EDITOR **DIRK RIEHLE**
Friedrich Alexander-University of Erlangen Nürnberg;
dirk.riehle@fau.de

Managing the Open Source Dependency

Tomas Gustavsson, PrimeKey

Organizations use open source software in a majority of computer application programs. Here we describe some of the technical challenges and offer recommendations about how to manage open source software dependencies and avoid the most common pitfalls that might be encountered through decision-making, automated scanning, upgrading, and strategic contributions.

In earlier articles in this column, we learned that open source is used in most of the current software produced as well as proprietary license code.[1] Being used either as stand-alone components or subcomponents within software, it is an important part of the software development lifecycle. Managing these dependencies is critical for the long-term maintenance of a product. We previously read about managing dependencies from a licensing and compliance perspective,[2] and in this article, we will look at managing the dependency from a technical perspective.

Clearly, managing software dependencies is not specific to open source software and is always an important thing to consider. Open source dependencies bring several advantages over proprietary licensed dependencies, such as easy access, new versions direct from the development team, and unlimited usage. Regardless of the ease of use, there are some specifics that are important to manage in the long-term software lifecycle.

WHAT IS AN OPEN SOURCE DEPENDENCY?

When talking about an open source dependency, it is important to first define what that means. Although this sounds simple, different people will have different interpretations, and when digging a little deeper, the meaning is not as simple as it may seem.

What is different about an open source dependency compared with other dependencies? A frequent misconception about open source software is that when it is free, it does not come with support. Managing an open source dependency where there is a business relationship with a vendor, such as a support subscription, is not much

Digital Object Identifier 10.1109/MC.2019.2955869
Date of current version: 12 February 2020

FROM THE EDITOR

Welcome back! In our article series, taking the perspective of a software developer, we have warmed up to using open source in products. After learning how to select the right components for our needs in the last installment of this column, we will now look at how to manage the resulting dependency. Expert Tomas Gustavsson takes us through an analysis of various aspects of this dependency and how to manage it. Security looms large, as it often does, but next to assurances, he also looks to more actively engage with an open source component's development community, often called contributing to upstream. Viewing open source communities as suppliers to your product is a helpful perspective, of which we will learn more in future columns. As always, happy hacking! — *Dirk Riehle*

different from managing any other dependencies apart from possible exit costs (which may be a topic for another article). Therefore, we will focus on the open source dependency, which can best be characterized by using an open source licensed subcomponent without a contractual relationship with a vendor, without warranties, and without a service-level agreement.

DEPENDENCY DEPLOYMENT FORMS

An open source dependency can be used in different deployment forms:

1. It can be employed in an open source library (for example, Apache Commons Java libraries) when developing another application (proprietary or open source licensed). The open source library is then bundled inside the new application.
2. An open source component can be added as part of a larger solution where the component is used as is but bundled in the complete solution. One example is using the MariaDB open source database to run an application where the MariaDB database is bundled in a larger solution installation package.
3. An open source component can be implemented as part of a larger solution where the open source component is installed separately. An example is using the Nginx open source webserver for running a web application, where the web application requires installing Nginx separately as a prerequisite.

Managing dependencies in these three cases has some similarities; but in the third case, the dependency lies outside the developer's control and is controlled by the end user. In this instance, the responsibility for managing the lifecycle of the dependency lies with the user of the open source application and, thus, falls outside the scope of this article.

COMMON DEPENDENCY PITFALLS

There are several best practices to follow in dependency management to avoid pitfalls. Common pitfalls are as follows:

- *Security vulnerabilities*: Vulnerabilities in subcomponents, which most often occur when using outdated versions, may cause security problems and breaches in the delivered solution. This has been proved, in many occasions, for example, the Equifax breach.[4] "Using Components With Known Vulnerabilities" is now on the Open Web Application Security Project (OWASP) top 10 list of the most critical web application security risks.[5]
- *Delayed and costly upgrade cycles*: When dependencies have not been upgraded for an extended period, sometimes for years, the process can become very resource consuming when an upgrade is finally required. This can be avoided if the dependency is maintained continuously or at an adequate frequency.
- *Interoperability problems*: Complex solutions often consist of many different parts that interact. When integrating the solution, different versions of the same open source dependency can cause interoperability problems, which then have to be solved during deployment instead of earlier.
- *Breaking functionality*: Often, open source components are used as building blocks of underlying abstraction layers, such as network communication and encryption. The principles and algorithms of the Internet infrastructure can, however, change, and if dependencies are not upgraded, critical functionality may stop working. A simple example includes the deprecation of insecure versions of the Transport Layer Security protocol.

