An Evaluation of Container Security Vulnerability Detection Tools

Omar Javed
Faculty of Informatics, Università dell Svizzera italiana (USI), Switzerland

Salman Toor
Scientific Computing Division, Department of Information Technology, Uppsala University, Sweden

ABSTRACT

Container is a lightweight virtualization technology which packages an application, its dependencies and an operating system (OS) to run as an isolated unit. However, the pressing concern with the use of containers is its susceptibility to security attacks. Consequently, a number of container scanning tools are available for detecting container security vulnerabilities. Therefore, in this experience report, we investigate the quality of existing container scanning tools by considering two metrics that reflect coverage and accuracy. We analyze popular public container images hosted on DockerHub using different container scanning tools (i.e., Clair, Anchore, and Microscanner). Our findings show that existing container scanning tools do not detect application package vulnerabilities. Furthermore, we find that existing tools do not have high accuracy.

CCS CONCEPTS

• :; • **Security and privacy**; • **Software and application security.**;

KEYWORDS

Additional Key Words and Phrases: Security Tools, Software Vulnerabilities, Empirical Studies, Docker Containers, Mining Software Repositories

ACM Reference Format:
Omar Javed and Salman Toor. 2021. An Evaluation of Container Security Vulnerability Detection Tools. In *2021 5th International Conference on Cloud and Big Data Computing (ICCBDC) (ICCBDC 2021), August 13–15, 2021, Liverpool, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3481646.3481661

1 INTRODUCTION

A cloud-based infrastructure alleviates the challenge of managing and maintaining application services across large dis- tributed computing environments [1][2]. However, the need for faster deployment, better performance, and continuous delivery of application services has led to the introduction of containers [3].

Containerization is a virtualization approach that sits on top of a physical machine and shares its host OS kernel and services [4]. The benefits of using containers over traditional virtualization approaches has lead to its growing adoption in the industry by 40% in the year 2020 [5]. Moreover, it is expected that 47% of the information technology service providers are planning to deploy containers in their environment [6].

One of the leading container technology is Docker that has more than 6 billion downloads [7], and over a million images on DockerHub [8]. However, Docker containers are susceptible to security threats [9]. For example, in July 2017 it was reported that an attacker hosted several malicious Docker (container) images on DockerHub. Before these images were taken down, they were downloaded more than 5 million times, which resulted in 545 Monero digital coins being mined (approximately $900 000) [10].

To identify security issues, Docker Inc. has a scanning service [11], which was formerly known as ProjectNautilus [12]. The service provides automated monitoring, validation, and detection of vulnerabilities for images hosted on DockerHub. However, the service is currently not available for all Docker images (i.e., it does not scan community images). Hence, it is important to investigate vulnerabilities in community-based Docker images.

Furthermore, a number of scanning tools are available (e.g., Clair [13], Anchore [14], and Microscanner [15]), which can analyze official as well as community images hosted on DockerHub. The approach employed by these tools is that it collects package information (e.g., OS packages), and compares it against a vulnerability database.

To demonstrate vulnerability issues in both official and community images on DockerHub, Rui et al. demonstrated vulnerability landscape of DockerHub images by analyzing OS packages [9]. However, their study did not explore vulnerabilities detected in the nonOS package (i.e., main application and dependencies or libraries packaged in the container).

Since there are several different container scanning tools, it is important to assess the quality of these tools to better understand their strengths and weaknesses. Such kind of assessment is important to improve the approach being employed by the vulnerability detection tools. This is because a high quality tool is important for addressing security issues as it can point to the component (i.e., application, library or OS) where the vulnerability exists, which can be fixed in a timely manner [16].

