On the Impact of Outdated and Vulnerable Javascript Packages in Docker Images

Ahmed Zerouali[1], Valerio Cosentino[2], Tom Mens[1], Gregorio Robles[3] and Jesus M. Gonzalez-Barahona[3]
UMONS - Belgium[1], Bitergia - Spain[2] and URJS - Spain[3]
ahmed.zerouali@umons.ac.be, valcos@bitergia.com, tom.mens@umons.ac.be, grex@gsyc.es, jgb@gsyc.es

***Abstract*—Containerized applications, and in particular Docker images, are becoming a common solution in cloud environments to meet ever-increasing demands in terms of portability, reliability and fast deployment. A Docker image includes all environmental dependencies required to run it, such as specific versions of system and third-party packages. Leveraging on its modularity, an image can be easily embedded in other images, thus simplifying the way of sharing dependencies and building new software. However, the dependencies included in an image may be out of date due to backward compatibility requirements, endangering the environments where the image has been deployed with known vulnerabilities. While previous research efforts have focused on studying the impact of bugs and vulnerabilities of system packages within Docker images, no attention has been given to third-party packages. This paper empirically studies the impact of npm JavaScript package vulnerabilities in Docker images. We based our analysis on 961 images from three official repositories that use *Node.js*, and 1,099 security reports of packages available on npm, the most popular JavaScript package manager. Our results reveal that the presence of outdated npm packages in Docker images increases the risk of potential security vulnerabilities, suggesting that Docker maintainers should keep their installed JavaScript packages up to date.**

***Index Terms*—Docker, npm, JavaScript, empirical analysis, security vulnerability, outdated software**

I. Introduction

In recent years, the way of developing software has significantly changed to cope with the continuous changes in the product development cycle and the need to accelerate the time to market. In this evolving scenario, containerized applications, and in particular Docker images, play a key role, improving portability, reliability and deployment [1].

A Docker image is a lightweight, stand-alone executable piece of software, that includes an entire runtime environment [2]. Thus, an image contains an application, plus all its dependencies, such as system and third-party packages, libraries, binaries, and configuration files. The content of an image is declared using a Dockerfile[1] and consists of a list of commands grouped into layers, each one identified by a unique hash signature. An image can be built upon another image, automatically inheriting its layers, and consequently its dependencies. By containerizing the application and its dependencies, differences in operating system distributions and underlying infrastructure are abstracted away. This promotes modularity and eases building new software. Docker images are freely available on registries such as Docker Hub, one of the largest registries providing a common place to build, update and share images among users. In Docker Hub, images are distributed using repositories, that allow users to develop and maintain several versions of different images (e.g., for different architectures), where each image can be tagged with different names (e.g., debian:stretch and debian:buster) to ease searching and use.

Official repositories contain public and certified images, supposedly secured and well maintained, from known organizations (e.g., ElasticSearch and MySQL). They serve as starting point to compose new images. *Community* repositories can be created by any user or organization. Linux-based Docker Hub images usually include system packages that correspond to the used Linux distribution (e.g., Alpine or Debian), plus a collection of third-party packages that come from popular package managers like *npm*, *PyPI* or *CRAN* (the default package managers for JavaScript, Python and R). An image often contains outdated packages to comply with backward compatibility requirements of the embedded application. In many cases having outdated packages could be legitimate and unavoidable. Nevertheless, such packages may be affected by old bugs and security vulnerabilities fixed in more recent package versions, thus they expose the production environment where the image is deployed to potential risks.

Previous work [3], [4], [5] has focused on measuring the vulnerabilities of system packages within Docker images, overlooking third-party packages. Given the increasing use of the latter, it is important to study the impact of using outdated and vulnerable releases of such third-party packages. This paper focus on the increasing use of JavaScript packages in Docker images [6]. We present an empirical analysis based on the notion of *technical lag* [7], which measures the difference (e.g., in time or in version) between the version of the package used in production and its most recent installable version.

