Depending on Dependencies

Securing your codebase on all levels

Oscar Persson

April 23, 2023

I/We certify that generative AI, incl. ChatGPT, has not been used to write this essay. Using generative AI without permission is considered academic misconduct.

1 Introduction

This section introduces the background for the essay, explaining the necessary background information, past events, and related material, in section 1.1. Section 1.2 explains the problem statement that this essay aims to answer.

1.1 Background

Following are related material for the purposes of this essay.

Package managers

A package manager, as described by Mozilla [12] is a piece of software which allows its user to manage their project dependencies. The manager does not only manage the direct dependencies, but also any sub-dependencies that might be necessary. This means that there are multiple levels to which a package manager may install packages on your computer per installation.

But where are these packages retrieved from? According to Mozilla [12], packages are retrieved from a central repository, called a *"Package Registry"*, such as npm, yarn, or PyPi, etc. A package registry is, as aforementioned, a centralised repository where any user may publish their own packages and the manager will sort any necessities to install and use the package in question.

It may be of note that an uploaded package to a registry, may always be updated by the author. These updates will then be pushed to all users, depending on their configuration regarding dependencies for their project, and an update of the dependencies will update the code. The code stored in a registry is thus never static, but volatile and may be subject to change at all times.

Sub-Dependencies

The nature of packages for development is a tree-based one. As Di Cosmo, et al. describes in the conference paper [11] "a **package** is a bundle that ships a (software) component, the data needed to configure it, and metadata which describe its attributes and expectations on the deployment environment", where the metadata also specifies any dependent **packages**. As a package may contain a package, this means that these packages may contain dependencies on their own, in an exponential, recursive, fashion. [5] created a dependency graph for the popular framework Ruby on Rails, which showcases this, 1.