GOOD DEPENDENCY MAINTENANCE PRACTICES

Because of the open nature of open source components, such dependencies have a tendency to appear in code as a result of the pressure on developers to get the job done and deliver at a fast pace. There is not always a rigorously vetted decision behind every dependency that gets included. Maintaining dependencies, whether open source or proprietary licensed, requires resources and adds to the technical debt of software, which is why managing this is an important part of the development process. By following a few best practices that can easily be integrated into a modern development

process, most dependency problems can be managed without difficulty.

A summary of practical best practices is

- establish a forum for conscious decisions on open source dependencies
- maintain a dependency list
- scan for security issues
- verify the integrity of downloads
- upgrade continuously
- contribute upstream
- support open source projects
- document the process.

Depending on your organization's development maturity, size, and processes, different levels of automation can be applied to all of these practices. If there are three words that would describe these practices in summary, they would be *consciousness, security,* and *automation*.

Conscious decisions

There are huge savings to be gained from using open source components, but there are also maintenance costs. Some dependencies are brought in for short-term gains, but in the long term, they cause great costs. Careful and deliberate decisions are key to success.

Limit your open source dependencies to deliberate decisions. Just because there is an open source implementation available does not mean you have to use it. If you use a single function from a library, the maintenance of this dependency may be greater than implementing this specific method yourself. On the other hand, if multiple methods from this library are used, the burden of maintaining your own implementations is often much higher than using the library. There is a balance, as always.

One important factor to consider when taking on a dependency is whether the project is actively developed. Many popular open source projects are very active and will release new versions when bugs are discovered, whereas other open source projects are created without either active users or active developers or have been abandoned. It may be wise to avoid nonactive projects because maintenance costs should be expected in practice, as if it was your own code.

Three good processes for deciding when to add a dependency are the following:

- Establish a group that decides on common dependencies across your products and projects, with members from different product teams who are aware of the organization's strategy for open source dependencies.
- Develop criteria for evaluating open source projects to aid in the decision to adopt.
- Automate detection of open source dependencies.

Maintain a dependency list

Maintaining a list of dependencies has multiple benefits. Some benefits have been described in earlier articles[1,2] related to licensing and compliance. The dependency list also comes in handy when maintaining the technical dependency. Maintaining lists makes it easier for developers to know the function of a certain dependency, something that is not always obvious in a large product. A well-structured list should contain pointers and URLs to indicate where the latest versions of the software can be obtained. This enables developers to get new versions of dependencies from the official source, avoids unverified code (see later), and enables quality assurance to identify the functions to test after dependencies have been upgraded.

When maintaining the technical dependencies, we use some of the same results as from earlier articles, like the bill of materials over open source components. Reflecting on the dependency maintenance should be a part of maintaining the bill of materials. With multiple solutions in an organization, it is also a good idea to look at the bills of material for all of the various components that make up a complete solution to remediate interoperability concerns.

Naturally, the work involved in maintaining dependency lists varies depending on the size of the organization and its products. Smaller product teams may be able to start manually, whereas larger product teams will need a high level of automation to be able to even determine the dependencies.[3]

Scan for security issues

With using subcomponents with known vulnerabilities that are on the OWASP top 10 list of most critical web application security risks, it is a signal that security issues need to be handled. Security issues arise in almost all software, be it proprietary or open source licensed. Larger software components (both open source and proprietary) file what is called *common vulnerabilities and exposures* when security issues are discovered. There are tools that can help you automatically scan for known vulnerabilities in subcomponents you depend on; when these are integrated in the development process, managing this risk becomes much easier.

Verify integrity

When software is retrieved from the Internet, it is important to verify its integrity. Malicious versions of software are sometimes placed on download sites, compromising systems where these versions are installed. In addition, download servers are sometimes breached so that the correct version of software is replaced by a malicious one. This happens to both open source software and non–open source software; because open source dependencies in our definition are always downloaded from the Internet, without specific contact with the vendor, it is very important to verify integrity.

There are two methods that are the most common to verify the integrity of open source dependencies:

- verifying a hash (checksum) of the downloaded software with a hash obtained from a trusted (or at least another) source
- verifying a digital signature of the downloaded software where

the verification key has been obtained from a trusted (or at least another) source.

The process can be either manual or automated, depending on how dependencies are managed and retrieved in your organization. The most important issue is that you know if and how integrity verification is performed.

Upgrade continuously

To avoid many of the common pitfalls, dependencies should be upgraded regularly. This is a maintenance burden in the short term, but it will save large costs over time. If development processes are designed to consider dependency maintenance, the costs can be kept down, and many of the costly pitfall issues avoided. To be able to upgrade continuously without spending an unreasonable amount of time, there are some basic development hygiene factors that you should have in place. One is the use of primarily automated testing that verifies that upgrades work as expected. Today, this practice is commonplace and implemented as part of a continuous integration pipeline, where automated tests are executed as soon as changes are made to the source code of a product. Another good practice is to have a process step to verify and upgrade external dependencies for each larger feature release of the product.

