Computers, Materials & Continua
DOI: 10.32604/cmc.2022.025096
Article

Tech Science Press

DAVS: Dockerfile Analysis for Container Image Vulnerability Scanning

Thien-Phuc Doan and Souhwan Jung*

School of Electronic Engineering, Soongsil University, Seoul, 06978, Korea
*Corresponding Author: Souhwan Jung. Email: souhwanj@ssu.ac.kr
Received: 11 November 2021; Accepted: 31 December 2021

Abstract: Container technology plays an essential role in many Information and Communications Technology (ICT) systems. However, containers face a diversity of threats caused by vulnerable packages within container images. Previous vulnerability scanning solutions for container images are inadequate. These solutions entirely depend on the information extracted from package managers. As a result, packages installed directly from the source code compilation, or packages downloaded from the repository, etc., are ignored. We introduce DAVS–A Dockerfile analysis-based vulnerability scanning framework for OCI-based container images to deal with the limitations of existing solutions. **DAVS** performs static analysis using file extraction based on Dockerfile information to obtain the list of Potentially Vulnerable Files (PVFs). The PVFs are then scanned to figure out the vulnerabilities in the target container image. The experimental shows the outperform of DAVS on detecting Common Vulnerabilities and Exposures (CVE) of 10 known vulnerable images compared to Clair– the most popular container image scanning project. Moreover, DAVS found that 68% of real-world container images are vulnerable from different image registries.

Keywords: Container security; vulnerability scanning; OCI image analysis

1 Introduction

Virtualization is applied to many fields in Information and Communication Technology (ICT) systems. In particular, 5G networks use virtualization as a way to optimize hardware capabilities. Traditional VM technology faces a performance problem due to virtualizing numerous components that were not necessary for the system repeatedly (i.e., multiple identical VMs running multiple kernels), which indirectly wastes unnecessary computation resources. In contrast, the container shows better performance [1]. By sharing the OS kernel, containers give significantly higher performance than VMs. Docker containers are being considered as a replacement for virtual machines in high-performance systems.

Container technology has many advantages over VMs, but they are not secure enough. Sultan et al. [2] pointed out four threats models of Container security and a set of protecting solutions, from software-based to hardware-based. Gao et al. argued that several theoretical attack vectors

break resource control (cgroup) [3] in the Linux container system. Therefore, developers often need to concentrate on checking for the vulnerability of their product, especially the container image.

Current vulnerability analysis solutions for Docker containers are inadequate. Scanning tools such as Trivy [4], Clair [5] entirely depend on the information extracted from package managers (e.g., dpkg, apk). Firstly, the scanning solution reads the metadata of the package manager inside the container image (e.g., dpkg metadata file is stored inside */var/lib/dpkg/status*). Then, the tool gets the package name as well as the version then compares it with the CVE database to point out the known vulnerabilities. However, applications installed directly from the source code compilation, downloaded from the repository, etc., are left out. Subsequently, vulnerabilities coming from these packages are not detected. In other words, the coverage of current solutions is low.

We propose **DAVS**–Dockerfile Analysis for Vulnerable Scanning. **DAVS** leverages a copious amount of information in the Dockerfile to scrutinize the corresponding container image. This scheme statically analyzes the container image, even without Dockerfile attachment needed, to obtain a Potentially Vulnerable Files (PVFs) list. The *PVFs* ingress into a Vulnerability Checking module for known CVEs (Common Vulnerabilities and Exposures) detection. This procedure helps DAVS deal with the shortcoming of previous scanning solutions by focusing on in-build-time compiled, downloaded, and added packages that previous vulnerable scanning solutions have not done. Moreover, Clair or Trivy can cooperate with DAVS to increase coverage.

This paper has the following contributions:

- A new scheme for analyzing Dockerfile, which tracks the behavior while constructing a container image, is introduced. This method is helpful to expose the installed packages' name and their version.
- This paper proposes DAVS-a framework to statically analyzes **Dockerfile** to extract **Potentially Vulnerable Files** (PVFs) that help detect known vulnerabilities (i.e., represented in the form of CVE metadata) in container images more efficiently.
- By using DAVS, this work gives a high warning to the community of the risks of using public container images.

To sum things up, this work proposes a new system to overcome the imperfections of previous container image vulnerability scanning. By pinpointing vulnerabilities in container images, DAVS helps to reduce security risks when deploying applications to the cloud or edge computing system. Although these systems use many different container engines, DAVS is applicable because it supports all OCI-compliant container engines (e.g., Kubernetes uses CRI-O, Docker, contained as its container runtimes). Therefore, DAVS can help to enhance the security of most current container systems.

This paper is organized as follows: Related works are listed in Section 2 to express the limitation of previous solutions. Section 3 introduces container technology and how to make a container image. Section 4 describes the architecture of DAVS and explains components and algorithms. The authors show how they evaluate DAVS in Section 5 and discuss their proposed framework in the last section.

2 Related Work

Docker image security is an essential part of Docker container system security. Docker advises developers and image composers to suppose that their distributed pipeline contains some actively harmful packages [6]. Docker provided the *Notary framework* to create a content trust network that can be enabled in a container registry [7]. On the other hand, several tools perform the vulnerable checking for container images [8]. Some open their source code, such as Clair, Trivy, Anchore [9],

Dagda [10]. Some provide a premium feature combined with their cloud or Docker Registry service Docker Security in Docker Enterprise.

