Assessing security vulnerabilities in docker containers: A survey on PaaS cloud ecosystem

Jainendra Kumar
School of Informatics and Computing
Indiana University
Bloomington, United States
jaikumar@iu.edu

Tejas Deoras
School of Informatics and Computing
Indiana University
Bloomington, United States
tdeoras@iu.edu

ABSTRACT

Container technology has really become very popular in the information technology industry these days owing to the plethora of benefits they provide. Major companies in the industry are now undergoing digital transformation where they are transforming their infrastructure to small services and are using containers as a basis. Containers play an important role in moving towards the adoption of Hybrid and Multi cloud computing ecosystem. Enterprises adopting container technology see substantial increase in their application deployment and packaging in terms of agility, speed, productivity and distribution. According to the 451-research analysis, by 2020 the container market would be worth $3 billion where Docker alone makes 83 percent of the market. In this paper we do a survey on current state of security vulnerabilities in Docker Hub images. We measure our survey results with previous studies and observe that the state of security on Docker Hub has improved, although many new vulnerabilities have been discovered. We establish the relation between vulnerable images and how they can easily become malicious by attackers. We also propose that machine learning classification can be used to detect both vulnerable and malicious images.

Keywords
Docker Hub; Docker containers; Vulnerabilities; Vulnerability classification; Containers.

INTRODUCTION

With the popularity of cloud ecosystem containers have become an important technique of quickly running and deploying services on multiple or single hosts. As containers have evolved the popularity has led to the creation of container marketplace which distributes large number of official and community images. Docker Hub is the world's largest library and community for container images. It provides over 100,000 container images from software vendors, open source projects and community.

More and more applications are now adopting microservice based architecture in which several microservices are packaged individually in a container and then loosely connected with other microservices. This makes the entire system more fault tolerant, modular, and hence separates out the concerns. As Docker has captured more than 70 percent of the market our research will focus solely on Docker containers. Unlike virtual machines, Docker's architecture is a little more complex as containers are executed with the Docker engine rather than a hypervisor. They are much smaller than virtual machine and enable faster start-up with better performance, modularity, less isolation and greater integration capability as they share the host's machine kernel.

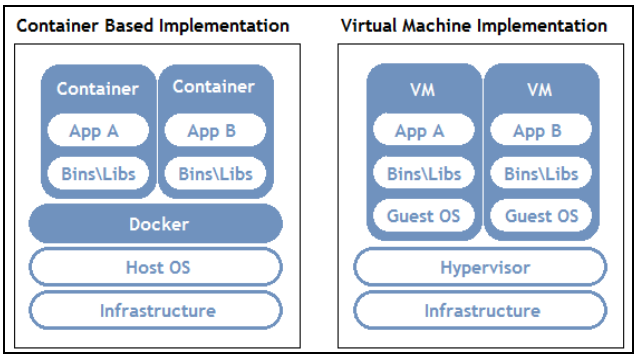

Figure 1. Architeture of container and VM based approach

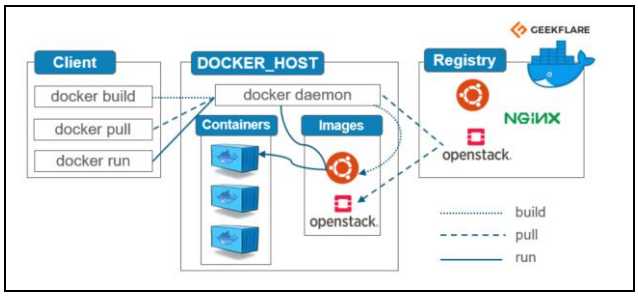

Figure 2. Architecture of Docker

Containers package code, system libraries, binaries and other files needed for the containerized application to run. Thus, containers encapsulate everything and become very modular as compared to a virtual environment running on top a Hypervisor. Docker's architecture is a mix of client-server architecture and involves four main parts: (1) Docker client through which one interacts with the Docker container. (2) Docker objects which are the main components of the container. (3) Docker Daemon which is a background utility responsible of sending commands and (4) Docker Hub which is a marketplace for official and community-based images.

