be non-exploitable)?

Properly tagging the reason for muting an alert helps manage these vulnerabilities over time and across projects, and reduces the chance of an issue being wrongfully ignored and slipping through the cracks.

Fix All Vulnerable Paths

For all the issues you're not ignoring, remember that remediation has to be done for *every vulnerable path*.

This is especially true for upgrades, as every path must be assessed for upgrade separately, but also applies to patches in many ecosystems.

Track Remediations Over Time

As already mentioned, a fix is typically issued for the vulnerable package first, and only later propagates through the dependency chain as other libraries upgrade to use the newer (and safer) version. Similarly, community or author code contributions are created constantly, addressing issues that weren't previously fixable.

Therefore, it's worth tracking remediation options over time. For ignored issues, periodically check if an easy fix is now available. For patched issues, track potential updates you can switch to. Certain SCA tools automate this tracking and notify you (or open automated pull requests) when such new remediations are available.

Invest in Making Fixing Easy

The unfortunate reality is that new vulnerabilities in libraries are discovered all the time. This is a fact of life—code will have bugs, some of those bugs are security bugs (vulnerabilities), and some of those are disclosed. Therefore, you and your team should expect to get a constant stream of vulnerability notifications, which you need to act on.

If fixing these vulnerabilities isn't easy, your team will not do it. Fixing these issues competes with many priorities, and its oh-so-easy to

put off this invisible risk. If each alert requires a lot of time to triage and determine a fix for, the ensuing behavior would likely be to either put it off or try to convince yourself it's not a real problem.

In the world of operating systems, fixing has become the default action. In fact, "patching your servers" means taking in a feed of fixes, often without ever knowing which vulnerabilities we fix. We should strive to achieve at least this level of simplicity when dealing with vulnerable app dependencies too.

Part of this effort is on tooling providers. SCA tools should let you fix vulnerabilities with a click or proactive pull requests, or patch them with a single command like `apt-get upgrade` does on servers. The other part of the effort is on you. Consider it a high priority to make vulnerability remediation easy, choose priority, choose your tools accordingly, and put in the effort to enrich or adapt those tools to fit your workflow.

Summary

You should always keep in mind that finding these vulnerabilities isn't the goal—fixing them is. Because fixing vulnerabilities is something your team will need to do often, defining the processes and tools to get that done is critical.

A great way to get started with remediation is to find vulnerabilities that can be fixed with a non-breaking upgrade, and get those upgrades done. While not entirely risk-free, these upgrades should be backward compatible, and getting these security holes fixed gets you off to a very good start.

CHAPTER 4

Integrating Testing to Prevent Vulnerable Libraries

Once you've found and fixed (or at least acknowledged) the security flaws in the libraries you use, it's time to look into tackling this problem continuously.

There are two ways for additional vulnerabilities to show up in your dependencies:

- Code changes added a vulnerable library to the app
- A new vulnerability in a dependency the app is already using was disclosed

To protect yourself, you need mechanisms to prevent vulnerable packages from being added, and to ensure you get alerted and can quickly respond to new vulnerability disclosures. This chapter will focus on the first concern, discussing how you can integrate SCA vulnerability testing into your process, and prevent the addition of new vulnerable libraries to your code. The next chapter will deal with responding to new issues.

Preventing new security flaws is conceptually simple, and very aligned with your (hopefully) existing quality control. Because vulnerabilities are just security bugs, a good way to prevent them is to test for them as part of your automated test suite.

The key to successful prevention is inserting the vulnerability test into the right steps in the process, and deciding how strict to make

it. Being overly restrictive early on may hinder productivity and create antagonism among your developers. On the flip side, testing too late can make fixing issues more costly, and being too lenient can eventually let vulnerabilities make it to production. It's all about finding the right balance for your team and process.

Here are a few considerations on how to strike the right balance.

When to Run the Test?

Before diving into the details of the test itself, let's review the most common spots in your workflow to integrate such a test (most of these should look familiar, as they are the same integration opportunities for other quality tests):

CI/build
: The most intuitive integration is to "break the build" if a code change introduced a new vulnerability. Because builds are triggered in various scenarios, you can choose to only run the test in a particular build context (e.g., before merging to the master branch).

CD/release
: A similar gate in which you can block deployment of a new vulnerable package is an automated release or deployment process. This is a common filter for non-negotiable quality criteria, and can serve as a security gateway too.

Pull request tests
: Testing as part of a `git pull` or `git merge` request is similar to CI testing, but is a bit more visible, and can test only the current code changes (as opposed to the entire resulting application). More on that delta later on.

IDE integration
: Probably the earliest time to test is to do so as you're writing the code. While it risks being annoying, testing as you type surfaces the problem extremely early, when it's easiest for the developer to take a more secure path.