These vulnerable scanning might have different techniques or architectures. However, the overall methods are similar. These tools collect vulnerabilities from many sources such as National Vulnerability Database (NVD), Common Vulnerabilities and Exposures (CVE), then store this information into a vulnerable database with the application (or package) name, version, CVE identity and the affected OS.

These scanners extract the installed packages and dependencies information inside the container image without running the container. They try to get as much detail as possible about the packages and dependencies, such as name and version. Finally, these tools compare with the vulnerable database to generate the final report.

Many studies have relied on the vulnerability scanning tools mentioned above to assess the security of image containers [11–17]. Shu et al. built DIVA–a framework that automatically discovers, downloads and analyzes container images on Docker Hub [2]. Their framework relied on the power of Clair to statically identify vulnerabilities. Michael Falk and his partner Oscar Henriksson utilized Outpost24's scanner to check the top 1000 Docker images [18]. Wist et al. downloaded and scanned over 2500 images from Docker Hub using the *Anchore* framework [19]. Liu et al. extracted any executed programs (e.g., JAR, Shell script) in the container images and scanned them using Virus totals [20]. Their proposal helps detect malicious container images. However, it cannot work for vulnerable detection. In a word, current studies on vulnerability checking on container images primarily leverage popular scanning solutions for their statistical research. Tab. 1 shows the characteristics of state-of-the-art techniques.

Table 1: The comparison of previous vulnerability scanning solutions for container images. *The symbol* × *means "not applicable."*

Scanner	Static analysis			Dynamic analysis
	OS packages	Application dependencies	Added binaries	
Trivy [4]	apk, rpm, dpkg, yum	gem, pip, poetry, composer, yarn, cargo	×	×
Clair [5]	apk, rpm, dpkg, yum	python	×	×
Anchore [9]	apk, rpm, dpkg, yum	gem, pip, npm	file contents	×
Dagda [10]	apk, rpm, dpkg, yum	java, python, nodejs, js, ruby, php	×	malware analysis using ClamAV
Liu et al. [20]	same Anchore	same Anchore	shell script, CMD parameters	strace, tcpdump
Docker Hub	apk, rpm, dpkg, yum	×	×	×

Previous container image scanning tools extract information from the packages, libraries and software within the image. However, these techniques are highly dependent on the Package manager (e.g., *apt*, *yum*, *dkpg*). Consequently, packages installed directly through compiling, downloading, or adding executables precisely to the image are ignored. Therefore, an additional solution is needed to extract the required information about the custom applications (i.e., the directly installed packages).

3 Background

3.1 Container Technologies

Docker container is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings [21]. Containers are isolated from each other and bundle their software, libraries and configuration files. They can communicate through well-defined channels. Containers require fewer resources than virtual machines since they are shared the same operating system kernel.

The Open Container Initiative (OCI) has been working since 2015 to standardize a container system's structures, specifications and workflows. At present, OCI contains two specifications: Runtime specification and Image specification.

In the Image specification, OCI expresses that the container image has a layered structure. All of the layers are read-only, which means they cannot be modified while using it. Any changes will be compressed and updated as a new layer of the new image's version (tag). Container images become containers at runtime. Then a read-write layer on top of the other layers will be generated for the operation. This architecture ensures that multiple containers using the same image have no collision.

3.2 The Making of Docker Container Image

As shown in Fig. 1, a Docker image is made by using two main ways. First, the image is built using Dockerfile. Docker builds images automatically by reading the instructions from a Dockerfile, a text file containing all commands to build a given image. A Dockerfile adheres to a specific format and set of instructions. *FROM, COPY and RUN* instructions become three corresponding layers from the bottom up to the top. Each layer has a hash value to distinguish, in case of the same Instruction, but different files, while the building executes, is added into the layers.

Second, the image is created after the runtime container changes (e.g., installing new packages, adding files, modifying files or directories). All the changes will be concentrated inside a new layer. The new image should have a new name tag as the identification to distinguish from others.

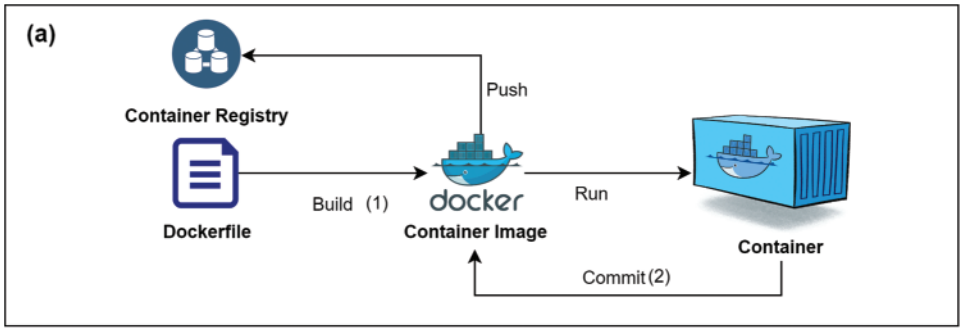

Figure 1: (Continued)