Our objective is to assess the status of JavaScript packages within official Docker images. We leverage on the information available in npm, the default package manager for the JavaScript runtime environment Node.js, and in Snyk.io, a continuous security monitoring service which has the biggest database of npm package vulnerabilities. We measure technical lag and security vulnerabilities for images that are based on the Debian or Alpine operating system and come from three popular Docker Hub repositories: *node, ghost* and *mongo-express*. More specifically, we focus on two research questions:

[1]https://docs.docker.com/engine/reference/builder/

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

SANER 2019, Hangzhou, China
ERA Track

RQ_1: *How outdated are npm packages used in official Docker images?* We analyze the technical lag induced by using npm packages within official Docker images by comparing these images at the date of their last update and at the date of data extraction.

RQ_2: *How vulnerable are npm packages in official Docker images?* We use a dataset of npm vulnerability reports to identify which and how many vulnerable packages are present in images.

II. Related Work

Gummaraju et al. [4] performed an analysis on Docker Hub images to understand how vulnerable they are to security threats. One of their main findings is that over 30% of the official images contain *high priority* security vulnerabilities. Shu et al. [3] did a larger-scale analysis with 356,218 images coming from both community and official repositories. They found that both types of images contain more than 180 vulnerabilities on average; many images have not been updated for hundreds of days; and vulnerabilities commonly propagate from parent images to child images. Our study is different in that we focus on how outdated packages are.

Kula et al. [8] conducted an empirical study on library migration that covers over 4,600 GitHub software projects and 2,700 library dependencies, in order to investigate to which extent developers update their library dependencies. They found that 81.5% of the studied systems still keep their outdated dependencies. Moreover, based on a developer survey they found that 69% of the interviewees claimed to be unaware of their vulnerable dependencies. Zerouali et al. [9] empirically analyzed the package update practices and technical lag for the npm distribution of JavaScript packages. Their results showed a high proportion of outdated package dependencies, indicating a reluctance to update dependencies to avoid backward incompatible changes. Cox et al. [10] analyzed 75 software systems and introduced different metrics to quantify their use of recent versions of dependencies. They found that systems using outdated dependencies were four times more likely to have security issues than up-to-date systems. Zapata et al. [11] found that such library-level dependency analysis tend to overestimate the risk: through a manual inspection of 60 client projects from three cases of high severity vulnerabilities, they found that up to 73.3% of outdated clients were actually safe from the threat.

Decan et al. [12] performed an empirical study of nearly 400 security reports over a 6-year period in the npm dependency network containing over 610K JavaScript packages. Taking into account the severity of vulnerabilities, they analyzed how and when these vulnerabilities are discovered and fixed, and to which extent they affect other packages in the packaging ecosystem in presence of dependency constraints. Lauinger et al. [13] studied the client-side use of JavaScript libraries. They found that "the time lag behind the newest release of a library is measured in the order of years" and that this is a major source of known vulnerabilities in websites using these libraries. They also observed that "libraries included transitively [...] are more likely to be vulnerable".

This paper differs from all previous work in the fact that we analyze the use of outdated JavaScript packages within Docker images.

III. Method

This section describes the steps followed to obtain the dataset used in our study: (1) identifying candidate Docker images, (2) extracting npm package data, (3) collecting security vulnerabilities, and (4) computing technical lag. For step (1), we relied on the official *node* image[2], which contains *Node.js*. We focus on *node* images that are based on the Debian or Alpine base images, since the use of the corresponding Linux distributions is widespread in Docker[3]. In step (2), we pulled and ran the candidate images locally and identify the installed npm package versions. In steps (3) and (4), based on the npm packages found, we computed their technical lag and identify the security vulnerabilities that affect their versions.

A. Identifying Candidate Images

Our empirical analysis relies on the official *node* image as the base to retrieve the Docker images that include npm packages dependencies. As a consequence of Docker's layering mechanism, if a Docker image is based on the *node* image, then all the layers of this *node* image will be included in it.