Platform-specific hooks
: Lastly, some platforms offer specific deployment hooks. You can run a test before a `git push`, as part of a Kubernetes admission controller before a container starts running, inside a Heroku or

Cloud Foundry buildpack, or as a plug-in to the Serverless framework. Many platforms offer test hooks, which may work well for such a vulnerability test.

This is not a comprehensive list, as in practice you can run tests for vulnerable packages at any point you run an automated test. In addition, not all SCA tools support all of these integration points. Of these, CI and pull-request integrations are the ones most commonly used.

Blocking Versus Informative Testing

Assume you've run a test and found some vulnerable libraries. What should you do now? Should you inform the developer the best way you can, and let the process continue, or do you block the sequence altogether? This is a decision you have to make every time you integrate a vulnerability test.

For the most part, early tests are better off being informative as opposed to blocking. Testing in the IDE offers a great opportunity to highlight a problem to the user as they type, but would not be well received if it doesn't allow the app to build. Similarly, pull request tests are often non-blocking, surfacing a flaw to the user but letting them push through if they think that's best.

On the flip side, later tests should default to breaking the automated flow. Such gates get closer to (or potentially are the last step before) deploying the vulnerability and exposing it to attackers, which is what we're trying to prevent. If you do make them informative, note that developers rarely look at the build or deployment output unless they failed to complete. This means you'll have to make sure they are informed about found issues in a specific way, typically via the capabilities the testing tool offers or a specific customization.

You can further refine this decision through policies. For instance, it's common to break the build on high-severity issues, but keep low-severity issues as informative only. Alternatively, you may choose to be very strict on your most sensitive production systems, while keeping such tests as a "best effort" for internal utilities that don't touch customer data.

Failing on Newly Added Versus Newly Disclosed Issues

When building CI tests, it's important to keep them consistent from build to build. Having a build that fails intermittently is extremely frustrating to developers, and can waste countless hours. The best practice is typically to ensure that a build that succeeded once will, if rerun without changes, succeed again.

Vulnerability testing, however, is different. As I mentioned, vulnerabilities are disclosed in *existing* components, that previously succeeded when being tested for known security flaws. SCA tests typically use the latest vulnerability DB whenever they change, which means the same test that previously succeeded may now fail, as new vulnerability information came to light.

This can present a problem both in early and late testing. Earlier on, a developer may commit code changes that have nothing to do with any dependency, only to have the build fail because of a newly disclosed vulnerability in a package the app has been using long before. To get the build to complete, the dev would need to upgrade or modify the use of this dependency, a time-consuming task they may not even be able or allowed to do. Later in the pipeline, breaking the build due to new information can prevent re-deployments, which may include such a test, hindering the agility and responsiveness of the organization.

An alternative path is to change the test to examine if a vulnerability was *added*, as opposed to whether a vulnerability exists. Unfortunately, CI systems (and other gates) tend to test the build as a standalone state, and it's hard—if not impossible—to find out what changes triggered the process. Practically speaking, CI/CD tests flag all vulnerabilities found, regardless if triggered by code changes or new disclosures.

Pull request (PR) testing, however, is designed precisely for such a scenario. When testing as part of a PR, it's easier to test both the current branch and the branch you're about to merge to, and only fail the test if you're about to introduce a new problem. PR tests also happen at a good time for a security check—just before a single developer's changes are merged into the team's stream. Some SCA tools offer PR testing as part of the product, and you can create your own PR tests to invoke some of the others. Figure 4-1 shows an

example of failing a test on pull request because it contained code changes that introduced a new security risk.

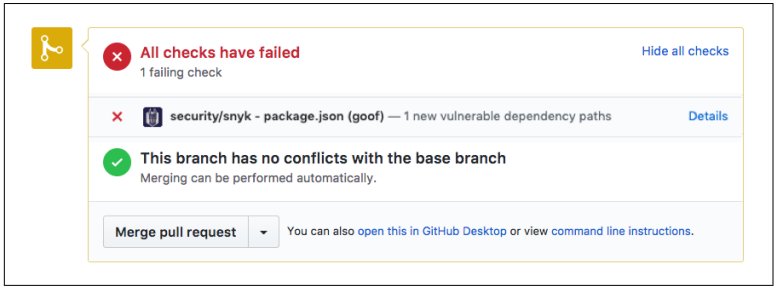


Figure 4-1. Failing a pull request test due to a newly added vulnerable library

To get the best of both worlds and the risk/disruption balance that's right for you, consider alternating between changeset testing and complete testing at different gates. For instance, you can introduce informative-only PR tests early in the process, alerting developers to the problem, which would (hopefully) get many of them to fix it early and cheaply. In addition, you can add a strict security gate as part of your deployment process that tests the entire application and disallows vulnerable libraries regardless of where they came from (unless they were explicitly acknowledged, as described in "Ignoring Issues" on page 35).

Platform-Wide Versus App-Specific Integration

When integrating SCA testing, you can also choose between a "top down" approach, where you integrate the test into a continuous or runtime platform and run it on all applications, and a "bottom up" approach, where a single app (and later more of them) adds this test to their code.