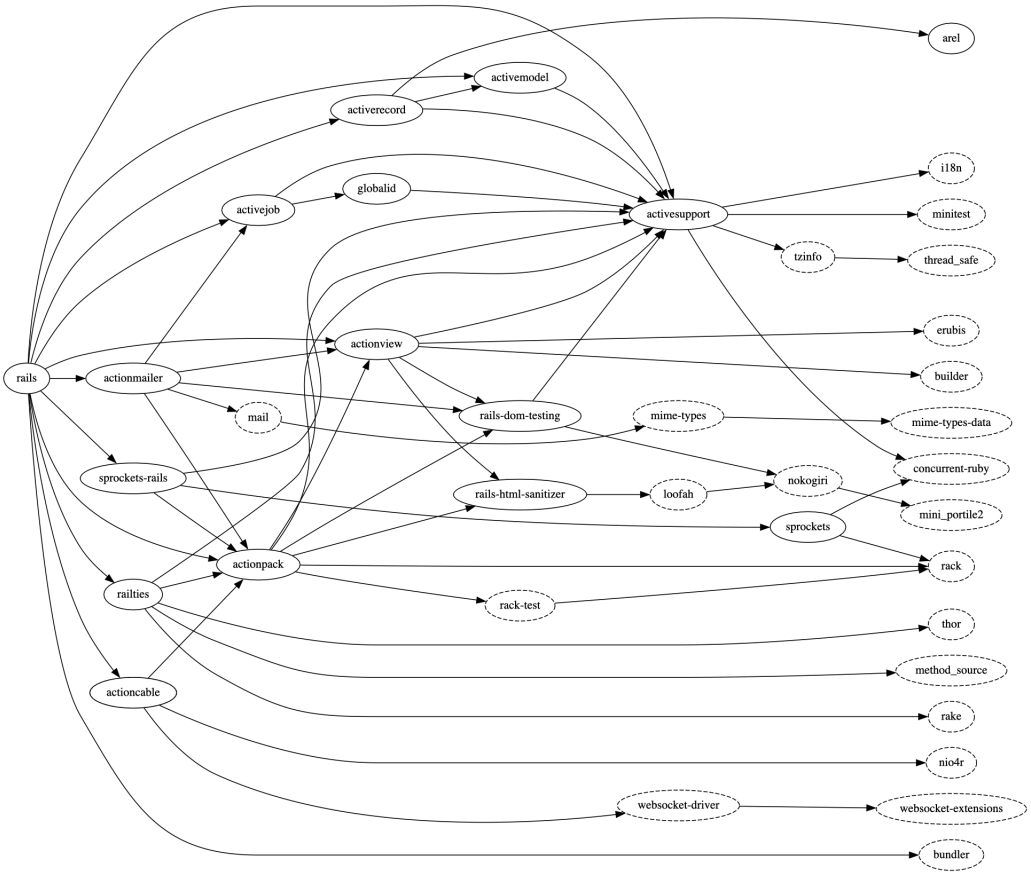

Figure 1: Ruby on Rails Dependency Graph

Leftpad

"How one programmer broke the internet by deleting a tiny piece of code" is an article written by Keith Collins [10]. In this article Collins outlines an event where one person managed to break web-development around the world through the package repository by removing his package from the npm registry.

From the, still removed, registry entry one can find the functionality and other information about Leftpad [2]. Leftpad is a function that pads a given character to the left of a provided string [2]. At the time of removal it was a tiny, rather unknown, package with only 11 stars on GitHub [10].

This begs the question, however. How did the deletion of this tiny piece of code, as Collins stated, "[...]break the internet"? The answer to that lies within the nesting of sub-dependencies and dependencies, mentioned in 1.1. At the time, there were a large amount of npm packages which relied on the functionality provided in Leftpad, however, the nested structure of web development meant that there also were a lot of packages relying on functionality from packages that were relying on Leftpad, and so on.

The reason Leftpad was removed was because of a dispute between the package registry npm, and Leftpad's author, Azer Koçulu. While further detail about the dispute will not be discussed, as it is outside of the scope of this essay, the dispute was of copyright nature, where npm decided to side with a larger corporation instead

of Koçulu. In retaliation to this, Koçulu removed Leftpad from the registry [10]. The implications of this is that each project is wholly dependent on the authors of the dependencies that they use, just like in the case of Leftpad.

Log4j

Log4j is a Java-based, open-source, logging framework, which is part of the Apache Logging Services [6]. Log4j, being a program module, was used in many applications as a software library; it was used as a dependency for many applications. Just like many other libraries out there, it alleviated the work-load for any developer using it, in this case with regards to logging. The module itself is described as being used to log and keep track of what is happening in a specific piece of software, application or online service, as described by Steven Perry [13] and the Apache web-page [3]. The information saved through such an application is often used in order to keep track of problems that occur during run-time.

The problem stemming from Log4j as Kaushik, et al. [9] states was named **Log4Shell**. Kaushik describes Log4Shell as a vulnerability where any system using Log4j is vulnerable to a remote code execution attack, where specific inputs to the program may allow a hostile user to take control over a system, or server. The issue affected commercial systems, smart phones, and other software, such as video games. Minecraft being a notable affected video game.

As is the nature of dependencies, Log4j being vulnerable to this exploit meant that any program which incorporated Log4j for logging purposes was also affected. Just like with the other dependency issues discussed in previous sections, [6] describes that is a commonly used component for many applications or other infrastructure that developers may have used during the development process, which means that the vulnerability is "hereditary".

Dependency Vulnerabilities

Having talked about specific vulnerabilities, which affected the world of web-development, one may wonder to which extent these vulnerabilities appear. GitHub, in 2018, celebrated 100 million open source repositories, while also showing an increasing trend of uploads of open source repositories, which still goes on until this day [15]. These open source applications, frameworks, or libraries do not just exist on their own. They are widely used within different software applications, one of the most prominent ones being React. One of the world's most popular web-development frameworks, which is an open-source repository.