On a Docker Hub images can be inherited from other images. For example, an official maintainer say RedHat can create an image of Ubuntu:18.04 operating system. After installing the necessary packages in the operating system the maintainer can tag the image as Ubuntu:latest or Ubuntu:v1 or Ubuntu:v2 and so on after every update. Any other community member can thus inherit that base Ubuntu image for building his own image for his software. For example, a community member can build an apache web server image on top of Ubuntu image which will act as a base operating system for Apache to run. This way all the packages and dependencies installed on the Ubuntu image will automatically be inherited by the Apache image.

MOTIVATION

As these containers have become popular it is becoming very easy for attackers to take advantage of vulnerabilities in docker images owing to the large number of packages installed in the images. Multiple enterprise surveys have indicated that security has now become a major concern when deploying applications through containers.

On October 18, 2019 security researchers at Palo Alto networks discovered a cryptojacking worm that propagates using containers in the Docker Engine (Community Edition) and has spread to more than 2,000 vulnerable Docker hosts. Dubbed Graboid by the researchers, the worm mines Monero cryptocurrency in shorts busts. Similarly, a dangerous bug in the Kubernetes open-source cloud container made a HTTP protocol violation in the Go language's standard HTTP library which made the proxy server ignore invalid headers and forward them to Go server without proper authentication.

Therefore, we can see as the containers are evolving and becoming more popular it becomes critical to have the images layers to be patched and updated on a regular basis. In addition, enterprises that use these images for their applications should employ a multi-layered security approach towards authentication, authorization, firewalls rules and network monitoring.
This brings us to the main goal of our research:
We want to observe what is the current state of the Docker Hub marketplace and whether the state of security has been improved or diminished. We also propose a novel technique based on machine learning classification which can be used to classify the image as vulnerable and identify certain features that are highly probable of malicious images.

PROBLEM STATEMENT

During this short research we want to address the following questions:

(1) What is the current state of security vulnerabilities present in official and community based images?
(2) What is the relationship between vulnerable images and malicious container images?
(3) Does a malicious image exhibit common characteristic and can those characteristics be used to build a detection system?
(4) How vulnerabilities spread from one container to another?

To answer these questions, we built a framework that automates the discovery, download and analysis of both official and community based docker images. We integrated our framework with Claire which is a vulnerability scanner for Docker images and statically analyzed 120 official images and 2230 community-based images (approx. 2% of all community-based images available) till this date.

We are in the process of analyzing more community images, however this report contains results which we assume will follow a similar trend and proportion if we analyze nearly all the community images.

Our major findings include the following:

(1) Significant number of community images on average contain more vulnerabilities than the official images.
(2) Root cause of vulnerability propagation is still the parent child relationship between the images. As new images are inherited from the base image vulnerabilities spread.
(3) As compared to previous studies on Docker Hub the composition and abundance of vulnerabilities on Docker Hub has significantly reduced.
(4) We propose certain features that can be used to detect a possible malicious image without dynamic code execution.
(5) We observe that vulnerabilities alone do not lead to malicious behaviour in images. Attackers can easily poison a non-vulnerable image. We prove this by performing a POC exploit on a non-vulnerable image.

We acknowledge the fact that we are not the first ones to study vulnerabilities on Docker Hub but to the best of our knowledge we are the first one to test a classification algorithm on Docker Hub images. In May 2016, Docker Inc. announced Docker security scanning service also termed as Docker Bench which is a script that check dozens of security best practices around deploying containers in production. However, our analysis demonstrates that there is still a significant need to safeguard container ecosystem by frequently updating and patching the images. The remainder of the paper goes as following: We describe our framework to discover, download, and analyze docker images. We describe the integration of our framework with Claire, a tool for vulnerability scanning of Docker images. We then describe our findings including the composition of vulnerabilities in both official and community images. We further describe how vulnerabilities spread from one image to another. Then we describe how we can use classification algorithm to classify an image as vulnerable. We describe certain features that link vulnerable image to malicious image and how machine learning models can be used to estimate if an image is malicious or not. We later conclude the paper by highlighting the importance of securing the container landscape by increasing the frequency of security patches and updates to both community and official images.