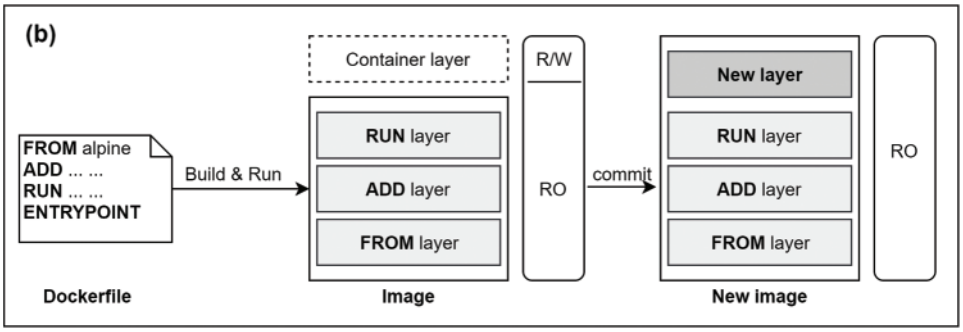

Figure 1: The making of a container image. (a): Container lifecycle (b) Two ways to make an OCI-based container image

4 DAVS Design

We take advantage of Dockerfile to track potentially vulnerable files (PVFs). In detail, Dockerfile contains a set of instructions that guide the docker daemon to run commands to build the docker image. By inspecting Dockerfile, the activity of container image-making progress is revealed. DAVS, as shown in Fig. 2, the container image will be analyzed through three steps: *Reversing Dockerfile*, *Layer mapping*, *Potentially Vulnerable Files extraction*. After these steps, the layer objects containing PVFs information are fed to the CVE-Bin-Tool to detect CVEs related to each container image layer.

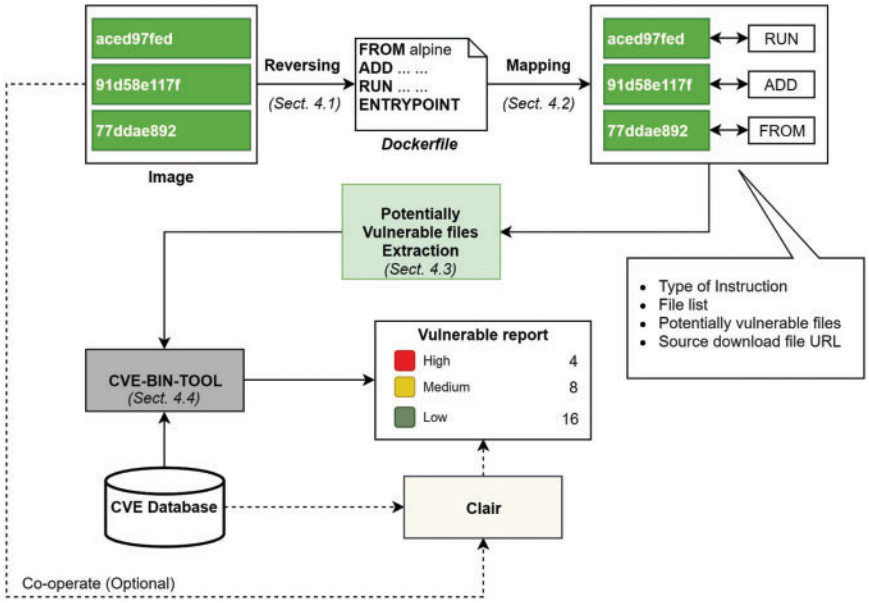

Figure 2: DAVS design. Container image scanning using Dockerfile analysis

4.1 Dockerfile Reversing

While storing the container image in a registry, Dockerfile is not attached in the compressed image. Therefore, we need a method to reverse (or extract) the Dockerfile from an unknown container image.

The OCI does not mention how to make an OCI-based image. However, there is a list of configurations that a container image should follow. Docker donated their image specification to OCI. In other words, OCI images can be built from Dockerfile. Buildah [22] is a popular project that supports build OCI-based images from Dockerfile. In contrast, a Docker image should have a similar configuration to an OCI-based image. Each container image contains the manifest file, which stores general information about the image, including the activities during the image-making progress. Taking advantage of the information described in the OCI standard, we develop the Dockerfile reversing algorithm from any OCI-based image (including Docker container image).

The workflow of *Dockerfile reversing algorithm* is described in detail in Fig. 3. The algorithm is developed by following two main steps. First, we extract the history field from the manifest file. There are several components inside the history field. Each component contains three properties: *created*, *created_by* and *empty_layer*. The *created* property informs the time of the action mentioned in *created_by*. The *empty_layer* property tells us whether this action will create an empty layer or not. Second, we translate the history component to the instruction name. The *created_by* property has two types of initiation: *#(nop)* and *non-#(nop)*. *Non-#(nop)* command will be translated to *RUN* Instruction. The first component will become *FROM* Instruction. *#(nop)* command contains the Instruction's name right after the *#(nop)* mark.