Snyk, a developer security platform, estimates in 2022 that the average count of dependencies within a project is 344 total dependencies [4]. Snyk does state, however, that the amount of dependencies may differ between projects using different languages; JavaScript using the most and Python using the fewest, of the languages displayed. They have also counted the average amount of vulnerabilities per project. The average amount of vulnerabilities within a project is 49, as of 2022. 40% of these vulnerabilities, ie. 18-20, are found in transitive dependencies, meaning that they are found within the sub-dependencies within the project. Additionally, they state that "70% – 90% of modern software applications contain open source software.".

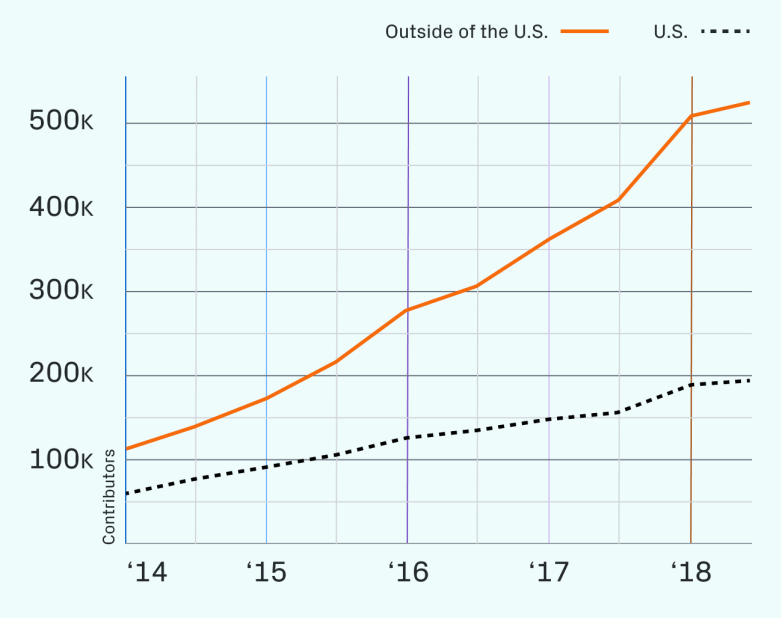

Figure 2: Trend graph of repository uploads to GitHub, 2018

1.2 Problem Statement

Having discussed a set of specific vulnerabilities which affected many of the users which were using the libraries, such as Log4Shell 1.1 and Leftpad 1.1, while also discussing the statistics of dependencies as well as vulnerabilities within any general, average, project, one may ask themselves how to keep their project safe and well protected. While the obvious solution to this problem would be to stop using dependencies, the practical answer is rarely that simple. As Snyk stated [4], the average project uses 344 open source library dependencies. The world of development of today is based on building upon what already exists, in order to alleviate the process. It was estimated by GitHub that a typical project consists of a majority, 60%-90%, of open source code or third-party code. This, however, does beg the question, *what can a developer, today, do to keep themselves safe from dependency vulnerabilities.* How do can a developer secure their code-base from its dependencies, and how much does it matter?

2 Analysis

In this section solutions for both preventing and detecting possible vulnerabilities within a project. The solutions which are brought up in this section will be in relation to the vulnerabilities that were introduced within 1.1.

2.1 Dependency Management

J. Williams and A. Dabirsiaghi, two authors for OWASP (Open Worldwide Application Security Project) wrote a paper about insecure libraries for Contrast Security [16]. In the paper they discuss a method called "Dependency Management", a process where a development team may identify which libraries their project is dependent on, ie. all dependencies and sub-dependencies, through recursive discovery. Through

this process development teams may keep their libraries up to date, ensure that licenses are appropriately upheld, and if vulnerabilities are found, they can quickly be quarantined or addressed. This (semi-) manual process is a tedious one, as project sizes grow, and more libraries are added to the project, as the dependency trees scale exponentially it becomes an exponentially larger amount of work. J. Williams and A. Dabirsiaghi [16] do however claim that the security benefits of this method have "yet to materialize", not for a lack of efficiency from the methods part, however, but for the lack of usage this methods shows in industry, most companies opting to handle libraries on an "ad hoc basis".