FRAMEWORK

Firstly, in a very short timeframe of nearly 2-3 months to study the entire Docker Hub registry is nearly an impossible task because of the following two reasons:

1) Although there is a list of official images but there is no official list of community-based images. Thousands of images are added and deleted every day which makes this study a challenge.
2) Docker Hub registry is too large to clone locally. An average image size is around 400 to 600 MB with some being as large as 4 GB. Therefore, downloading all images in one go is not a possible solution.

Therefore, we had to overcome the following challenges and build a framework to automate the image discovery, image download, analysis, report generation and statistics.

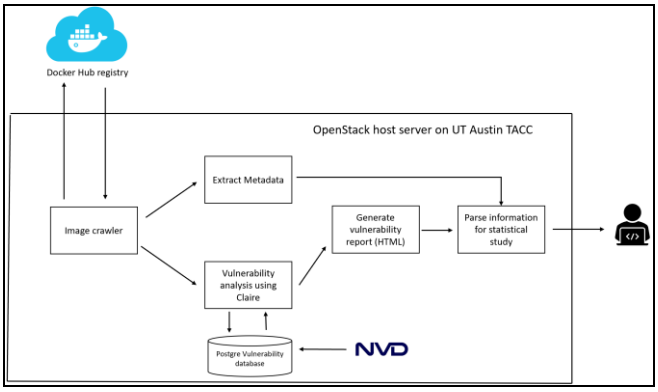

Figure 3. Image Discovery and analysis framework

As shown in Figure 3. our framework is deployed on local university OpenStack server called TACC which is in partnership of University of Austin, Texas and Indiana University. The framework consists of the following parts:

Image Crawler

Image crawler is a stream-based script written in Python which is responsible for pulling unique images from Docker Hub. For now we have used sequential based approach where an image is first looked on the Docker Hub, if found is pulled on the local host. The script then the script calls the second component of our framework i.e. Clair vulnerability scanner which analyses the image and generates the report as an HTML document. We collect the following metadata about the image as given in the table below.

Image ID	256 bit unique ID for each image
Image Name	A named convention for both official and community images
Layer ID	Unique ID of each layer signifying relationship
Image size	Size of images ranging from MB to GBs
Number of pulls	Number of times the image is pulled from the registry
Star rating	Docker hub stars signifying satisfaction score
Last update date	Latest image update time
Category of image	Official or Community

Table 1. Data collected from the images

Once the image is scanned and the vulnerability report is generated the script deletes the image to make way for the next image. This is done so that the host machine does not run out of space as it is nearly impossible to clone the entire docker hub registry on a single host machine.

For analyzing the official images we found the official image list from the Github repository available on the internet. All the official repository names were passed as a string to our script which were sequentially appended to the docker pull command. However, there is no list of community-based images available anywhere on the internet, therefore we used DockerHub API to search images based on the official image name passed to the API as a string. This would result in all the community images matching the given official search string as a JSON object as shown in the Figure 4.

For this limited study and to make our results uniform we extracted the metadata of only top 10 most popular and top 10 least popular community-based images matching a given official image name.

This way we gathered information of about 20 images of each 120 official images.


```
{
    "repo_name": "anapsix/alpine-java",
    "short_description": "Oracle Java 8 (and 7)
    "star_count": 432,
    "pull_count": 42686479,
    "repo_owner": "",
    "is_automated": true,
    "is_official": false
},
```


Figure 4. Sample metadata from Docker Hub API call

Our study shows that on average it took around 2 minutes for the image to be pulled on the host machine depending on the image size which ranges from as small as few megabytes to as big as 3 gigabytes. We also made sure that no two images are downloaded twice by maintaining the list of images which have already been analyzed.

Vulnerability analysis using Clair

Clair is an open source project for the static analysis of vulnerability in appc and docker containers.
Vulnerability data is continuously imported from a known set of sources such as National Vulnerability Database, Red Hat vulnerability database, etc. and correlated with the indexed contents of container images in order to produce lists of vulnerabilities that threaten a container.