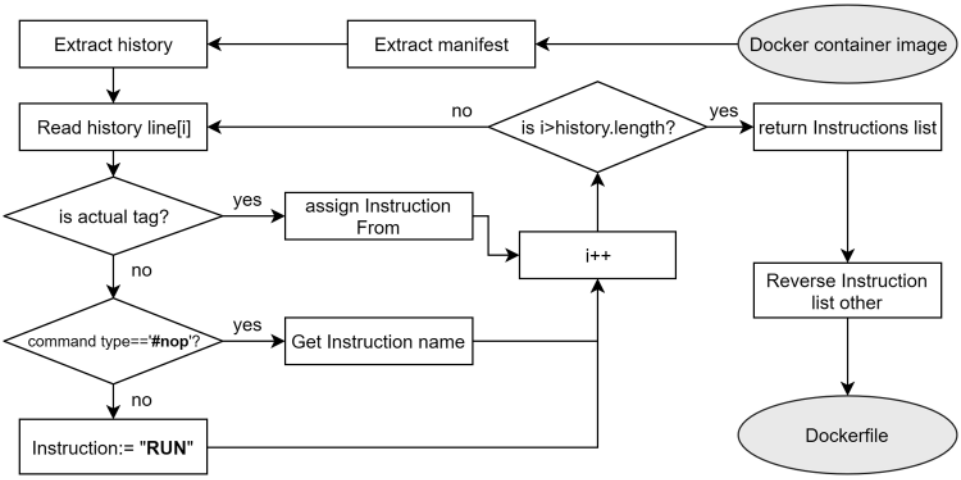

Figure 3: Dockerfile reversing algorithm for OCI-based container image

4.2 Layer Mapping

After getting the Dockerfile from the container image, we need to align each Instruction with its respective layer correctly. The Layer Mapping module ensures the correct process for extracting PVFs. For example, the analysis of Dockerfile figures out that the second Instruction (e.g., RUN command try to download a binary from the Internet) could generate PVFs. However, the second Instruction is used to guide the building framework to make the first image layer. As a result, the failed mapping between each Instruction in Dockerfile makes the finding PVFs broken.

Following the guideline of Dockerfile Reference [23], **FROM, RUN, ADD, COPY** are the instructions that guide the Docker daemon to create a new layer. Based on the Dockerfile, we can know the order of the Instructions. The layer's name can be extracted from the manifest of the container

image. However, it has a reverse order compared to the Dockerfile. Therefore, we have carefully mapped the layer with the Instruction in Dockerfile.

The mapping **layer object** contains the following structure:

- Type of Instruction: is layer or not
- File list: The list of files that belong to this mapping
- Potentially vulnerable files: The list of files that might be vulnerable
- Layer ID: the corresponding Layer ID of the container image

The mapping process is described in Fig. 4. The method takes the Dockerfile content (reversed from a container image) and the Layer ID list, extracted from the container image manifest, as the input and returns the complete mapping (in a list of mapping layer objects).

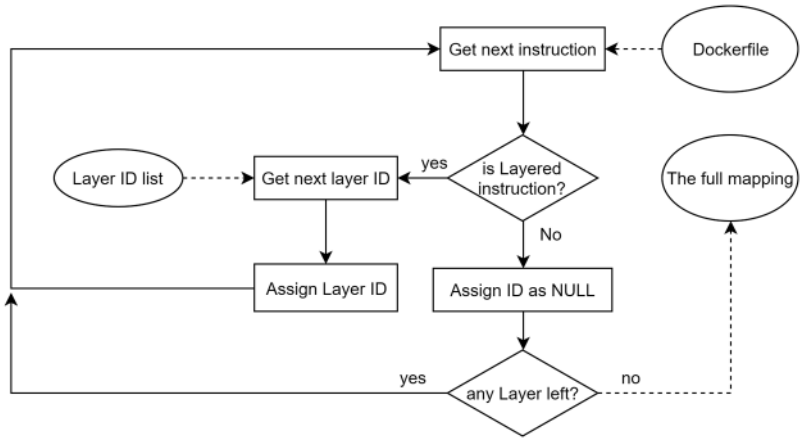

Figure 4: The mapping algorithm of *Dockerfile's Instructions* and OCI-based container image *layers*

4.3 PVF (Potentially Vulnerable File) Extraction

The most crucial step of DAVS is to analyze the Dockerfile to locate which part of the container image needs to be checked. Dockerfile neatly reflects the activities of the image-making process. Following this behavior, DAVS determines the list of files that we need to scan further, but these files should not overlap with the previous scanning solution for saving cost.

Potentially Vulnerable Files (PVFs) are the list of files that are added into the layer or downloaded from the local network (or the Internet) or other file-sharing sources. These files are custom service which is not installed by the package manager (e.g., dpkg). PVFs are also binary files with the executing privilege or shared object files intended to load by a program. DVAS will not focus on the file that is trusted. For example, the based image from official vendors such as Alpine, Ubuntu, Redhat.

Based on the definition, we filter out the PVFs from the container image layer-by-layer using the Dockerfile information and the mapping from the previous step with the following rules:

- **FROM instruction layer**: The files inside FROM instruction layer are from the based operation system (e.g., alpine, ubuntu, centos), from which the container image is built. Therefore, we don't need to extract the vulnerable files list from this layer if the layer is from a well-known operating system.
- **ADD/COPY instruction layer**: The files inside this layer are always needed to check for vulnerability. Therefore, we need to filter out the files which have the executable privilege.