Therefore, in this experience report, we investigate the effectiveness of existing container scanning tools by con- sidering two metrics, Detection Coverage and Detection Hit Ratio (DHR) which reflects the tool's coverage and accuracy (Section 4). Our methodology involves crawling 1000 Docker images on DockerHub and selected the ones that fulfills popularity, tool support and viability of our defined analysis (i.e., availability of open-source security tools which can be used to analyze the code and reproducibility of the analysis). One of our main criteria is popular Java-based container applications because Java is one of the most popular programming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCBDC 2021, August 13–15, 2021, Liverpool, United Kingdom
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9040-8/21/08…$15.00
https://doi.org/10.1145/3481646.3481661

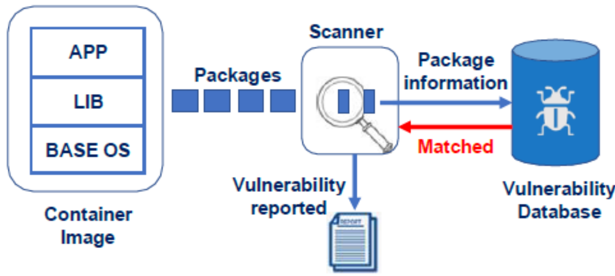

Figure 1: A typical container scanning approach for package vulnerability detection

languages (https://pypl.github.io/PYPL.html., which makes it susceptible to security attacks. For example, a vulnerability related to remote code execution (CVE-2018-11776) was found in Apache struts [17]. This has affected many users and developers. It has been estimated that almost 65% of the Fortune 100 companies use Apache structs [18]. This makes Java-based container applications important to study.

Based on this premise, we select real-world Docker images of Java-based container applications based on their popularity in terms of download. Furthermore, we analyze popular Docker images to investigate the following research questions:

RQ1: What are the most vulnerable nonOS packages present in Java-based container applications and can these vulnerabilities be easily fixed?

RQ2: What is the detection coverage of the existing container vulnerability scanning tools?

RQ3: What is the accuracy of vulnerability detection of existing container scanning tools?

Our study indicate shortcomings in the approach used by the existing container scanning tools, which we hope researchers and practitioners will improve in designing new container scanning tools. In this regard, our study makes the following key contributions:

We use two metrics for assessing the tools' quality, which are the coverage and accuracy of the container scanning tools (Section 4).

We find that the applications packaged in Docker images are missed by the container scanning approach, making detection coverage of the tools questionable (Section 6.3).

We evaluate the accuracy of existing container scanning tools and find that tools are missing a significant number of vulnerabilities (Section 6.5).

Dataset We provide complete information of the detected vulnerabilities, along with a list of all analyzed Docker images with its project information in the dataset [19].

2 BACKGROUND

In this section, we first explain the container scanning approach employed by the existing tools, followed by a discussion on existing container vulnerability detection tools and services. Furthermore, we will also discuss studies that investigate vulnerabilities in Docker images.

2.1 Container scanning approach

In a typical container scanning approach, a number of packages in a container image are scanned by the tool, which analyzes the package name and its version number. This information is then compared against a vulnerability database, which contains a list of entries from publicly known security attacks (or exposures) also known as Common Exposures and Vulnerabilities (CVE). If the analyzed package and its version matches the entry in the database, this is reported as a vulnerability. The scanning approach is shown in Figure 1.

Based on this idea, several different types of vulnerability assessment tools and services have been provided, such as official Docker scanner, GitHub security alerts and enterprise service (e.g., RedHat Containers). Docker Inc. has a scanning service [11], which provides monitoring, validation, and identification of vulnerabilities for official container images hosted on DockerHub. However, the service is currently not available for community-based images. Similarly, GitHub also provides a service for alerting about vulnerabilities in the dependencies of a project [20]. However, the service is relatively young, and is expanding to support multiple languages.

2.2 Docker image analysis

Previous studies have conducted Docker image vulnerability analysis. Martin et al. [3] provide an assessment based on a literature survey regarding the exploitation and mitigation of Docker images from an external attack. However, the study does not assess real-world Docker images hosted on DockerHub to identify security vulnerabilities that can potentially be exploited by an attacker.