We deployed both Clair and its associated Postgre vulnerability database as a docker container using the docker compose file given in the clair documentation.
Once our containers were up and running our image crawler python script would call clair for analysis once the image is successfully pulled.

Official images analyzed	120
Community images analyzed	2230
Average time take to pull and analyze image	6 minutes
Average size of image	680 MB
Estimate data downloaded	1560 GB

Table 2. Image analysis statistics

Clair uses static analysis to collect and extract the following information from the docker images:

Total number of vulnerabilities	Total vulnerabilities in both official and community images
Vulnerability ID	Unique ID of each vulnerability
Vulnerability description	Description specifying the vulnerability details
Severity rating	Rating as per NVD vulnerability metrics
Associated packages	Packages installed which are vulnerable

Table 3. Data collected from Clair

Clair identifies the vulnerable images by matching the image data with Common Vulnerabilities and Exposures database and other databases such as Red Hat Security Data which is constantly streamed and updated via a Postgre database deployed along with Clair. Clair does not perform any dynamic analysis on the containers, however once the image is updated clair does not report the same vulnerability again in the new layer if the vulnerability is patched.

For instance, if the old image is created on the base layer of say Ubuntu and let's suppose the base layer contains a vulnerability in the glib package, then once the image is updated via apt-get update or apt-get upgrade resulting in the new layer then clair would not show the same vulnerability in the new image layer. Therefore, clair takes care to not report vulnerable package present in old layer package if that package is patched in the new layer. We also tested other vulnerability scanners such as Snyk and Anchore engine but we found Clair to be most suitable for our study owning to its high accuracy in analysis, relatively quick image analysis, readily available APIs to extract meta data about the image and easy integration in our framework.

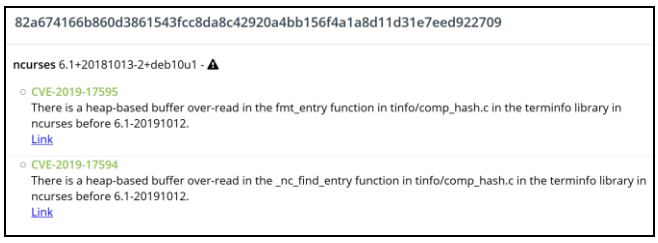

Figure 5. Sample output from clair

HTML image parser

Our final component in the framework is the HTML image parser written in Python. This module is responsible for extracting data from the HTML reports outputted after analysis. Once all the reports are generated and stored, the reports are passed to this module which uses Python's JSoup library to parse data and generate statistics about the reports.

The entire process took about one month to analyze 120 official images and nearly 2% of the total community images. One limitation of our approach during data collection and analysis is to follow a sequential processing of images. The entire process could be sped up had we used multiprocessing on multiple host machines.

OBSERVATIONS

To determine the composition of vulnerabilities on Docker Hub we show several metrics from our analysis. Table 4 and Table 5 shows top 10 images with highest vulnerabilities both official and community.

Official Image	No. of vulnerabilities
node	490
pypy	489
known	254
buildpack-deps	250
perl	250
python	250
ruby	250
drupal	240
mediawiki	240
sentry	229
nextcloud	223

Table 4. Topmost vulnerable official images

As we see in both the table there are few images that are common in both the tables namely pypy official image corresponds to terbiumlabs/pypy-flower and ephillipe/pypy-docker in community image list. Similarly, buildpack-deps official image also has community version named circleci/buildpack-deps. We think that this is because both the official image and its community counterparts share the same library packages and both contain high number of vulnerabilities. It could also be a fact that both images

are created off from the same base images or the community image is inherited from the official image.