App-specific integrations are more aligned with other quality tests you perform. For the most part, functional tests are created and run for each application, possibly leveraging some testing infrastructure you set in place. This means you can run the test at the right time for each specific application, handle its results as befits this app's nature, and empower developers to own this responsibility.

Platform-wide testing is a better way to ensure the testing is done, and enforce compliance on all applications. You can ensure vulnerable libraries are addressed throughout, and set the policies of which risks are acceptable and which ones are not. When testing centrally, you should decide the level of control each application can get. For instance, can an app opt out of this testing? Can a developer ignore an issue? Even platform-wide testing would likely require some form of app-specific controls.

A good rollout approach is to start with app-specific testing and, once you're ready, expand to protecting your applications platform wide. Doing so does have some technical tool requirements, though. For app-specific integrations, check if your SCA tool can be invoked as a CLI tool, or can integrate with a platform but only be "turned on" for specific applications. For platform-wide integration, confirm the solution can then expand to integrate with all applications, and apply central policies as well. Choosing a tool that fits the evolution you envision can save you significant pain down the road.

Integrating Testing Before Fixing

If you weren't testing for vulnerable libraries until now, odds are your initial tests would find many flawed dependencies. Fixing those libraries, even if you make it as easy as you can, can take a considerable amount of time. In the meantime, your developers will continue to add new libraries to your application, and new vulnerabilities will be found in the dependencies you already use.

As is the case with many quality concerns, a useful first step when tackling this problem is to try and set a baseline, and introduce integrated testing to make sure you don't slip past it. Doing so will help ensure you're not getting worse, while buying you time to eat away at the problems that exist in the system today.

There are two ways to baseline vulnerable library use. The first is to ignore all current issues, and just start testing from now on. This approach is effective in keeping the tools from alerting on old issues, but is clearly poor from a security perspective, as it hides real and immediate security risks you've not addressed. Still, it's a possible way to go, and you can separately run other tests that don't ignore the issues, and trigger steps to fix the found issues through that path.

The second way is to integrate tests so that they only break on issues introduced by code changes, but disregard past issues. As described before, this approach is most natural in the context of pull request tests, but may be doable in other testing contexts, or supported in the SCA tool you use.

Integrating testing before taking on the broader fix is a powerful way to roll out an SCA tool. It lets you get started right away, will prevent real and immediate problems, and will raise awareness to this risk among your developers. Over time, this approach will also gradually trigger upgrades and fixes to your dependencies, eating away at your security technical debt.

In addition, gradual rollout will set you up to track your dependencies across the board, which in turn will help you respond better to newly disclosed vulnerabilities via notifications, as we'll see in the next chapter.

Summary

Preventing the addition of new vulnerable libraries is a key part in making such protection continuous. When doing so, the key trade-off is between blocking a build or deployment and informing developers. Once that's decided, proceed to choosing where in the workflow and toolchain to place such gates or information.

My recommendation is to start by integrating informative testing of change-sets, for instance in the pull requests, which keeps you from slipping while raising awareness to this concern among your team. In fact, I can't think of any reason *not* to flag new vulnerable libraries during the development process.

CHAPTER 5

Responding to New Vulnerability Disclosures

The techniques to find, fix, and prevent vulnerable dependencies are very similar to other quality controls. They revolve around issues in our application, and maintaining quality as the application changes. The last piece in the vulnerable library puzzle is a bit different.

In addition to their known vulnerabilities, the libraries you use also contain *unknown* vulnerabilities. Every now and then, somebody (typically a library's authors, its users, or security researchers) will discover and report such a vulnerability. Once a vulnerability is discovered and publicly disclosed, you need to be ready to test your applications for it and fix the findings quickly—before attackers exploit it.

The Significance of Vulnerability Disclosure

A new vulnerability disclosure is not a new vulnerability. The security flaw existed in your library's code all along, introducing a weakness in your application. However, when a vulnerability is made public, its likelihood of being exploited skyrockets.

Consider the effort required for an attacker to find a vulnerability in a library. There are literally millions of open source libraries out there, with tens of thousands of them in heavy use. A large portion of these libraries are in active development, with regular code changes introducing new functionality and fixing old bugs. Many of

these packages are also quite complex, reaching many thousands of lines of code. Understanding the code in all of these libraries and finding a security flaw is a massive undertaking.

When a vulnerability is disclosed, an attacker is spared all this effort. Instead, they can jump straight to the next steps: assessing whether the vulnerability is interesting enough to exploit, building such an exploit if so, and seeking out potential victims through fingerprinting or automated tests.

It's no wonder that once a severe vulnerability is disclosed, it quickly shows up in automated attack tools around the world, running up the exploit counts as attackers rush to capitalize on the new low-hanging fruit. Long story short, a known vulnerability requires far more urgent action than a hidden one.