Wist et al, [21] analyze vulnerability landscape of DockerHub images to identify most vulnerable type of images and packages. Moreover, Rui et al. [9] conduct a study on security vulnerabilities in both official and community images on DockerHub. However, the study analyzes vulnerabilities that are related to only OS packages, and the authors do not study the vulnerabilities that are present in non-OS packages. Furthermore, the evaluation and the analysis is conducted on only one container scanning tool (i.e., Clair). We, on the contrary, investigate the effectiveness of the existing container scanning tools.

3 TOOL SELECTION

Table1 lists different types of Docker container scanning tools. We explored a number of different container scanning tools from different sources [27][28][29], and identified around 30 container scanning tools. Commercial tools are excluded because complete features of the tools are not available in a trial version. Therefore, it will not provide an accurate comparison of the tool.

Furthermore, we assess only those container scanning tools, which detect package vulnerabilities. This is because such vulnerabilities are of high concern when deploying containers [9]. In Table 1, we highlight (in dark grey color) tools whose functionality is to detect vulnerabilities in OS and/or non-OS packages. From the table, we can see that out of 8 tools, only Anchore identify vulnerabilities in non-OS packages, which shows that there is a lack of container scanning tools that detect vulnerabilities in application and its dependencies packaged in the container.

Table 1: List of Docker scanning tools

Name	Functionality
Clair [13]	Identifies OS vulnerabilities in Docker images
Anchore [14]	Identifies OS and nonOS vulnerabilities in Docker images
Microscanner [15]	Identifies OS vulnerabilities in Docker images
AppArmor [22]	Prevernt access to filesystem and network
Calico [23]	Virtual networking security detection
Cilium [24]	HTTP-layer and network layer security detection
Open Policy Agent [25]	Security policy enforcement
Notary [26]	Verify the integrity and origin of the content

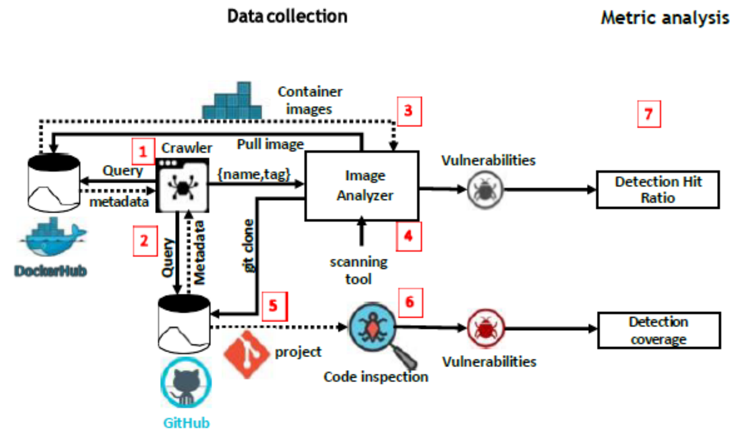

Figure 2: Methodology for collecting and analyzing Docker images. Steps for data collection and metric analysis are marked in red box with numbers. The description of these steps are provided in section5

4 METRIC SELECTION FOR ASSESSING TOOL QUALITY

Our study uses two metric to assess the quality of container scanning tools. We define and explain the reason for selecting each of the metric as follows:

Detection Coverage: The lack of container scanning approach analyzing non-OS package of the container is the main reason for selecting this metric. There are three different categories of packages that are contained in a container

— application, dependencies (or library), and OS packages. We investigate whether the existing container scanning tool (i.e., Anchore) which analyzes both nonOS and OS package is able to detect vulnerabilities in application, libraries and OS packages. Hence, this factor indicates tool *coverage.*

Detection Hit Ratio (DHR): This factor demonstrates the tool's effectiveness in terms of vulnerability detection, i.e., the number of vulnerabilities successfully detected by a tool from a given set of vulnerabilities. The higher the detection, the better is its effectiveness. Therefore, this factor indicates tool *accuracy.* Furthermore, the important aspect of computing DHR is the number of detection misses, therefore, we explain our procedure for finding detection miss for each tool in Section 6.4. We compute DHR by using the following formula:

$$DHR = \frac{\text{Detection Hit}}{(\text{Detection Hit} + \text{Detection Miss})}$$

where

Detection Hit is the number of vulnerabilities detected.
Detection Miss is the number of vulnerabilities missed.