- **RUN instruction layer**: In the action guided by *RUN* instruction, the author may download the extra files by using *wget, curl, scp, ftp, nc and* other network establishment commands. In addition, if the *RUN* instruction contains make, build, setup and other compiling commands required to be checked. Moreover, if *RUN* instruction contains downloadable and compilable commands, all the executable files will become PVFs.

Fig. 5 shows the parts in Dockerfile covered by DAVS and scanning tools (e.g., Clair). The highlighted sections are files added to the container image or loaded from the Internet and directly compiled during image building. Besides, the scanning tool only covers the packages installed in the image through the package manager apt-get. Obviously, this image's most important components (i.e., *FFmpeg application)* are compiled directly via the make install command. Therefore, the package manager (e.g., *apt-get*) cannot store FFmpeg's information. The scanning tool will now ignore large amounts of information about the packages contained in the images, resulting in a lack of vulnerability detection.

```
FROM ubuntu:18.04
ADD file:8a9218592e5d736a05a1821a6dd38b205cdd8197c26a5aa33f6fc22fbfaa1c4d in
/
CMD ["bash"]
LABEL maintainer=phithon <root@leavesongs.com>
RUN /bin/sh -c apt-get update \
    && apt-get install -y autoconf automake build-essential [...]\
    && wget -qO- https://www.ffmpeg.org/releases/ffmpeg-2.8.4.tar.gz
    && cd /usr/src \
    && ./configure --pkg-config-flags="--static" --disable-yasm \
    && make \
    && make install \
    && rm -rf /usr/src/*
CMD ["ffmpeg"]
LABEL maintainer=phithon <root@leavesongs.com>
RUN /bin/sh -c set -ex \
    && apt-get update \
    && apt-get install -y --no-install-recommends php-cli \
    && rm -rf /var/lib/apt/lists/*
```

Figure 5: Comparison of the coverage parts between PVFs extraction method and Clair (a popular image scanning tool). Green highlights represent PVFs extraction coverage. Yellow highlights represent Clair coverage

4.4 Vulnerability Checking

After getting the potentially vulnerable files (PVFs) list, we use CVE-Bin-Tool [24] to extract the name and the version to which PVFs belong. We considered the limitation of CVE-Bin-Tool, which only provides a set of well-known software as well as the string extraction and matching to detect known software versions. However, this work does not focus on finding new vulnerabilities but help developers and container system administrator be aware of the risk of using certain container images.

Fig. 6 shows the workflow of scanning an OCI-based image. The container layer can be reused in several container images. DAVS extracts and scans PVFs independently for each layer (identified by layer ID). Detection results are also saved corresponding to each layer, identified by layer ID. In the case of images used the same couple of layers, the scanned layers do not need to be re-executed to increase system performance.

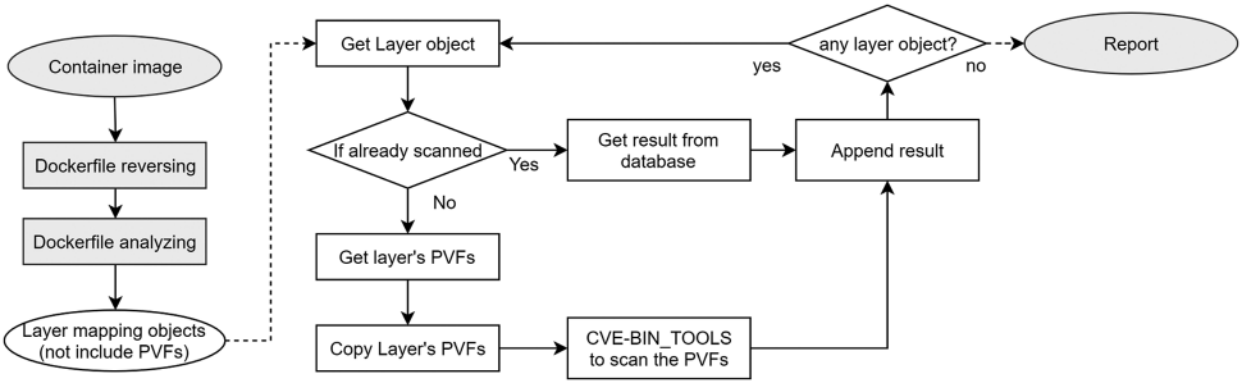

Figure 6: Vulnerability checking progress

5 Experience Setup and Evaluation

This paper has several research questions to evaluate the proposed scheme:

- ***RQ1****: How well does* ***DAVS*** *detect known CVE on vulnerable images over the previous container image scanning solution?*
- ***RQ2****: How much is the detecting coverage rate that DAVS could be used to improve previous container image scanning solutions?*
- ***RQ3****: What is the situation of real-world container image in popular public registries?*

5.1 Experience Setup

We collect and download the container images from public registries to build two datasets: (1) Container images contain known vulnerabilities; (2) The latest updated container images on Docker Hub come from many different developers. For the (1) datasets, we archived 10 images from Vulhub [25], the collection of vulnerable images for training purposes. The detail of dataset (1) is shown in Tab. 2. For the (2) datasets, we archived 1200 top latest updated containers from Docker Hub [26], 200 from quay.io [27].