Setting Up for Quick Remediation

The good news is that a vulnerability disclosure also makes it easier for defenders to act. Once a vulnerability is made known, you can find the applications using the library in question and fix or otherwise mitigate the risk.

Because vulnerabilities are discovered often, it's critical to make this process quick and efficient, allowing you to remediate the issues faster than attackers can exploit them. Setting up for fast response includes a few steps:

1. Monitor which dependencies each application is using.
2. Get a feed of vulnerability notifications.
3. Automate matching and notification of a new disclosure to your dependencies.
4. Automate the remediation steps.

Let's dig into each of these steps in a bit more detail.

Monitoring Which Dependencies Your Apps Are Using

To be able to ascertain if a future vulnerability affects you, it's critical to know which applications you have, and which libraries each application uses. The enumeration of the artifacts (including libra-

ries) each app is using is often referred to as the application's Bill of Materials (BOM), a term borrowed from the world of physical manufacturing, describing the materials used to create the final product.

Keeping the BOM up to date is critical, as an outdated one can easily cause more harm than good. There are three primary ways to keep it continuously updated:

- Monitoring your source code management (SCM) platforms
- Integrating into your continuous deployment (CD)
- Monitoring deployed applications

Let's delve into the details of each.

Source Code Management Platform Integration

The first way to track your dependencies is by attaching directly to your source code management (SCM) platform, such as GitHub, BitBucket, or GitLab. When using an SCM, there is almost always a "master" branch that represents the code currently deployed to production, including the dependency-related manifest files.

Monitoring dependencies in SCM is elegant and powerful. It enables great prevention and remediation capabilities, as discussed in previous chapters, and allows for automated remediation pull requests, as I'll explain shortly. I highly recommend monitoring the dependencies used via SCM integration. That said, monitoring source code can sometimes be inaccurate, due to several scenarios.

First, as I've explained in Chapter 2, source code scanning requires the testing tool to approximate what would be the eventual dependency tree. This approximation can be very accurate (depending on the tool you use), but it's not as perfect as testing a built app, especially in platforms that do not use a lock file or applications using a more complex build flow.

Second, source code is *always* a few steps ahead of deployed code. By definition, you modify your source code, test and evolve it, and *then* deploy. In the fastest scenario, merges to the master branch are immediately rolled out, meaning the master source code branch is just minutes (or even seconds) ahead of the deployed code. Most setups, however, are not quite that fast, allowing source code to evolve for hours, days, or weeks before deploying it. Therefore, if

you're monitoring your dependencies in SCM, you may be monitoring tomorrow's dependencies and be blind to vulnerability disclosures that affect the libraries used in production today.

Lastly, monitoring source code doesn't capture potential complexity in how you deployed code in the past. Are you using a *canary* version of your code to test out new changes? Do you have an older version of the application deployed to support a large historic customer? Did someone manually publish a modified version of your app to some machine when addressing an ops problem? Source code monitoring would remain blind to such scenarios.

Fundamentally, monitoring dependencies by integrating into your SCM is a great first step, as it's the easiest and most elegant integration into your developer's workflow, and greatly simplifies prevention and remediation steps. However, to be fully protected, you should *also* monitor dependencies in deployed code.

Monitoring Deployed Code

If source code is the start of the dependency journey, the final destination is the deployed application. As I mentioned in Chapter 2, looking for vulnerable libraries in a deployed application gives you the most accurate picture of the security flaws you actually deployed, regardless of how they got there.

The tools you need to continuously monitor deployed apps depends on the platform running these apps. If you are managing your own servers, the common way to monitor deployed applications is using infrastructure monitoring (IM) products, which many endpoint security solutions include. These tools are not aware of specific applications, but rather attempt to discover which components exist on every system (e.g., host, container), and report on known flaws in them. The tools in the infrastructure monitoring are quite evolved and do a great job reporting on vulnerable OS dependencies, but typically fall short or completely ignore vulnerable application dependencies. If you're using such a tool, make sure to inspect how well it would report on issues in application dependencies.

Infrastructure monitoring products unfortunately don't work for applications deployed on a Function-as-a-Service (a.k.a FaaS or Serverless), such as AWS Lambda or Azure Functions, nor for Platform-as-a-Service (PaaS) solutions like Heroku and Cloud Foundry. In their absence, some SCA tools can connect directly to

such platforms and inspect each app for vulnerable libraries. As the marketplaces around PaaS and FaaS offerings evolve, I expect we'll see more continuous monitoring solutions show up, including ones inspecting for vulnerable libraries.

For containers, properly labeled registries offer another way to track deployed code. SCA solutions that support containers can often connect to registries and thus alert you if an image with labels implying it's in production use has a vulnerability in it. Connecting to Kubernetes to see which container images are actually run is another way to track this, though this feature is less prevalent amidst the tools.

An alternative to monitoring deployed code is to take a snapshot of the dependencies used as part of the application's (hopefully continuous) deployment process, which brings us to the third common location for monitoring the dependencies used.