5 METHODOLOGY

We present our methodology to automatically find a set of popular Docker images. Our methodology is based on collecting images from DockerHub and their corresponding projects on GitHub. We refer to projects as GitHub repository of the Docker image. The images are analyzed by container scanning tools, and an image's corresponding project code is used for code inspection by using SpotBugs [30] for finding security bugs in the application code.

We first query DockerHub API to find a set of popular public (i.e., community-based) Docker images (1 in figure 2). In order to find the corresponding image source, we query build information[1] of the image, which provides the corresponding project's GitHub link. We then query GitHub API (2 in figure 2) to select only Java projects. To detect vulnerabilities, we analyze the set of Docker images using container scanning tools such as Clair, Anchore, and

[1]https://hub.docker.com/v2/repositories/\protect\T1\textdollar\protect\T1\textbraceleftname\protect\T1\textbraceright/autobuild/

Table 2: Severity level ranked by number of detected vulnerabilities in non-OS packages

Severity level	Number of Detected Vulnerabilities
Medium	10 080
High	3 316
Low	1 137

Microscanner. We refer to this phase as "Image analyzer", which downloads (i.e., pull) the image from DockerHub (see 3 in Figure 2). We use a security plugin of SpotBugs (a static code analyzer) to detect vulnerabilities in the application code (i.e., non-OS) of the Docker image (6 in Figure 2). We demonstrate the reliability of ground truth information of vulnerabilities in the application code in section 6.3. Finally, we evaluate the tools based on two metrics detection coverage and DHR. (7 in Figure 2).

6 EVALUATION

The analysis of this study has been conducted on a community cloud with Ubuntu Linux 16.04.4 LTS operating system. Furthermore, the configuration of the virtual machine is 4x 2.2 GHz vCPUs, 8 GB of RAM. In our evaluation, we will begin by analyzing non-OS package vulnerabilities in order to evaluate detection coverage. Secondly, we will compare three tools and investigate their accuracy by computing their DHR.

6.1 Identifying vulnerabilities in non-OS packages

According to a report, 80% to 90% of applications use OSS library's components [31]. Therefore, we scan Docker images to detect non-OS package vulnerabilities (i.e., OSS libraries used by the container-based application). We find that only Anchore can analyze non-OS packages. To detect non-OS packages used by the container-based application, we selected "non-os" option during image scan by Anchore. This will exclude other type of vulnerabilities such as vulnerabilities from OS package. Clair and Microscanner do not have non-OS package vulnerability feeds.

While crawling DockerHub images, we search for the image's most recent tag name[2]. The identification of tag is necessary because Anchore fails to analyze an image if the correct tag value is not provided. Furthermore, we find that not all images use ":latest" as a tag to indicate its most recent version. Out of 59 images, we find that 37 images use the ":latest" tag for providing its latest image version. Therefore, based on the correct tag value, we analyze the most recent version of 59 Docker images to detect vulnerabilities in non-OS packages.

Table 2 shows different severity levels for the vulnerabilities detected in non-OS packages. We find in total 14 533 vulnerabilities in 800 packages. Out of 14 533, 1 137 are low severity level vulnerabilities, meaning that they do not pose any significant threat and are not exploitable by an attacker, whereas medium and high vulnerabilities can be exploited. Therefore, after filtering low severity level vulnerabilities, there are still 13 396 (i.e., 10 080 and 3 316) vulnerabilities left in 795 packages. This shows that high number of nonOS packages have vulnerabilities that can be exploited by an attacker. Hence, the presence of 13 396 vulnerabilities make these Docker images susceptible to threats. These fixed vulnerabilities were present in 55 (out of 59) Docker images, which are reported in Table-A (see [19]).