Community Image	No. of vulnerabilities
ephillipe/pypy-docker	1240
devacto/clojure-test	1235
kaggle/fsharp	1149
lenadroid/fsharp-micro	1148
ehdr/known	1032
centurylink/drupal	947
kreisys/cassandra	712
circleci/buildpack-deps	710
terbiumlabs/pypy-flower	692
singli/rocket.chat	689
afoard/rocket.chat	686

Table 5. Topmost vulnerable community images

Clair also classifies vulnerabilities as per their severity ratings. It provides five types of security rankings: “Negligible”, “Low”, “Medium”, “High” and “Critical”. However, in our study we have selected a more standard convention of “Low”, “Medium” and “High” as per the NVD severity rankings. We extract these labels from the html report generated by Clair.

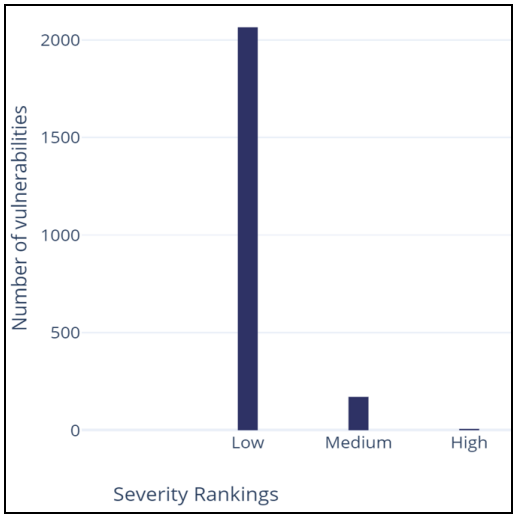

Figure 6. Vulnerability composition in official images

As observed in Figure 6 and 7 we see that the vulnerability composition in both official and community images follow a similar pattern where there is significantly higher number of vulnerabilities with rating as Low. This is followed by medium and high vulnerabilities.

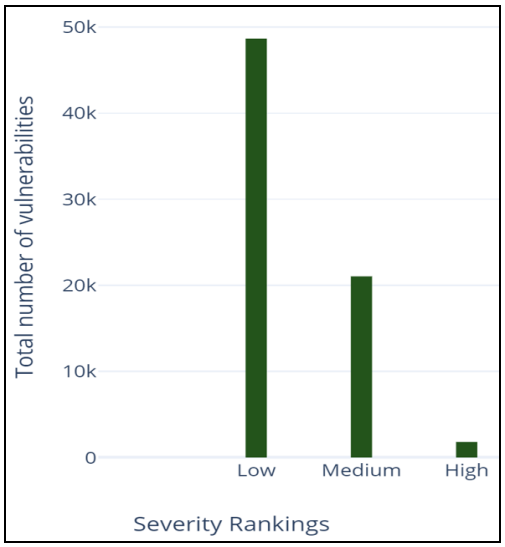

Figure 7. Vulnerability composition in community images

However, if we make a comparison against the average number of vulnerabilities per image. Our statistics indicate that the composition of vulnerabilities in a community image is almost twice of that of an official image. This is depicted in Figure 8.

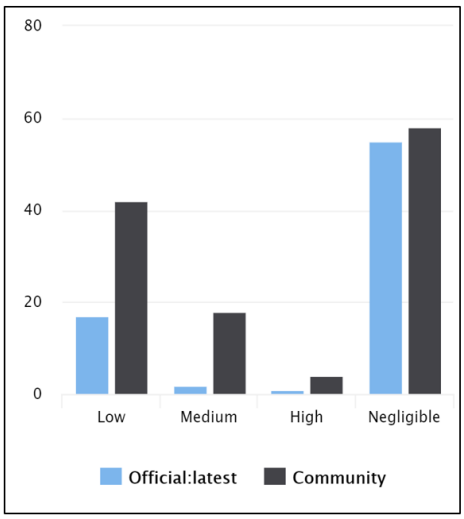

Figure 8. Average number of vulnerabilities per image

We reason that this is because of the fact that official images are well maintained where vendors update or patch their images regularly. On the other hand, we observed that unlike official images, most of the community images are not updated regularly and likely to contain higher number of vulnerabilities. Therefore, image age is another factor that we seek to analyse in our study. Age of an image layer can be easily calculated by subtracting the last update

timestamp of an image from the time at which we are analysing that image.