Integrating into Continuous Deployment

The path from source code to a built application is captured in our deployment process. If you have a continuous deployment system, you can use it to track which dependencies you're *about to deploy*, which you could then use to monitor for vulnerabilities over time. Note that unlike the "prevent" step, which could run in either CI or CD, updating the BOM should be integrated into your continuous deployment workflow, but not the CI (i.e., you shouldn't update the BOM every time you build, only when you're about to deploy).

Different tools offer different means to track your BOM in CD. Platform integrations, such as a Jenkins Plugin or Heroku Add-on, integrate into the build system via plug-ins and relate the downloaded packages to the builds to automatically track the BOM. CI systems like Jenkins allow you to store metadata with each build, which can be used to include the BOM. And other tools let you explicitly take a snapshot by calling an API or CLI command at the right time.

Capturing the BOM during CD is typically more accurate than source code, and can be *as* accurate as monitoring built apps if you also know which apps are deployed and where, so you can respond accordingly. In reality, it tends to work better for capturing the primary and most active applications, which tend to be tracked better, but can get complicated in tracking the long tail of apps you have

deployed, or in more elaborate deployment scenarios (e.g., having multiple live versions in parallel).

There's no single perfect place to track your BOM, as each phase has pros and cons. My recommendation is to track it as part of your source code always, and to amend that by either tracking it in CD or monitoring deployed apps.

Getting a Feed of Vulnerability Notifications

Now that we know the dependencies used at any given time, we need the second data feed: a feed of known vulnerabilities.

If you're using a commercial SCA tool, odds are it already includes a vulnerability DB within. When running tests using an online solution, the DB is likely to be updated continuously. However, if the SCA tool you use supports running without internet connectivity and you use it in that fashion, you will need to download the DB frequently to ensure it's up to date.

If you are monitoring your libraries for vulnerabilities without a complete SCA solution, or wish to augment such a tool, you can also license a vulnerability DB directly. Many SCA vendors will offer such a license, and some independent threat intelligence companies sell such vulnerability feeds as products in and of themselves. When licensing a DB directly, make sure it covers open source *libraries* well, as most vulnerability DBs focus on operating system dependencies but contain minimal information about security flaws in application dependencies.

While a vulnerability DB is always a part of your SCA treatment, the different databases vary greatly in the coverage they offer. The number and caliber of DBs change regularly, requiring you to evaluate the quality of your solution's DB before you buy. It's also important to assess the DB against your technology stack, as some DBs do better in certain ecosystems (e.g., Maven) but have minimal coverage for others (e.g., npm), or vice versa.

Beyond the specific differences, there are two particular aspects of a DB we can call out: non-CVE vulnerabilities and early notifications.

CVEs Are Not Enough

As I mentioned at the start of this book, known vulnerabilities are often classified in the public MITRE DB and assigned a CVE. The existence and details of CVEs are in the public domain, allowing all tools to easily query for them and include them in the testing. Many of the open source SCA tools, notably OWASP Dependency Check, rely exclusively on CVEs and the Common Product Enumeration (CPE) specifications in them to detect vulnerable libraries.

Unfortunately, getting a CVE is a hassle, and because developers and SCA vendors are offered little incentive to file for one, the coverage, accuracy, and content quality of CVEs is severely lacking in the world of open source libraries. CVEs do not capture most new vulnerabilities, the specifications (e.g., vulnerable version ranges) of the vulnerabilities they do capture are often incorrect, and the descriptions they include are typically too short to be helpful.

Many commercial SCA solutions go beyond CVEs and curate their own databases with additional vulnerabilities and richer data, finding issues more accurately and helping to fix them better. When using an SCA solution, check that its DB tracks non-CVE vulnerabilities well.

The failure of the CVE system to properly contain known vulnerability information is not ideal, and has sparked many conversations about potential long-term solutions that aren't solely commercial. Hopefully it'll be solved in the future, but in the meantime, if you want to properly secure your libraries, you will do well not to rely on CVE information alone.

Early Notifications

By definition, known vulnerabilities are public knowledge, and commercial vulnerability databases often track other databases to discover and grow their own lists. In fact, most SCA solutions use a DB that is purely based on copying other databases, and are practically never the first to disclose a new issue. As a consumer, this back-scene process doesn't impact you greatly, as long as the DB you are using is built legally and is sufficiently accurate and comprehensive.

If you *are*, however, using a DB that is first to uncover issues, you can take advantage of that by signing up to early notifications. These

DBs typically make certain customers aware, confidentially, of vulnerabilities that are going through a responsible disclosure process, allowing them to address these issues before the rest of the world (including attackers) finds out about them. Early notifications are especially valuable for high-value attack targets, such as financial institutions, governments, and similar organizations, as their attackers are often faster then average, while as organizations they're slower than average to properly defend.