6.2 Most Vulnerable nonOS packages

We provide information about nonOS package vulnerabilities (i.e., application and its dependencies packaged in 55 Docker images) in Table-A (refer to [19]). The column "highest" (in Table) presents most vulnerabilities detected in a package, whereas the column "most vulnerable package" represents its corresponding package name. We can observe that in most cases one package is the main culprit in contributing to the total number of vulnerabilities detected. For example, P1 has 128 vulnerabilities detected in 4 packages (or dependencies), in which 120 different vulnerabilities are detected in the MySQL package, which represents more than 90% of the totally detected vulnerabilities in P1. These vulnerabilities in the MySQL package are because of a Denial of Service (DoS) attack.

Our analysis also shows that the most vulnerable Docker image is P39, which is affected by 1 085 different vul- nerabilities. Around 50% of different vulnerabilities in P39 is again because of the MySQL package. We can observe from Table-A that the attack surface on MySQL packages is high compared to other packages. For example, P29 has vulnerabilities detected in two packages with 262 different vulnerabilities, out of which 261 vulnerabilities are because of MySQL. On further investigation, we find that most of these vulnerabilities can be fixed by updating to a new version. From the table-A, we note that two projects (i.e., P43 and P52) have the same package and version (i.e. jackson- databind-2.9.8) but different values. This is because in case of P52 there is only one dependency of jackson-databind-2.9.8 and in P43 there are multiple libraries which are also using the same version of jackson-databind. Therefore, we are calculating accumulated vulnerabilities ('highest' column of table-A) that is being detected in a package. In P55, there are a total of 6 vulnerabilities from six different packages, each having one vulnerability. Therefore, 'highest' column showing vulnerabilities detected in a package is 1 and we just mentioned one package name out of the six different packages, e.g., qs-6.2.1. Based on our analysis, we address our first research question:

RQ1: *What are the most vulnerable nonOS packages present in Java-based container applications on DockerHub and can these vulnerabilities be easily fixed?*

We identified vulnerabilities in nine nonOS packages in Java-based container applications on DockerHub. These packages include mysql, jackson-databind, tomcat, maven, hadoop, gradle, mercurial, batik, and queryparser. For brevity we exclude their version

[2]https://registry.hub.docker.com/v2/repositories/\protect\T1\textdollar\protect\T1\textbraceleftimagename\protect\T1\textbraceright/tags/

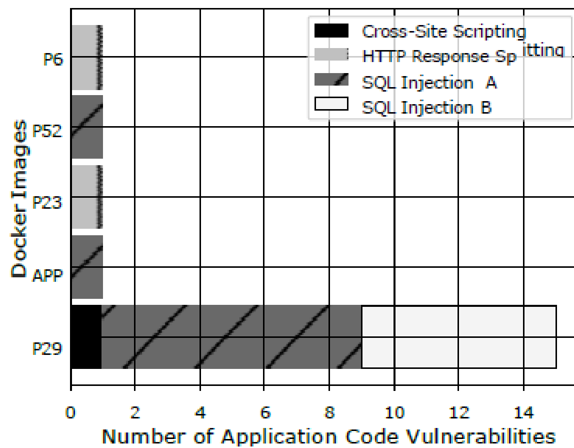

Figure 3: Vulnerabilities detected by code inspection. APP represents an application package vulnerability found in an image which was not detected by Anchore.

numbers. Furthermore, we find that jackson-databind, which is used to provide data binding functionality has vulnerabilities which will affect 19 out of 55 projects.

By checking National Vulnerability Database (NVD) on the identified vulnerabilities, we find that only approx. 8% of vulnerabilities (i.e., 1 039 out of 13 396) do not have fix available. However, major of reported vulnerabilities have fixes such as updating to a new version. Hence, we can answer this question positively by stating that nonOS package vulnerabilities detected in Java-based container applications on DockerHub can be easily fixed.

6.3 Identifying application code vulnerabilities with code inspection

Table-A in [19] shows that that no vulnerability has been detected in the application package of all the analyzed images (this can be seen in the "application" column). To detect vulnerabilities with code inspection, we use SpotBug's security plugin. We analyze the code of 59 popular Docker images by compiling and fetching the project from GitHub. These are the source code of the image.