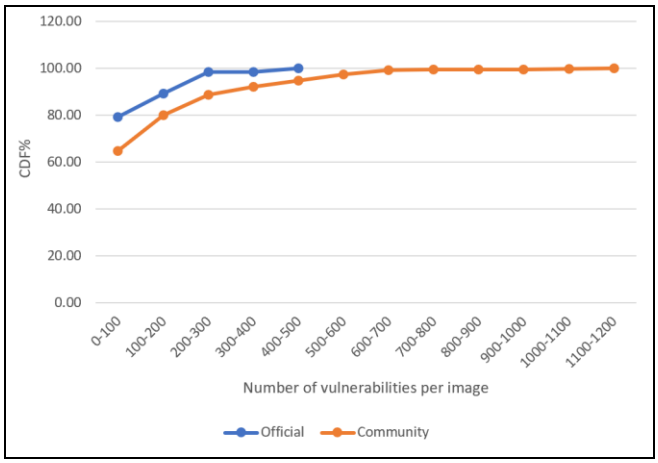

Figure 9. CDF of the number of vulnerabilites per image

Figure 9 shows the CDF of the number of vulnerabilities per image both in official images and community images. It is observed that all the 100 percent of official images have less than 400 vulnerabilities on average but this is not true of community based images where the number of vulnerabilities go as high as 1000 which is more than twice as that of official images. This further collerates the fact that community images are not well maintained and patched as official images which come directly from the vendor. We also suspect that users build community images separate from official image to serve their specific purpose i.e. a user would buid an image that contains additional packages specific to his requirements that are not provided in vendor based images. This is also one of the reasons that we suspect may cause higher number of vulnerabilities in community images.

To prove our hypothesis we see analyse the couchbase official image and in total we found 60 vulnerabilities. We did the same analysis with community based image named couchbase/server and found 62 vulnerabilities. The two additional vulnerabilities existed in package named ‘E2fsprogs’ which is a set of utilities for maintaining the ex2, ex3 and ext4 file systems in Linux.

Similarly, additional packages specific to the user requirements were installed in the community based images which were not present in the official image.

CVE ID	Severity	Number of affected images	Type
CVE-2017-11164	High	81	Uncontrolled recursion
CVE-2013-4235	Medium	81	Race condition
CVE-2019-9192	High	81	Uncontrolled recursion
CVE-2018-20796	High	81	Uncontrolled recursion
CVE-2017-7245	High	81	Buffer overflow

Table 6. Top 5 most frequent vulnerabilities in official images

CVE ID	Severity	Number of affected images	Type
CVE-2013-4235	Medium	637	Race condition
CVE-2019-9192	High	637	Uncontrolled recursion
CVE-2017-11164	High	626	Uncontrolled recursion
CVE-2016-2781	Medium	598	Obtain information
CVE-2018-7169	Medium	588	Obtain information

Table 7. Top 5 most frequent vulnerabilities in community images

Table 6. and Table 7 show the top five vulnerabilities that exist in official and community-based images respectively.

VULNERABILITY PROPAGATION

We observe that vulnerability propagation in docker registry happens due to image dependency and inheritance. If new images are created from the base image containing vulnerabilities, then those vulnerabilities are directly transferred to child image as well.
This is the root cause of vulnerability propagation in the docker registry. We can visualize this with the help of an image inheritance tree.

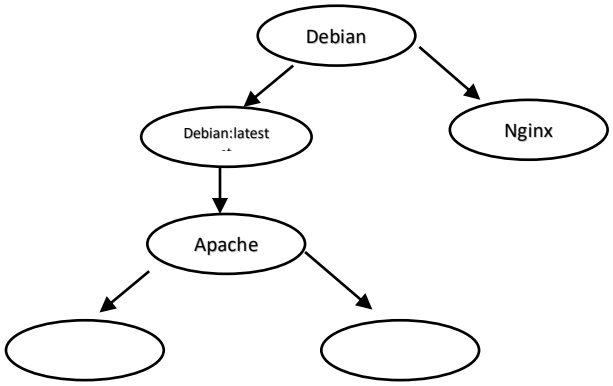

Figure 10. Image inheritance tree

For example, as we can see from Figure 10. Debian is the base operating system image, two images 'Debian: latest' and Nginx are inherited from that image. When images are inherited all packages and dependencies are transferred from the base image and so the vulnerabilities in the original packages as well.