While certain vulnerabilities that stem from being dependent on third-party software, such as the Leftpad removal outlined in 1.1, other vulnerabilities stemming from the code in each of the libraries may be discovered. Most of the vulnerabilities within the code are not necessarily obvious, either. J. Williams and A. Dabirsiaghi say that libraries often vary in size from 10,000 to 200,000 lines of code, with a custom java application containing 5 to 10 security vulnerabilities per 10,000 lines of code. Manually, or even semi-manually, checking this amount of data will prove to be a very time consuming process.

While being described as a tedious and time-consuming process. In theory, having full control of the supply chain would yield significant security benefits, but because of the aforementioned downsides, it is rarely done.

2.2 Automated Dependency Mapping Techniques

In order to alleviate the labour done in 2.1, one may fully automatise the process of finding vulnerabilities within dependencies. Dependency mapping is a technique, previously mentioned in 2.1, which allows the developer to keep track of an overview of the entire application behaviour, however, mentioned in regards to manual execution. There are many tools one can implement to run this process automatically, each using one of several strategies in order to build this dependency view. Trend Micro, a Japanese cyber-security software company, outlines several of these methods in [1] some of which will be outlined quickly, following.

Automated ADDM , Application Discovery and Dependency Mapping, is a clear first step from manually mapping the dependencies. Most *automated ADDMs* are custom-build applications which maps and visualises the project dependencies. The purpose is to provide a comprehensive insight into the application, while noting possible abnormalities, and performing root-cause analysis in case of errors [1].

Agent on server , is a technique where several monitoring agents are placed on servers and infrastructure with the purpose of monitoring traffic in real time. The monitoring is then performed on both incoming and outgoing traffic. As the topology of the mapping changes, these changes can then be recognised and understood from this surveillance. The possible downside of this method is that the agents need to be placed strategically and manually in order to achieve maximum efficiency.[1].

While more techniques exists these were chosen for differing greatly from each other in many aspects, but mostly for how they differ in one being a more proactive method, Automated ADDM, whereas Agent on Server is a more passive method. As discussed

in and by [4], the average number of vulnerabilities in a given project is high, and more vulnerabilities may be discovered at all time. Methods which continuously monitor these have thus been chosen for discussion.

2.3 DevSecOps

To the field of DevOps, another layer has been added, security; DevSecOps. The focus of DevSecOps is that of DevOps, providing an efficient and effective way of building, deploying, and managing software. However, the addition of ensuring security and stability of the projects is added. [7] describes *dependency management* as a critical role in ensuring this security for a project, and thus is a critical role in DevSecOps. One of the common steps that DevSecOps implements are tools performing, amongst others, some of the techniques discussed in 2.2.

2.4 Challenges in Dependency Mapping

While providing a developer complete oversight of their project, automated dependency mapping techniques are not perfect. Faddom, a security company which specialises in dependency mapping tools [8], describes the largest downside for this technique to be an inability to provide "real-time, repeatable, and comprehensive mapping across environments". Different approaches are described to provide different results.

In comparison, however, Schulz, et al. states in their conference paper that contrary to the automated versions of dependency mapping "the manual expert-driven approach does not scale" [14].

3 Conclusion

To conclude, there are many challenges to take into consideration while keeping your project safe, and no perfect solution. Previous breaches such as Log4j and Leftpad, as discussed in 1.1 and 1.1, respectively, are only a few examples of what may happen, but also display the possibility that vulnerabilities are not only opened up or exploited by malicious actors. This, in and of itself, makes it very difficult to keep a system entirely secure while using third-party software or code. While there are many methods of keeping track of both dependencies and possible vulnerabilities in a project, none appear to be a perfect, one-fits-all. Depending on the scale of the project, ones resources, or future prospects, one may pick different solutions. There are no doubts, however, that one needs to keep track of the dependencies, as well as their dependencies, ad infinitum, in order to maintain security within a project.

3.1 Key Take-away

Always keep track of your dependencies. If you don't know what you put in, you can not guarantee what you get back.