We extracted all available images (i.e., latest and tagged ones) from the 124 official repositories through the Docker API. We remotely inspected them using skopeo, a tool to gather information of remote images registries[4]. We found that 961 out of 8,891 unique official images make use of *node* and are based on the Debian or Alpine operating system. All these images are coming from only three repositories: *node*, *ghost*[5] and *mongo-express*[6]. Table I shows the number of images considered for this analysis per repository and operating system.

TABLE I
NUMBER OF ANALYZED IMAGES GROUPED BY REPOSITORY AND OPERATING SYSTEM.

OS	**Debian**			**Alpine**	
Repository	node	ghost	mongo-express	node	ghost
Images	785	40	17	84	35

B. Extracting npm Package Data

When installing a JavaScript package from npm, a *package.json*[7] file is created locally on the machine, containing information such as the package name, version, description, etc. To inspect which packages are installed in each image, we pull and run the image locally, and then locate all *package.json*

[2]https://hub.docker.com/_/node/
[3]https://www.ctl.io/developers/blog/post/docker-hub-top-10/
[4]https://github.com/containers/skopeo
[5]Ghost is a free and open source blogging platform written in JavaScript
[6]Mongo-express is Web-based MongoDB interface, written with express
[7]https://docs.npmjs.com/files/package.json

files in it. Next, we identify package names and versions found in the files. To determine if a package version is outdated, we rely on the dataset extracted from *libraries.io* on March 13th 2018 [14]. This dataset contains metadata from the manifest of each package, based on the list of packages provided by the official registry of npm.

C. Collecting Security Vulnerabilities

To identify security vulnerabilities that affect installed npm package versions, we manually gathered from *Snyk.io* all 1,099 security reports that were published before May 3rd 2018[8]. Each security report contains vulnerability information about the affected package, the range of affected releases, its severity, its type (i.e., how vulnerable it is), the date of its disclosure, and the date when it was published in the database. For each security report, we identified the name and list of packages versions affected using the free dataset of libraries.io [14].

D. Quantifying Outdated Package Releases

The notion of technical lag is used to quantify how outdated a package release is compared to the latest available release [7]. Based on previous work [9], we calculate and analyze the technical lag of npm packages in Docker images in terms of versions and time.

Suppose that an npm package *P* has the following series of successive version numbers, (0.0.1, 1.0.0, 1.1.0, 1.1.1, 2.0.0), and suppose that an image *C* uses version 1.0.0 of *P*.

- The *time lag* is the difference in time between the release dates of the currently used version (1.0.0) and the highest available version (2.0.0).
- The *version lag* is defined as the number of updates between the used version 1.0.0 and the highest available version 2.0.0, grouped by their update type (i.e., patch, minor or major). From 1.0.0 to 1.1.0 a minor update happened (only the middle number changed). From 1.1.0 to 1.1.1 a patch update happened (only the right number changed). From 1.1.1 to 2.0.0 a major update happened (since the left number changed). Hence, for this example, the *version lag* is 1 major + 1 minor + 1 patch.

IV. RQ_1: How Outdated Are npm Packages Used in Official Docker Images?

To answer the first research question we analyse the technical lag of npm packages in Docker images based on (1) the date of the last available update and (2) the time when we performed the analysis (March 13th 2018). This will provide insights about the state of npm packages while their images were still being maintained, and their state at a later time, when they possibly accumulated more technical lag.

A. Technical Lag at the Date of the Last Update

Packages that are up-to-date have no technical lag. Therefore, we first started by exploring how many packages are outdated. We found that the majority of the used npm packages are up to date. The median proportion of up-to-date packages is 64% for Debian images and 57% for Alpine images. These proportions are still fairly low and show a high potential of problems due to outdated package releases.