Recently, tools have become available to address the complexity of maintaining dependencies. These tools can assist in automating everything from detecting reported security vulnerabilities early to automating the upgrade of dependencies.

Contribute upstream

One thing that is different when using open source dependencies is that every user has the ability to contribute development upstream. *Upstream* denotes the open source project that you rely on in your solution. You do not have to contribute to the open source dependency in code, but these contributions can occur in many ways using different work practices,[6] such as

- bug reports and bug fix code
- feature requests and feature code
- support for other users and participation in forums
- documentation and translations.

> The process of interacting with open source projects and managing open source dependencies should be documented.

Making this investment in strategic open source projects may have great benefits for the organization, and the reasons to do it can be different among organizations.[7] Apart from strategic benefits, there are many practical maintenance benefits:

- not having to maintain your own code changes separately from the main project
- making upgrades easier, as changes are already part of the main project
- receiving community support as helping others makes them willing to help you
- limiting dependency on specific persons in your organization
- the potential to receive improvements and additional testing (for example, on a different scale) on contributed features
- knowledge sharing within your organization.

In the long term, contributing upstream can save large amounts of time for your organization even if there is a short-term perceived cost.

In practice, contributing upstream involves the tasks of locating the project website and finding out from that site which communication mechanisms are available. In most cases, there are email lists and web forums and, commonly, an issue tracker for bug reports and feature requests. For projects hosted on popular open source collaboration platforms, such as GitHub (https://github.com/), there are standard tools available that are the same for all projects, making it easy to incorporate new projects once you are familiar with the platform.

Support open source projects

Depending on the criticality of dependencies, you will want to limit the risk that a dependency is abandoned. Some open source projects are run by commercial organizations, whereas others are run by individual volunteers. Regardless of how the open source project is run, if you depend on a subcomponent for the medium or long term, it needs to be sustainable. Sustainability should be one of the criteria used when selecting open source subcomponents. There are many ways to improve the sustainability of open source projects, ensuring that there are developers able and willing to work on the project. If the project has commercial backing, purchasing a support contract is one way. Financial contributions can also be given if the project is managed by a foundation. If financial contributions are not possible, other contributions should be considered; one example is sponsoring with infrastructure (such as test servers). Remember, subcomponents that are not sustainable may end up being maintained by yourself in the long term. It is more cost-effective to do it right from the start than to simply assume that open source is without cost.

Document the process

When working in an organization, the process of interacting with open source projects and managing open source dependencies should be documented and become part of the development process. If not well documented, the

dependency management may become dependent on specific individuals, and they might leave the organization.

The described practices can be seen as a base level on your way to mastering open source dependencies. There are many additional practices, which we may count as advanced best practices, that you will discover as your processes evolve. Some lessons will always be learned the hard way, but applying some best practices will help you from getting into trouble too early. If these are done with consideration, there are fantastic benefits of open source software dependencies that you can reap on the way to open source engagement proficiency.[3]

REFERENCES

1. M. Ballhausen, "Free and open source software licenses explained," Bird & Bird LLP, Hamburg, Germany, June 17, 2019. [Online]. https://dirkriehle.com/2019/06/17/free-and-open-source-software-licenses-explained-miriam-ballhausen-ieee-computer-column/
2. H. Schöttle, "Open source license compliance—Why and how?" Osborne Clarke, Munich, Germany, Aug. 8, 2019. [Online]. https://dirkriehle.com/2019/08/08/open-source-license-compliance-why-and-how-hendrik-schoettle-ieee-computer-column/
3. J. McAffer, "Getting started with open source governance," GitHub, Oct. 8, 2019. [Online]. https://dirkriehle.com/2019/10/08/getting-started-with-open-source-governance-jeff-mcaffer-ieee-computer-column/
4. M. Varmazis, "Equifax felled by a months-old Apache Struts vulnerability," Sophos, Abingdon, UK, Sept. 14, 2017. [Online]. https://nakedsecurity.sophos.com/2017/09/14/equifax-felled-by-a-months-old-apache-struts-vulnerability/
5. OWASP, "OWASP Top 10—2017: The ten most critical web application security risks." Accessed on: Jan. 8, 2020. [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf
6. S. Butler et al., "On company contributions to community open source software projects," *IEEE Trans. Softw. Eng.*, to be published. doi: 10.1109/TSE.2019.2919305.
7. S. Butler et al., "Maintaining interoperability in open source software: A case study of the Apache PDFBox project," *J. Syst. Softw.*, vol. 159, p. 110,452, Jan. 2020. doi: https://doi.org/10.1016/j.jss.2019.110452.

TOMAS GUSTAVSSON is the chief technology officer at PrimeKey, Solna, Sweden. Contact him at tomas.gustavsson@primekey.com.

Digital Object Identifier 10.1109/MC.2020.2967208