Table 2: Dataset (1) evaluation. The number of known vulnerabilities related CVEs

Image name	Clair	DAVS	CVE-Bin-Tool, all file scanning
FFmpeg	0	64	64
Bash Shellshock	0	6	6
Postgressql:9.6.7	0	14	14
Ghostscript python:9.21	11	74	74
mysql:5.5.23	0	0	0
nginx:1.4.2	0	9	9
openssh:7.7	0	127	127
Nodejs:8.5.0	0	14	14
Heartbleed openssl	0	88	88
Samba:4.6.3	0	28	28

We use Python3 as the primary coding language for the DAVS framework. The framework is run in an Intel NUC mini PC with a Core-i3 chipset and 8 GBs of Memory, running Ubuntu 20.04 LTS.

5.2 Evaluation

RQ1: Detection of related CVEs to the known vulnerable container images

Dataset (1) is used to check the vulnerability of corresponding image detection accuracy. A package of a specific version may be affected by multiple CVEs. Therefore, we not only define the exact number of related CVEs (i.e., CVEs that affect the package) but also check the CVE's ID that vulnerable images are built for.

We compare the detection accuracy of **DAVS** with **Clair**–The most popular container image scanning solution. Clair scans the whole image as usual. In contrast, **CVE-Bin-Tool** is designed to scan a set of files or directories, not for the container image. Therefore, we extract the image from the repository using Docker save command. After that, we scan the whole extracted file from the image. Tab. 1 shows the evaluation result of the dataset (1). Clair scans images very fast because it extracts package information from package manager metadata. However, Clair gives low accuracy when 9/10 images could not detect the correct CVE ID for which the target images are built. We use **CVE-Bin-Tool** to scan all of the files in the image. Nevertheless, the scanning time is high, as shown in Fig. 7. **DAVS**, with its *PVFs* filtering rules, keeps the high accuracy of detecting related CVEs of the target vulnerable image and significantly decreases the scanning time compared to all files scanning using CVE-Bin-Tool.

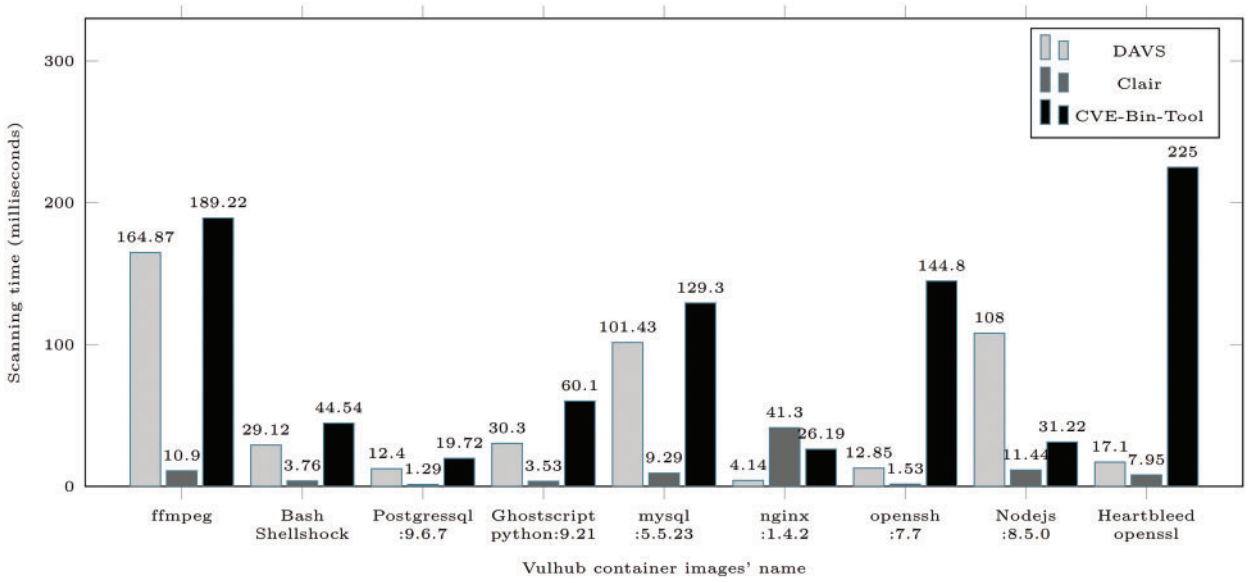

Figure 7: Scanning time comparison (in ms)

RQ2: The extra number of CVEs that DAVS outperform Clair.

We inspect the number of CVEs that DAVS and Clair can detect on both Datasets (1) and (2). We also filter the number of CVEs that DAVS and Clair overlap. As shown in Fig. 8, the number of CVEs detected by DAVS accounts for more than 50% of the CVEs that DAVS and Clair can detect, which proves the PVFs filtering rules work well. On the other hand, the overlap rate is inconsequential.