Figure 1 shows the time lag distribution of all outdated packages, grouped by year and operating system (Alpine and Debian). We found a statistically significant difference ($p < 0.001$) when comparing the distributions of Alpine and Debian for each year using the Mann-Whitney U test. However, the difference between the time lag of outdated packages was small, according to the effect size ($|d| = 0.16$) computed with Cliff's delta, a non-parametric measure quantifying the difference between two groups of observations. We also notice that the time lag is increasing over time, which could be a consequence of old images that have been re-uploaded to Docker Hub without having updated their contained npm packages.

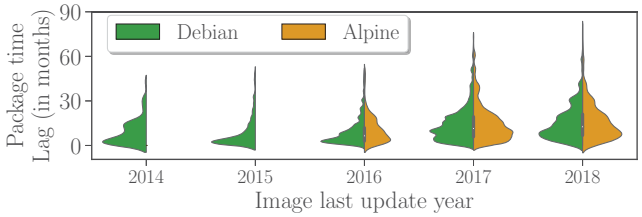

Fig. 1. Yearly distribution of the time lag (calculated at the date of the image's last update) for all outdated packages, grouped by operating system.

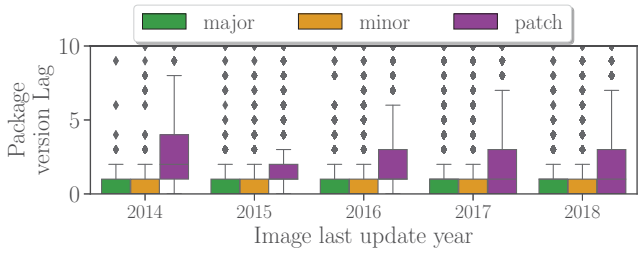

Fig. 2. Yearly distribution of version lag (calculated at the date of image's last update) for all outdated packages in images.

Figure 2 shows the version lag distribution of all outdated packages, grouped by the image's last update year. The version lag seems to remain quite stable over time. When considering all packages in the images, we found that the median version lag is 0 major + 0 minor + 1 patch. Thus, at the time of the image's last update, outdated npm packages were mainly missing patch updates, which is expected since patch updates are released frequently [9].

B. Technical Lag at the Date of the Analysis

We computed the technical lag of used npm packages on March 13th 2018. Compared to the proportions at the date of the last update, we found that up-to-date packages decreased to 41% and 34% for Debian and Alpine images, respectively. Figure 3 shows the distribution of the version lag found at

[8]https://snyk.io/vuln?type=npm

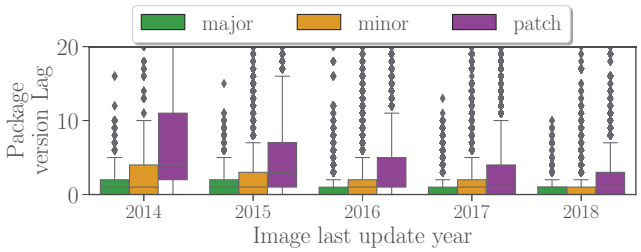

Fig. 3. Yearly distribution of version lag (calculated at the date of March 13th 2018) for all outdated packages in images.

the date of the analysis. The lag appears to decrease over time with different rates for patch (clearly visible), minor or major updates. Packages of images last updated in 2014 have a median version lag of 1 major + 1 minor + 4 patch. Images last updated in 2018 have a lower median version lag of 0 major + 1 minor + 1 patch.

Findings: Docker deployers that use old Node.Js images might be missing updates, including one major update.

V. RQ_2: How Vulnerable Are npm Packages in Official Docker Images?

A security vulnerability is a fault that could be exploited to breach the system. They are the most important bugs and require more experienced developers to fix them [15]. With this research question, we evaluate to which extent Docker images have vulnerabilities caused by outdated npm packages.

After identifying which installed npm packages are affected by the vulnerabilities reported on Snyk.io, we found that from 1,099 known vulnerabilities, only 74 are affecting the npm packages in Docker images. This number of vulnerabilities are affecting 3.7% (i.e., 52) of all (i.e., 1,412) unique installed packages. However, all images are affected by these vulnerable packages. We found that most of the vulnerabilities (54%) were disclosed between 2017 and 2018. 40% of the vulnerability reports have medium severity, 35% have high severity and 25% have low severity.