Early notifications require, by definition, exclusivity. If a service offered early notifications to its free or low-cost customers, attackers can easily sign up to receive it as well, defeating the entire point. Therefore, early notifications are, without fail, offered only to enterprise customers or an otherwise hand-curated list of companies. If you work at a large organization and would like to stay a step ahead, look for a DB that is frequently first to disclose vulnerabilities, and inquire about early notification options.

Automating Matching and Notification

Given a DB and the list of dependencies, you're ready for the final move: intersecting the list of dependencies with the list of vulnerabilities. As long as you have an accurate set of dependencies and vulnerabilities, this step is not that hard—it simply requires looking up each library and version pair in your list in the DB.

Once a vulnerability is found, you can send off notifications in the manner you see fit. As expected, SCA tools offer various notification capabilities, ranging from emails to web hooks to slack notifications. You should also consider more subtle aspects of the notifications, such as which issues merit a notification (e.g., only alert on high-severity issues).

Notifications are a well-understood topic in the world of security and ops, and the majority of SCA functionality in this space is not at all unique, so I will not spend more time on those. I will, however, focus on more SCA-specific topics—notifying dev versus ops, breaking the build on new vulnerabilities, and automating remediation steps.

Who You Should Notify and How

There are at least two audiences that need to find out about a new vulnerability disclosure: security operations and dev.

Security operations (or regular operations in teams that have no SecOps) are the typical recipient of security alerts. They are hopefully familiar with assessing the severity of an incoming issue, responding with the right urgency, and staying alert for potential exploits of this new threat. This team should, at the very least, be alerted on high-severity issues.

The dev team will need to do the actual fixing, and so should be notified as well. Dev teams are accustomed to getting external bug reports as well, typically in the form of a logged issue. What isn't standard, though, is how to address the urgency of the issue. You should inspect your internal processes to determine the best way to raise urgency in addressing a high-severity security issue, to keep it from being lost in the backlog, or simply pushed to the next sprint.

Automating Remediation Steps

While logging an issue is a standard way to alert the dev team, an even better way is to submit a pull request with the changes needed to fix the problem. The best bug reports are those that come with a fix built into them!

Certain SCA tools offer automated fix pull requests as part of their notifications flow. If connected to your source control, such tools can determine the changes needed to fix the issue (as explained in Chapter 3), and proactively open a pull request with those changes. Automated fix PRs are good for various reasons, including:

- Clear indication of what it takes to fix the issue, saving quite a few steps in the triage process
- Faster remediation, as opening a fix PR with the needed changes is a step a developer would have to do anyway
- Better visibility, as typically dev teams are faster to notice and respond to PRs than they are to issues
- Less likelihood of being ignored, as dev teams aim to not keep PRs open forever, and are more likely to either close or merge the issue

Fix PRs are a great way to expedite remediation, but they don't go all the way. If your system allows it, consider keying off these fix PRs to trigger a non-production deployment and all the tests that go with it. Ideally, by the time a human looks at the issue and approves the fix, the fixed version is just one step away from production. Figure 5-1 shows an automated fix pull request to a newly disclosed vulnerability, made possible by using source code scanning.

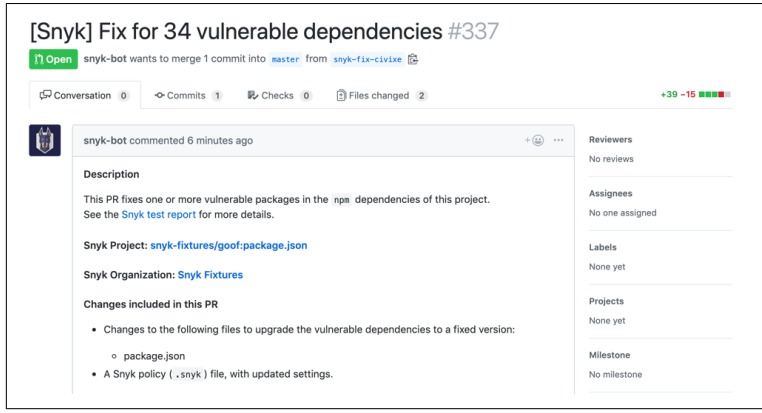


Figure 5-1. Example of a fix pull request created when a new vulnerability was disclosed

Breaking a Build on a New Vulnerability

Chapter 4 discussed the impact of breaking builds on newly disclosed vulnerabilities versus newly added vulnerabilities. Failing builds (or other gates) is also a form of notification, but a somewhat brutal one. There's no point in repeating the information from Chapter 4 about testing changesets versus builds, but it's worth referring back to "Failing on Newly Added Versus Newly Disclosed Issues" on page 42 when planning your notifications process.

Becoming Vulnerable Due to Dependency Chain Updates

This chapter focused on responding to new vulnerability disclosures that affect you, which is the only way your deployed applications may suddenly have known vulnerabilities. However, your *source code* may also become vulnerable because of changes in its dependency tree.