We find vulnerabilities in application code of 5 Docker image's project, as shown in Figure 3. This figure shows a stacked bar chart with Docker images (i.e., its GitHub project) on y-axis, and number of detected vulnerabilities on the x-axis. There are in total 19 vulnerabilities detected in different application code of 5 projects. Based on our analysis, we did not find these 19 vulnerabilities as false-positives. We provide detailed analysis of vulnerabilities in our dataset [19].

We provide detail about the vulnerabilities identified in the application package which are missed by Anchore. Out of 19 vulnerabilities identified in application package, 16 are because of malicious SQL statement injection, 2 HTTP Response Splitting, and 1 Cross-Site Scripting. Therefore, most of the vulnerabilities detected are due to SQL injection. For SQL related vulnerabilities, developers create a query by concatenating variables with a string. For example; *"Select * FROM customers WHERE name =" + custName.* This makes the code vulnerable to SQL injection, because if an attacker gets hold of the system, s/he can concatenate malicious data to the query [32]. To differentiate between each detected vulnerability, we further categorize SQL injection into 2 categories such as 'SQL injection-A' and 'SQL injection-B'. SQL injection-A represents vulnerability, which occurs due to "Nonconstant string passed to execute or addBatch method on an SQL statement". Similarly, category B represents "a prepared statement is generated from a nonconstant String". We find that P29 contains the highest number of application code vulnerabilities. It has 15 vulnerabilities of 3 different kinds (i.e., cross-site scripting, SQL injection-A and SQL injection-B).

Furthermore, we also find a vulnerability in the application code of Docker image. We refer to this project as APP in Figure 3. Acnhore reported zero vulnerabilities for nonOS package for this project. The name of the image is terracotta/terracotta-server-oss (shown as APP in Figure 3), which is an official Terracotta Server repository having more than 50 000 downloads on DockerHub. This is being missed by Anchore. The reason for missing out on vulnerability is because of lack of container image features (e.g., installed packages and version). For example, Anchore will fail to detect vulnerabilities if vulnerable packages are installed using source code. With these findings, we can now answer our second research question:

RQ2: *What is the detection coverage of the existing container vulnerability scanning tools?*

Our findings from table-A in [19] show that Anchore has feeds which analyzes nonOS package vulnerabilities do not detect vulnerabilities in the application package (even though the container images selected in our study are quite popular). With code-level inspection, we identified 19 vulnerabilities in the application package of 5 different Docker images. Therefore, existing scanning tools are missing out on vulnerabilities that are present in 8,5% (5 out of 59) of images in our dataset. Hence, our analysis highlights a limitation in existing container scanning approach for detection coverage. Furthermore, existing scanning tool such as Anchore also passed an image even though a vulnerability is present in app code.

6.4 Finding detection miss for container scanning tools

In order to compute DHR, we find vulnerabilities detected and vulnerabilities, which are missed by the tool. We find the number of detection by scanning images, and filtering the detected vulnerabilities based on whether it was fixed or not. For OS package vulnerability detection, each tool provides information about vulnerability status (i.e., fixed or not). Therefore, we use this field to filter only fixed vulnerabilities. For finding detection miss by container scanning tools, we formally describe our procedure. Given the set of Docker images (i.e., 59 in this study), let C_i (where $i = \mathbf{1}...n$ | n=3 for this study) be a container scanning tool. Each C_i detects vulnerabilities v_i by matching package vulnerability information from its database. A vulnerability $v_i$consists of fixed and unfixed elements, and is in the form (image name, package name, package version, CVE identifier). Let F_i be the filtered set that contains set of fixed vulnerabilities detected by C_i . Let $F_t$be the filtered set that contains total set of fixed vulnerabilities detected by all tools.

Table 3: DHR of container scanning tools

Tools	Detection Hits	Detection Miss	DHR	DHR(%)
Clair	7 215	12 798	0,36	36,05
Anchore	13 149	6 864	0,66	65,7
Microscan	2 617	17 396	0,13	13,08

To compute DHR for each tool, we compare F_i of each tool with F_t. This procedure allows us to find how many vulnerabilities are detected by each tool.