We identified the type of affecting vulnerabilities so that maintainers can assess the severity of the vulnerabilities depending on their proper use of the containers. We found 27 types of vulnerabilities in total, affecting 52 unique packages, of which the *ReDoS* (Regular Expression Denial of Service) vulnerability type was responsible for 55.7%. Table II shows the frequency for the top 5 vulnerability types affecting npm packages in Docker images.

There is a mean of 16.6 vulnerabilities per container and a median value of 10. Figure 4 shows a scatterplot of the last update dates of images against the number of vulnerabilities found in them. It shows that recently updated images have less vulnerabilities than older ones. However, for *low* severity vulnerabilities the evolution is different. We computed Pearson's R and Spearman's ρ correlation between the number of vulnerabilities and the date of the last update. We found that the number of *low* severity vulnerabilities has a weak positive

TABLE II
The top 5 vulnerability types found for npm packages in Docker containers.

Vulnerability type	% affected containers	# unique affected packages
ReDoS	100	29
Uninitialized Memory Exposure	100	7
Prototype Pollution	100	4
Access Restriction Bypass	100	1
Prototype Override Protection Bypass	79	1

correlation with the last update of an image ($R = 0.16$, $\rho = 0.25$). *medium* severity vulnerabilities have strong negative correlation ($R = -0.74$, $\rho = -0.8$) while *high* severity vulnerabilities have a weak to moderate negative correlation ($R = -0.28$, $\rho = -0.58$). Ignoring the unimportant (i.e., low severity) vulnerabilities, we conclude from this that the number of vulnerabilities per image is higher for older images than for more recent ones.

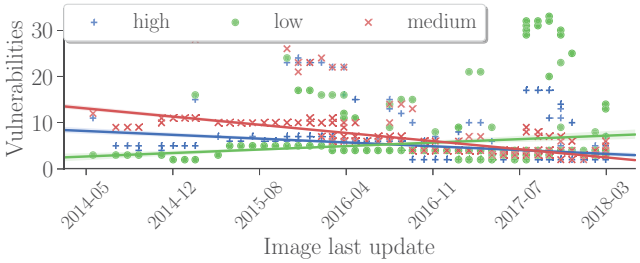

Fig. 4. Image last update and number of vulnerabilities found in it.

Findings: All official *node*-based images have vulnerable npm packages, with an average of 16 security vulnerabilities per image. Older images are more likely to have more vulnerabilities.

VI. Discussion

Ideally, Docker images should depend on the the most recent available version of their packages in order to benefit from the latest functionality, security updates and bug fixes. However, it may be possible that maintainers prefer to give higher priority to backward compatibility and stability instead of upgrading their software, since the latter would require a considerable effort. To assess this, we studied the presence of technical lag and vulnerabilities of third-party npm packages used in Docker images.

When we calculated the technical lag at the last update date of each image, we found that most of the packages are up-to-date, and that technical lag is low. Conversely, at the date of our data extraction), we detected an important shift, as technical lag goes up with 1 major version. This lag may induce the presence of security vulnerabilities.

In this paper, we only analyzed *official* Docker images, which are supposed to be more secure and well maintained

since they are the base images of other official and community images. Thus, the number of vulnerabilities found in the analyzed images is an underestimation of the real number of vulnerabilities that might be found in community images that depend on them (without updating) and where more dependencies, activity and development are expected.

Most research and tools around Docker (e.g., quay.io) focus on system packages. However, we found that even on the basis of the small number of vulnerability reports that we analysed, potential security vulnerabilities are frequently present in npm packages used in Docker images. Thus, a detailed analysis of such packages should definitely be considered in the future to advance in the state of the art and to provide more complete services.