Similarly, Apache is built on top of 'Debian:latest', therefore, if any of the vulnerabilities in Debian is unpatched then it will be propagated to all of the child images as well. Since community images are created by users and those images are not updated frequently, their inherited vulnerabilities are not fixed.

VULNERABLE VS MALICIOUS

Vulnerable is the state of being exposed and vulnerability is a possible weakness in the software, packages and libraries installed on the container. On the other hand, images become malicious when they run an exploit code that can potentially harm the host machine. This could be a potential malware, trojan, or any other payload injected by the attacker.

One of our major observations is that although attackers can exploit vulnerabilities, but malicious nature of images is alone not linked to vulnerabilities. In other words, we found that attackers can easily poison a non-vulnerable image by pushing an exploit code while building the Dockerfile for that image. Attackers can cloak configuration filenames to make it look like a perfectly normal Dockerfile. Therefore, careful inspection of Dockerfile and images should be done while pulling the images from untrustworthy repositories. We prove this by performing a proof-of-concept exploit on a non-vulnerable Nginx server image. We build a Nginx server image on top of 'Debian:buster-slim' operating system which is very minimal distribution of Debian and contained almost negligible vulnerabilities. Once our Nginx server was up and running, we rebuilt the image by injecting a malicious python payload that would steal the host ip address, Nginx configuration files and mail the attacker via an SMTP server.

```
FROM debian:buster-slim
LABEL maintainer="NGINX Docker Maintainers <docker-maint@nginx.com>"

RUN apt-get install -y nginx \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
    && echo "daemon off;" >> /etc/nginx/nginx.conf \
    && apt-get update

EXPOSE 80
CMD ./nginx_start.sh
```

Figure 11. Non-malicious Nginx image

```
FROM debian:buster-slim
LABEL maintainer="NGINX Docker Maintainers <docker-maint@nginx.com>"

RUN apt-get install -y nginx \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
    && echo "daemon off;" >> /etc/nginx/nginx.conf \
    && apt-get update

EXPOSE 80
RUN mkdir -p /home/ubuntu/se
COPY nginx_conf.py nginx_start.sh /home/ubuntu/se/
WORKDIR /home/ubuntu/se/
RUN chmod +x nginx_start.sh
CMD ./nginx_start.sh
```

Figure 12. Malicious Nginx image

Figure 11 and Figure 12 shows the original and modified Dockerfile. As we can see from the images above in our malicious Dockerfile we have injected an example payload file 'nginx_conf.py' in the container's home directory. This payload gets executed when the user deploys the image. The nginx server starts on the container network, however the payload when executed in the background steals the ip address of the host machine, the open ports, and the server's configuration files which then sends the details to the attacker via an SMTP server call which gets executed in the background. This way the attacker gets the sufficient information for initiating a DDoS attack or even hijacking the user's applications running on the container.

This is just a small example to show that how non vulnerable poisoned images can be malicious, and hence vulnerability is not a sufficient condition for malicious behavior in images.

MALICIOUS IMAGE DETECTION

Currently, there does not exist a technique where we can detect malicious images without dynamically running the container and observing its behavior, nor the docker hub registry provides a security rating to the users. Although, it does provide user reviews for the image and also provides external service called Docker Bench for Security which is a script that checks for dozens of common-best practices around deploying containers in production. However, we feel these security features are not adequate enough to rule out the threat of malicious images without actually deploying the container images. Therefore, we hypothesize that there are certain features that we have extracted that could provide a possible indication of a malicious container image.