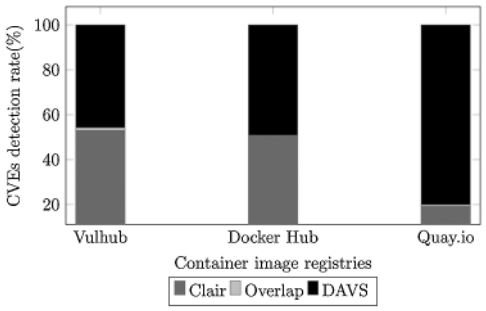

Figure 8: The detected CVEs rate (number of detected CVEs over the total of DAVS and Clair) that DAVS and Clair can archive

We built an automatic crawler for dataset (2) to obtain the list of container images in Docker Hub and Quay.io. Then directly pull and scan the images with DAVS and Clair. While evaluating the real-world container images from DockerHub, we found that DAVS detects an extra 53% of CVEs over Clair. In addition, Clair cannot detect any CVEs in **336** vulnerable container images, which are successfully discovered by our method.

RQ3: The risk of using public container image.

To evaluate the safety of using public container images, we use DAVS integrated with Clair for checking the dataset (2). Through all **1400** images, we found that nearly **68%** of container images are vulnerable, as shown in Fig. 9. Therefore, the use of public container images is not safe, especially from Docker Hub– the most popular container registry.

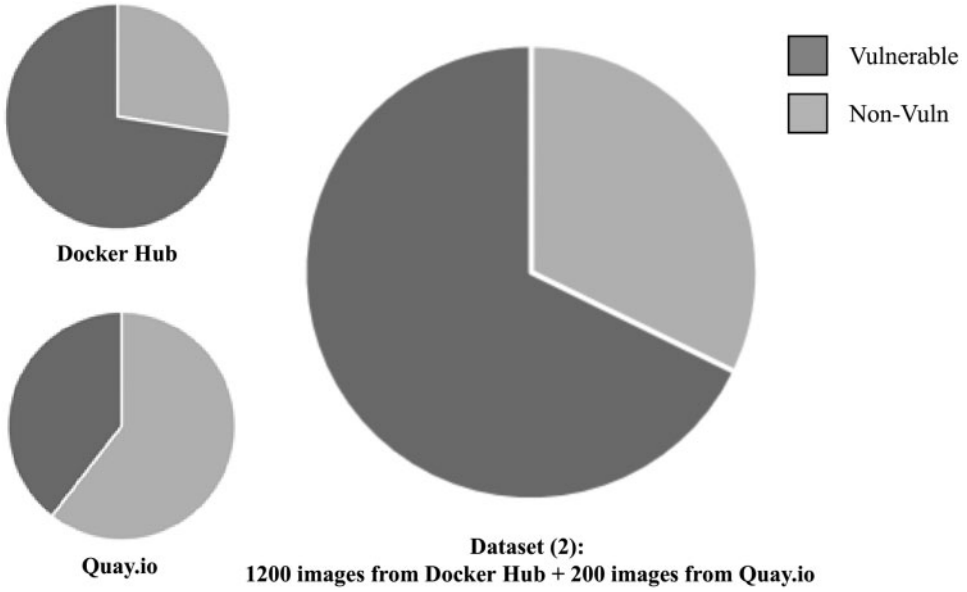

Figure 9: The vulnerable images rate in real-world

6 Discussion and Future Work

DAVS is a solution to overcome the shortcomings of the previous container image scanning tools. While these projects rely on package managers' information, installed packages on the Internet or compiled during the building container image are ignored. With the Dockerfile analysis technique, DAVS tracks the workflow of the container image-making efficiently. Potentially Vulnerable Files are extracted and scanned that help to improve the detecting coverage.

DAVS is still costly since it needs to check a list of files in an image layer. Even the scanning process is divided with the layer-by-layer scanning operation, the complexity of the PVFs filtering algorithm is still high. Moreover, DAVS depends on CVE-Bin-Tool, which has limited functionality, which is only implemented string extraction and inspection, on checking packages' names and versions of unknown binaries inside the container image. We let the optimization for our future work.

We also consider that scanning solutions missed software errors due to many other reasons, such as the late update of the CVE database. Then, dynamic analysis solutions, such as anomaly detection, are needed to enhance the security of the container system. The checking of users' data in the cloud system while preserving privacy is a significant challenge [28–30]. The integrity checking of container images is also one of our future works.

Acknowledgement: This work is the extended work of "DAVS: Dockerfile Analysis for Vulnerable Scanning" in *MobiSec 2021: The 5th International Symposium on Mobile Internet Security*, Jeju Island, South Korea.