VII. Threats to Validity

Our empirical analysis about technical lag and risk of vulnerabilities only considered official Docker Hub images. Hence, the results cannot be generalised to community images, where the problem is likely to be more important because there is more freedom and changes in such images. In a similar vein, we only analyzed *node*-based images. Other official images that are not based on the *node* image might have npm packages installed on them. Our findings cannot be generalised to them. Finally, our results cannot be generalised to packages in other languages (or package managers). However, the methodology itself can be applied easily to such types of packages.

Our findings might be biased by the limited (and low) number of security reports that were available on Snyk.io for npm packages. Our results are therefore an underestimation of the actual number of npm package vulnerabilities in Docker images, as it is very likely that many vulnerabilities are unknown or not reported. However, even with a small number of vulnerability reports we could find a relation between outdated images and their number of npm vulnerabilities.

Also, we only analyzed the presence of vulnerabilities in Docker images at the date of the analysis. To show if packages were vulnerable when images were last updated, a time-related analysis similar to the one considered in IV-A could be performed.

VIII. Conclusion and Future Work

This paper presented an empirical analysis of the use of JavaScript packages in official Docker images based on the *node* image for *Debian* and *Alpine* Linux distributions. 961 unique images were retrieved and analyzed against 1,099 security reports extracted from *Snyk.io*, a well-known registry of vulnerabilities for *npm* JavaScript packages. Based on the notion of "technical lag", we studied the impact of the presence of third-party packages on the technical lag and security of Docker images.

The findings reveal the presence of outdated npm packages in Docker images and the risk of potential security leaks. We found that the technical lag and the number of vulnerabilities are related to the last update date of images, which suggests that npm and Docker users should keep up with the updating process of their base images and installed packages.

As future work, we plan to extend our analysis to third-party packages from other programming languages, thus providing a wider assessment and comparison of technical lag in Docker images. Leveraging on the experience gained, we intend to create a tool to automatically gather and analyze relevant information of packages (e.g., bugs, technical lag) based on periodic snapshots of Docker images, and recommend available updates and patches of such packages.

Acknowledgment

This work was partially supported by the EU Research FP (H2020-MSCA-ITN-2014-642954, Seneca), the Spanish Government (TIN2014-59400-R, SobreVision), the Excellence of Science Project SECO-Assist (O015718F, FWO - Vlaanderen and F.R.S.-FNRS).

References

[1] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[2] Docker Inc, "Docker - build, ship, and run any app, anywhere," https://www.docker.com/, accessed: 01/11/2018.

[3] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on Docker Hub," in *7th ACM Conf on Data and Application Security and Privacy*. ACM, 2017, pp. 269–280.

[4] J. Gummaraju, T. Desikan, and Y. Turner, "Over 30% of official images in docker hub contain high priority security vulnerabilities," https://banyanops.com/blog/analyzing-docker-hub/, 2015.

[5] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the relation between outdated package containers, security vulnerabilities, and bugs," in *SANER*, 2019.

[6] Business Wire, "npm, inc. report: State of javascript reveals most popular web development frameworks," https://www.businesswire.com/news/home/20180104005052/en/npm-Report-State-JavaScript-Reveals-Popular-Web, accessed: 01/11/2018.

[7] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, and D. Izquierdo, "Technical lag in software compilations: Measuring how outdated a software deployment is," in *ICOSS*. Springer, 2017, pp. 182–192.

[8] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2017.

[9] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *ICSR*. Springer, 2018, pp. 95–110.

[10] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Int'l Conf. Software Engineering*. IEEE Press, 2015, pp. 109–118.

[11] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages," in *ICSME*, Sep. 2018, pp. 559–563.

[12] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Working Conf. Mining Software Repositories*. IEEE, 2018.

[13] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *NDSS Symposium*, 2017.

[14] A. Nesbitt and B. Nickolls, "Libraries.io open source repository and dependency metadata," March 2018. [Online]. Available: https://zenodo.org/record/1196312

[15] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on Firefox," in *MSR*. ACM, 2011, pp. 93–102.