6.5 Comparing DHR of the tool

We describe DHR as a measurement to understand a tool's effectiveness for detecting OS package vulnerabilities. For this measurement, we collect all fixed vulnerabilities (i.e., confirmed vulnerabilities) that were reported by all three tools. From this set of vulnerabilities, we then identify how many vulnerabilities are detected (and missed) by a tool.

Table 3 shows the DHR for each tool. The worst detection capability among the three tool is that of Microscanner. The DHR for Microscanner is very low, i.e., only 13,08%. On the other hand, Anchore shows the best DHR among the three tools which is 65,7%. While Anchore is the best performing tool, it is still missing 6 864 i.e., ≈34% of OS package vulnerabilities, which are being detected by Clair and Microscanner.

Based on our findings, we can now answer our second research question:

RQ3: *What is the accuracy of vulnerability detection of existing container scanning tools?*

Based on the metric DHR, we find that Anchore has a better DHR (65,7%) compared to the other two tools. This is because Anchore has a frequent vulnerability update mechanism. The anchore-engine periodically fetches newer version of the vulnerability data, and if the vulnerability information appears in the database during the scan, the engine will report the analyzed package having a vulnerability. Even with the frequent update mechanism, Anchore still misses ≈34% vulnerabilities making all tools miss considerable vulnerabilities.

7 CONCLUSION

The use of containers present a security concern for which a number of container scanning tools have been developed. In this experience report, we investigate the effectiveness of container scanning tools based on two metrics which represents coverage, and accuracy. Our analysis shows that both coverage and detection hit ratios does not provide encouraging results of the tools. Based on our experience from this study, we encourage the research community to focus on improving and developing better container vulnerability detection tool. In the future, we plan to extend our analysis to understand vulnerabilities that result due to problems in container environment.

ACKNOWLEDGMENTS

This research was undertaken as part of the eSSENCE strategic collaboration for eScience[3] id="fn3"> http://essenceofescience.se/tag/uppsala/ </fn>. We would also like to thank SNIC cloud under the project number SNIC 2021/18-7 for providing the computational resources.

REFERENCES

[1] Shruti Chhabra and Veer Sain Dixit. 2015. Cloud Computing: State of the Art and Security Issues. *ACM SIGSOFT Software Engineering Notes* (2015), 1–11.

[2] Leonardo Montecchi, Nicola Nostro, Andrea Ceccarelli, Giuseppe Vella, Antonio Caruso, and Andrea Bondavalli. 2015. Model-based Evaluation of Scalability and Security Tradeoffs: A Case Study on a Multi-Service Platform. *Notes in Theoretical Computer Science* (2015), 113 – 133.

[3] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. 2018. Docker ecosystem - vulnerability analysis. *Computer Communications* (2018), 30–43.

[4] David Bernstein. 2014. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* (2014), 81–84.

[5] 451research. 2020. Container status.https://451research.com/images/Marketing/press_releases/Application-container-market-will-reach-2-7bn-in-2020_final_graphic.pdf?p=job%2FojM67fw2. (Accessed on 10/07/2020).

[6] Diamanti. 2020. Container adoption benchmark survey. (Accessed on 10/10/2020).

[7] Docker blog. 2020. InfraKit, a toolkit for creating and managing declarative, self-healing infrastructure.https://blog.docker.com/2016/10/introducing- infrakit-an-open-source-toolkit-for-declarative-infrastructure/. (Accessed on 12/10/2020).

[8] DockerHub. 2020. DockerHub Main page.https://www.docker.com/products/docker-hub. (Accessed on 12/08/2020).

[9] Rui Shu, Xiaohui Gu, and William Enck. 2017. A study of security vulnerabilities on DockerHub. *Data and Application Security and Privacy* (2017), 269–280.

[10] Arstechnica. 2020. Backdoored images downloaded 5 million times finally removed from DockerHub.https://arstechnica.com/information-technology/2018/06/backdoored-images-downloaded-5-million-times-finally-removed-from-docker-hub/. (Accessed on 09/10/2020).