Funding Statement: This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea Government (MSIT) (No. 2020-0-00952, Development of 5G edge security technology for ensuring 5G+ service stability and availability.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] R. Dua, A. R. Raja and D. Kakadia, "Virtualization vs containerization to support PaaS," in *2014 IEEE Int. Conf. on Cloud Engineering*, Boston, MA, USA, pp. 610–614, 2014.
[2] S. Sultan, I. Ahmad and T. Dimitriou, "Container security: Issues, challenges and the road ahead," *IEEE Access*, vol. 7, pp. 52976–52996, 2019.
[3] X. Gao, Z. Gu, Z. Li, H. Jamjoom and C. Wang, "Houdini's escape: Breaking the resource rein of linux control groups," in *Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security*, London, United Kingdom, pp. 1073–1086, 2019.
[4] Aquasecurity. *Trivy*, 2021. [Online] Available: https://github.com/aquasecurity/trivy.
[5] Quay.io, Inc. *Clair*, 2021. [Online] Available: https://github.com/quay/clair.
[6] S. P. Mullinix, E. Konomi, R. D. Townsend and R. M. Parizi, "On security measures for containerized applications imaged with docker," *arXiv preprint*, arXiv:2008.04814, 2020.
[7] *Docker. Co. Content trust in Docker*, 2021. [Online] Available: https://docs.docker.com/engine/security/trust/.
[8] V. Jain, B. Singh, M. Khenwar and M. Sharma, "Static vulnerability analysis of docker images," *IOP Conference Series: Materials Science and Engineering*, vol. 1131, pp. 012–018, 2021.
[9] *Anchore, Inc. Container security solutions for devsecops*, 2021. [Online] Available: https://anchore.com/.
[10] Dagda, Inc. *Dagda*, 2021. [Online] Available: https://kubedex.com/resource/dagda/.

[11] V. Adethyaa and T. Jernigan, *Scanning docker images for vulnerabilities using Clair amazon ECS ECR and AWS CodePipeline*, AWS Compute Blog, 2018. [Online] Available: https://aws.amazon.com/blogs/compute/scanning-docker-images-for-vulnerabilities-using-clair-amazon-ecs-ecr-aws-codepipeline/.
[12] K. Brady, S. Moon, T. Nguyen and J. Coffman, "Docker container security in cloud computing," in *10th Annual Computing and Communication Workshop and Conf. (CCWC)*, Las Vegas, NV, USA, pp. 0975–0980, 2020.
[13] B. Kaur, M. Dugŕe, A. Hanna and T. Glatard, "An analysis of security vulnerabilities in container images for scientific data analysis," *GigaScience*, vol. 10, no. 6, pp. 02–05, 2021.
[14] A. Manu, J. K. Patel, S. Akhtar, V. Agrawal and K. B. S. Murthy, "A study, analysis and deep dive on cloud PaaS security in terms of docker container security," in *2016 Int. Conf. on Circuit, Power and Computing Technologies (ICCPCT)*, Nagercoil, India, pp. 1–13, 2016.
[15] A. Zerouali, T. Mens, G. Robles and J. M. Gonzalez-Barahona, "On the relation between outdated docker containers, severity vulnerabilities, and bugs," in *2019 IEEE 26th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China, pp. 491–501, 2019.
[16] O. Javed and S. Toor, "An evaluation of container security vulnerability detection tools," in *2021 5th Int. Conf. on Cloud and Big Data Computing (ICCBDC)*, New York, NY, USA, pp. 95–101, 2021.
[17] M. K. Abhishek and D. Rajeswara Rao, "Framework to secure docker containers," in *2021 Fifth World Conf. on Smart Trends in Systems Security and Sustainability (WorldS4)*, London, United Kingdom, pp. 152–156, 2021.
[18] O. Henriksson and M. Falk, "Static vulnerability analysis of docker images," in *Degree Project in Master of Science in Engineering*, Karlskrona, Sweden: Blekinge Institute of Technology, 2017.
[19] K. Wist, M. Helsem and D. Gligoroski, "Vulnerability analysis of 2500 docker hub images," *Advances in Security, Networks and Internet of Things*, Switzerland: Springer, Cham, pp. 307–327, 2021.
[20] P. Liu, S. Ji, L. Fu, K. Lu, X. Zhang *et al.,* "Understanding the security risks of Docker hub," in *European Symp. on Research in Computer Security*, Guildford, United Kingdom, pp. 257–276, 2020.
[21] Docker. Co. *What is a container*? 2021. [Online] Available: https://www.docker.com/resources/what-container.
[22] Buildah Co. *Buildah*, 2021. [Online] Available: https://buildah.io.
[23] Docker. Co. *Docker reference*, 2021. [Online] Available: https://docs.docker.com/engine/reference/builder/.
[24] Intel. Co. *intel/cve-bin-tool*, 2021. [Online] Available: https://github.com/intel/cve-bin-tool.
[25] Vulhub. *Vulhub-docker-compose file for vulnerability environment*, 2021. [Online] Available: https://vulhub.org/.
[26] Docker. Co. *Docker hub*, 2021. [Online] Available: https://hub.Docker.com/.
[27] Quay.io. *Quayio container image registry*, 2021. [Online] Available: https://quay.io/search.
[28] F. Chen, F. Meng, T. Xian, H. Dai, J. Li *et al.,* "Towards usable cloud storage auditing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2605–2617, 2020.
[29] J. Chang, B. Shao, Y. Ji, M. Xu and R. Xue, "Secure network coding from secure proof of retrievability," *Science China Information Sciences*, vol. 64, no. 12, pp. 1–2, 2021.
[30] Y. Ji, B. Shao, J. Chang and G. Bian, "Flexible identity-based remote data integrity checking for cloud storage with privacy preserving property," *Cluster Computing*, vol. 24, no. 3, pp. 1–13, 2021.