4 Reflection

As discussed in 2.2, vulnerabilities are plenty within any written code, and third-party code is no exception. With the average project containing 344 dependencies, and the

average amount of vulnerabilities being 49, it may feel impossible to avoid. One may even assume that the most secure method would be to write all code from scratch. This is something that is not feasible for a lot of projects, however, and still does not guarantee a non-vulnerable project. The discovery of vulnerabilities in any code is always an "arms-race" which will probably never be solved. While the solutions brought up in this essay are common, none provide 100% security, and no method probably does. Some rely on exploits being found by someone already, or on the developer to find them themselves, automatically or manually, which is very difficult to ensure. While this essay does not necessarily only focus on dependency security as part of a DevSecOps pipeline, it is important to know how to apply it both inside and outside one.

4.1 State-of-the-art

As DevSecOps becomes more popular, security becoming an essential addition to DevOps, in today's development world, it is important to keep track of which aspects pose security risk to a project. While only being one part of DevSecOps, dependency security is of a large importance because of how dependent developers are on third-party dependencies in today's development sphere. Packages, libraries, and external tools are here to stay, thus it is important to be able to keep track of possible security breaches within them.

References

[1] Dependency Mapping for DevSecOps — trendmicro.com. https://www.trendmicro.com/en_us/devops/23/a/dependency-mapping-plus-tools.html. [Accessed 22-Apr-2023].

[2] left-pad — npmjs.com. https://www.npmjs.com/package/left-pad. [Accessed 20-Apr-2023].

[3] Log4j 2013; Apache Log4j - logging.apache.org. https://logging.apache.org/log4j/2.x/. [Accessed 20-Apr-2023].

[4] State of Open Source security 2022 — Snyk — snyk.io. https://snyk.io/reports/open-source-security/. [Accessed 20-Apr-2023].

[5] The package dependency graph 2013; Programmer's Compendium — destroyallsoftware.com. https://www.destroyallsoftware.com/compendium/the-package-dependency-graph?share_key=8047865d63625b8b. [Accessed 20-Apr-2023].

[6] Log4j vulnerability - what everyone needs to know — ncsc.gov.uk. https://www.ncsc.gov.uk/information/log4j-vulnerability-what-everyone-needs-to-know, 2021. [Accessed 20-Apr-2023].

[7] CODARIO. DevSecOps and Dependency Management - Codario — codario.io. https://codario.io/blogs/devsecops-and-dependency-management/. [Accessed 21-Apr-2023].

[8] Faddom. IT Mapping: Challenges, Approaches, and Why to Use ADM — faddom.com. https://faddom.com/it-mapping/#two. [Accessed 23-Apr-2023].

[9] Kaushik, K., Dass, A., and Dhankhar, A. An approach for exploiting and mitigating log4j using log4shell vulnerability. In *2022 3rd International Conference on Computation, Automation and Knowledge Management (ICCAKM)* (2022), pp. 1–6.

[10] Keith Collins. How one programmer broke the internet by deleting a tiny piece of code. https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code, accessed 2023-04-19.

[11] Kolter, J. Z., and Maloof, M. A. *Learning to detect and classify malicious executables in the wild*. Springer, 2010, pp. 487–506.

[12] Mozilla Developer Network. Understanding client-side tools: Package management. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Package_management, accessed 2023-04-19.

[13] Perry, J. *Log4J*. O'Reilly Media, 2009.

[14] Schulz, A., Kotson, M., Meiners, C., Meunier, T., O'Gwynn, D., Trepagnier, P., and Weller-Fahy, D. Active dependency mapping. In *Theory and Application of Reuse, Integration, and Data Science* (Cham, 2019), T. Bouabana-Tebibel, L. Bouzar-Benlabiod, and S. H. Rubin, Eds., Springer International Publishing, pp. 169–188.

[15] Warner, J. Thank you for 100 million repositories — The GitHub Blog — github.blog. https://github.blog/2018-11-08-100m-repos/. [Accessed 20-Apr-2023].

[16] Williams, J., and Dabirsiaghi, A. *The Unfortunate Reality of Insecure Libraries*. Contrast Security, 2014, pp. 0–26.