[11] Docker. 2020. Docker Security Scanning.https://beta.docs.docker.com/v17.12/docker-cloud/builds/image-scan/. (Accessed on 10/10/2020).

[12] Docker blog. 2020. Docker Security Scanning safeguards the container content lifecycle.https://blog.docker.com/2016/05/docker-security-scanning/. (Accessed on 05/10/2020).

[13] Clair. 2020. Vulnerability static analysis for containers.https://coreos.com/clair/docs/latest/. (Accessed on 02/11/2020).

[14] Anchore. 2020. The Open Platform for Container Security and Compliance.https://anchore.com/. (Accessed on 15/11/2020).

[15] Aqua's Microscanner. 2020. Microscanner: New free image vulnerability scanner for developers.https://www.aquasec.com/news/microscanner- new-free-image-vulnerability-scanner-for-developers/. (Accessed on 10/04/2020).

[16] Lotfi Ben Othmane, Golriz Chehrazi, Eric Bodden, Petar Tsalovski, and Achim D. Brucker. 2017. Time for Addressing Software Security Issues: Prediction Models and Impacting Factors. *Data Science and Engineering* (2017), 107–124.

[17] Synposys. 2020. The latest Apache Struts vulnerability.https://www.synopsys.com/blogs/software-security/cve-2018-11776-apache-struts- vulnerability/. (Accessed on 09/11/2020).

[18] JAXenter. 2020. Apache Struts threatens remote code execution.https://jaxenter.com/new-vulnerability-discovered-apache-struts-148646.html. (Accessed on 09/12/2020).

[19] Dataset. 2020. Vulnerability detection.https://www.dropbox.com/sh/j831hoqzb0fb60u/AADf5mMhFHJEiaUn3i46CYYla?dl=0. (Accessed on 05/12/2020).

[20] GitHub. 2020. Introducing security alerts on GitHub.https://github.blog/2017-11-16-introducing-security-alerts-on-github/. (Accessed on 10/08/2020).

[21] Katrine Wist, Malene Helsem, and Danilo Gligoroski. 2020. Vulnerability Analysis of 2500 Docker Hub Images. arXiv:2006.02932[cs.CR]

[3] http://essenceofescience.se/tag/uppsala/<fn

[22] App Armor. 2020. The Linux Kernel documentation.https://www.kernel.org/doc/html/v4.15/admin-guide/LSM/apparmor.html. (Accessed on 10/04/2020).
[23] Calico. 2020. Project Calico - Secure Networking for the Cloud Native Era.https://www.projectcalico.org/. (Accessed on 10/08/2020).
[24] Cilium. 2020. Cilium.https://cilium.io/. (Accessed on 10/08/2020).
[25] Open policy. 2020. Open Policy Agent.https://www.openpolicyagent.org/. (Accessed on 12/10/2019).
[26] Notary. 2020. Trust over arbitrary collections of data, https://github.com/theupdateframework/notary. (Accessed on 10/08/2020).
[27] Kubedex. 2020. Container Scanning - kubedex.com.https://kubedex.com/container-scanning/. (Accessed on 10/04/2020).
[28] Sysdig. 2020. 29 Docker security tools compared.https://sysdig.com/blog/20-docker-security-tools/. (Accessed on 10/10/2020).
[29] Techbeacon. 2020. Top open-source tools for Docker security.https://techbeacon.com/security/10-top-open-source-tools-docker-security. (Accessed on 10/10/2020).
[30] SpotBugs. 2020. Find bugs in Java Programs.https://spotbugs.github.io/. (Accessed on 12/10/2020).
[31] Snyk. 2020. Snyk - The state of open source security.https://snyk.io/opensourcesecurity-2019/. (Accessed on 12/07/2020). [28]
[32] Stephen Thomas, Laurie Williams, and Tao Xie. 2009. On automated prepared statement generation to remove SQL injection vulnerabilities. Information and Software Technology (2009), 589 – 598.