For instance, say your app depends on package `A@^1.0.0`, meaning `A` at version 1.0.0 or newer, but less than `2.0.0`. Let's also assume version `A@1.0.1` is the latest version, and has no further dependencies. Now, what happens if a new version is released, `A@1.0.2`, which uses a vulnerable package `V`?

The applications you deployed yesterday would remain unharmed, as they were built before `A@1.0.2` came out. They resolved the version range to `A@1.0.1`, downloaded it, and deployed a vulnerability-free app. However, if you build your app again, without any code changes whatsoever, the version range will resolve to `A@1.0.2` and the build will pull in the vulnerable package `V`!

A similar scenario may occur during conflict resolution or de-duplication performed by your package manager. It's quite possible a new dependency, either direct or indirect, would change the eventual versions picked for every library, and introduce a new vulnerability.

While not as severe as a new disclosure, it's important to monitor your *source* code for these types of changes as well, to facilitate fixes. Because your deployed applications are unharmed, there is no need to look for these issues in built apps. SCA tools that can monitor source code are typically able to address this concern.

It's worth noting that this scenario only happens if you are using version ranges and not using a lock file. If you are using a pinned version (most common in Maven) or using a lock file that explicitly states each library's version (e.g., *Gemfile.lock*, *yarn.lock*), your library versions are guaranteed to stay the same.

Summary

Responding to newly disclosed vulnerabilities is where your true defensive skills shine. Each vulnerability disclosure triggers a race between attacker and defender, and while attackers are fine with only an occasional win, you need to beat them every single time.

The key to success lies in speed. Learn about the issues quickly, map them to the relevant vulnerable application, and either fix or mitigate the risk. Just like I stated in the summary to Chapter 2, fast response without comprehensive coverage is not going to be enough, so make sure all your apps are under control.

CHAPTER 6

Choosing a Software Composition Analysis Solution

Continuously tracking your application's dependencies for vulnerabilities and efficiently addressing them is no simple feat. In addition, this is a problem shared by all, and is not an area most companies would consider their core competency. Therefore, it is a great opportunity for the right set of tools to help tackle this concern.

As mentioned before, the category of tools addressing this concern is currently known as Software Composition Analysis (SCA). Throughout this book, I've referred to different capabilities an SCA solution may or may not have, and the implications therein. Those questions were meant to assist you in designing the right process and selecting the right tools to help.

In this brief chapter, I'd like to offer my opinions about which properties you should care about the most when choosing a tool. The SCA tooling landscape is evolving at an extremely fast pace, so I'll avoid making statements about which tools handle each requirement well, and instead try to stick to the concepts, which will hopefully stay true over a longer period of time!

Choose a Tool Your Developers Will Actually Use

To successfully deal with open source security, you need your developers (and DevOps teams) to operate the solution. Given the fast

pace of modern development, boosted in part by the use of open source itself, an outnumbered security team will never be able to keep you secure. Therefore, the SCA solution you choose *must* be designed for developers to be successful with.

Unfortunately, all too often, security tools (including SCA solutions) simply don't understand the developer as a user. Integrating into an IDE or creating a Jenkins plug-in does not automatically make a tool developer-friendly, nor does adding the term "DevSecOps" into your documentation. To be successful with developers, tools need to be on par with the developer experience (DX) other dev tools offer, adapt to the user flows of the tools they connect to, and have the product communicate in a dev-friendly manner.

Notable capabilities I mentioned that affect DX are integrating with source code systems, testing pull requests (PRs), opening fix PRs, and having a rich and easy to use CLI. That said, the best way to know if your dev team will love the solution you choose is to involve them in the selection process, and ask them explicitly if this is a tool they'll enjoy using. If the answer is no, you're at high risk of buying an expensive solution only to fail in getting it adopted.

Aim to Fix Issues, Not Just Find Them

SCA tools are typically considered testing tools, flagging security problems at relevant times. However, the real goal isn't to find vulnerabilities, but to fix them! In your plan for addressing this risk, be sure to go past the *find* and into the *fix*.

SCA tools differ greatly in how they support remediation. Some tools offer detailed advisories explaining the vulnerability, how the library entered your application, and what can you do to fix. Other tools go even further and automate the fix creation through fix PRs and patches. On the flip side, some tools don't go much further than listing the flaws.

You should consider what you see as the ideal remediation process, and then select the tools that help you tee up continuous fixing for success.

Verify the Coverage of the Vulnerability DB

Even a perfect SCA solution can't protect you against a vulnerability it doesn't know about. The depth of the vulnerability database used is critical to getting the protection you're after.

The most common pitfall in today's SCA landscape is relying exclusively on CVEs. As I've explained before, vulnerabilities in application libraries are simply not sufficiently covered by CVEs. Following that, you should inspect the caliber of the DB sources, such as the security expertise in the team that maintains it and the means of vulnerability discovery. Lastly, inspect the quality of the advisories themselves, to ensure they give you the information you'll need.

If you're especially security conscious, you'd do well to also seek out databases that are first to disclose vulnerabilities, and offer early notifications on such issues. History shows the number of manually researched vulnerabilities is typically low, but their average severity and the prevalence of the affected libraries is quite high.

Ensure Your Tool Understands Your Dependencies Well

Understanding dependencies is one of those problems that seems easy until you truly dig in. When using an SCA solution, you're relying on the tool's ability to properly detect the libraries you are using. If it misses a library, it can easily miss a vulnerability.

One frequent shortcut tools take is to inspect direct dependencies, but not expand the graph to include indirect ones. As I mentioned before, most of the libraries you use (and thus most of the vulnerabilities they'll hold) are indirect dependencies, pulled down by the packages you explicitly asked to use. It shouldn't be too hard to check if the tool you're inspecting goes deep into the dependency graph.

Beyond that, it's hard to assess the depth of understanding a solution holds. Choose a couple of apps you're very familiar with and see how well it expands their graph, especially in the application stacks that matter to you most.

Secure containers with a developer perspective.

Containers are firmly in the twilight zone between infrastructure and apps. Some look at them as the evolution of virtual machines (VMs), and thus handle securing their dependencies the same way. The team that handles patching VMs (including cloud VMs) is tasked with securing containers as well, as they're the ones who know how to patch servers.

Others, including myself, see containers more as the evolution of the app. While previously your build produced a `zip` or `war` file, it now produces containers—including the OS. These containers now behave *like the app*. They're owned and managed by the dev team, their source is stored in the git repository, and they're versioned and tested alongside the app. Most importantly, updating a dependency in a container requires a build to produce a new container image—which is firmly an application and developer action.

To secure container dependencies successfully, you must engage your dev team. A separate team managing VMs rarely has the technical permission to run a new build, has no context of how a container and app interact, and is almost always severely outnumbered. You need to accept that securing container dependencies is an expansion of the app's scope, not an adjustment to how you handle VMs, and choose your tools, practices, and team ownership accordingly.

Choose the Tool That Fits Tomorrow's Reality Too

The way we develop software is changing, increasingly getting faster and more continuous. Many companies are in the midst of a digital transformation journey, looking to increase their productivity and speed while maintaining the safety and reliability they're accustomed to.

Security practices and tools have not evolved as quickly, and many still rely on gates and pauses to allow audits or bring in the experts. Using the wrong security solution can get in the way of improving your development methodologies, which can prove very costly.

The tools you choose should clearly fit your needs today, but it's equally important to ensure they fit the way you *want* to develop software, and help you get there, not get in the way.

CHAPTER 7
Summary

Open source is amazing, and is here to stay, but it's also a complicated beast when it comes to ownership, trust, and security. Unlike purchased software, once you download an open source library you are on your own. There is no vendor to notify you about issues, no commitments that vulnerabilities will be found or fixed, and nobody to sue to if you got hurt.

If you're using open source (and you most certainly are), you need to take on the responsibility of keeping it secure. You must set up the tools and processes that will help you stay safe, and raise awareness of this risk throughout your organization. The 2017 Equifax breach serves as a very painful lesson of what could happen if you don't.

While open source security is a broad topic, the most critical part of it is dealing with known vulnerabilities in open source libraries. This book suggested a framework for dealing with this risk, split into four steps:

- *Find* the vulnerabilities in your dependencies
- *Fix* the found issues quickly and efficiently
- *Prevent* additions of new vulnerable libraries during dev
- *Respond* to newly disclosed vulnerabilities in libraries you already use

These logical steps should hold true regardless of the tools you choose for addressing this problem, but I definitely suggest you lev-

erage a Software Composition Analysis (SCA) solution to help you get them done. Each chapter points out various tool attributes to consider, tips on how to integrate them, and the trade-off you'll need to make.

As mentioned in the introduction, this book didn't cover the entirety of open source risk. I didn't discuss open source tools and frameworks, barely mentioned legal risks, and definitely didn't dive into potentially malicious libraries. The best practices in these areas are still being shaped, and I recommend participating in events like DevSecCon to be a part of this evolution.

As open source continues to mature, I'm certain the practices and technologies for dealing with its risks will improve as well. Open source is created by the community, and that same community should participate in keeping it secure. In the meantime, make sure you are a responsible member of both this ecosystem and your organization, accept ownership for controlling the risk from the libraries you use, and stay secure!

About the Author

Guy Podjarny is the cofounder and CEO of Snyk.io, focusing on securing open source code. He was previously CTO at Akamai and founder of Blaze.io. He also worked on the first web app firewall and security code analyzer, and was granted over 10 patents in security analysis techniques. Guy is a frequent conference speaker, author of *Responsive & Fast* and coauthor of *High Performance Images*, and the creator of Mobitest.