Features per image
Total vulnerabilities count
High vulnerability count
Medium vulnerability count
Low vulnerability count
Image size
Pull count
Star count
Image age
Image category
Owner's contribution
Image layer count
Number of Tags

Table 8. Features for malicious image detection

Vulnerability count: Indicates the total number of vulnerabilities in the image. A highly vulnerable image is more likely to be malicious.
Severity rating of the image: This signifies the number of low, medium and high vulnerabilities.
Image size: Higher the image size more likely it is for the image to contain higher number vulnerabilities due to more number of packages installed.
Pull count: Number of times the images have been pulled. A non-malicious image coming from the vendor would have a trust factor and would contain high number of pulls. On the other hand, a malicious image pushed by an attacker would contain very less pulls.
Start count: Docker Hub gives star count to the image repositories available on the registry. The star rating is calculated by the number of stars given to the image by the users. We hypothesize that malicious images would certainly have a very poor star count or negative reviews.
Image age: Image age is calculated by subtracting the last update time stamp of the image from the time of analysis. Our hypothesis is that a malicious image will have a very few update count as there is no reason for an attacker to update the image once a malicious payload has been injected. Very few updates would also make the image more vulnerable. An image with few updates will also naturally have less number of image layers.
Image category: This defines whether the image is an official, community or verified (docker certified). Only community images can be suspected as malicious.
Owner's contribution: This denotes the contribution of owner on the docker hub. An attacker pushing a malicious image would have an insignificant contribution, the number of images in his account would be low and there is a high chance that his account would be newly created.
Layer count: Layer count, along with Tags will give us the idea the number of times the image is updated and the number of times the new version of the image has been pushed in the registry.

We believe that the above factors would be a strong indicator to classify an image as malicious or not, irrespective of the number of vulnerabilities existing in the image. A decision tree classifier would be an ideal algorithm for this problem. Since, we do not yet have the data of malicious images, this problem at the moment remains unsolved. Perhaps, future work in extracting the data about the malicious images from registries outside Docker Hub and then checking whether our factors hold true to the data would deem beneficial. This is a difficult in a sense that attackers would frequently delete their images or have their accounts blocked once their images have been identified as malicious. So, the data would only be available with the registry owners i.e. Docker itself.

RELATED WORK

There have been previous studies in the area where researchers have analyzed Docker Hub images using third party tools. One of the many studies has been conducted by Shu et al where they built a multiprocessing framework called DIVA for image vulnerability analysis and show vulnerability propagation using Image dependency graph. As their study is old, our observations are quite different than theirs. Their study shows that around 85 percent of

both official and community images contain vulnerabilities with a severity rating of high. However, our statistics relate that the average number of low or negligible vulnerabilities are twice than that of high vulnerabilities which shows that most of the high vulnerabilities are being patched or fixed by the vendors. This also shows that vendors give more importance to high vulnerabilities over vulnerabilities which are classified as medium or low.

There have been other studies as well, one of which is by Banyan Collector which facilitates vulnerability detection by deploying image containers and then running script inside the container to collect metadata about the containers. Apart from clair there are multiple tools namely Snyke, Anchore engine for container scanning. Twistlock is another tool that provides container security which is specific to images hosted on IBM's Bluemix cloud.
We believe our study maybe the first one to propose features that could be used in the machine learning classification models for malicious image detection. There also exists an article published by Raman, V. which describes different algorithmic techniques especially generative algorithms, discriminative algorithms, and automata-theoretic algorithms which observe signals from containers, orchestrators, hosts and logs to detect anomalies.

There are other related works as well in the area of virtual machine image security. Bugiel et al. provided a systematic analysis of security on images hosted on Amazon's marketplace where they analyse how attackers can inject malicious payload in Amazon Machine Images (AMI). Public virtual machines images can also be easily modified similar to the containers; therefore, vulnerability propagation may also be similar.

CONCLUSION

In this project we analyzed the official and community-based images on the Docker Hub registry. We developed a sequential framework for data collection and analysis using a third-party tool called clair. We observe that vulnerabilities alone do not lead to malicious behavior in images and identify the relationship between vulnerable image and malicious. We lay importance on the state of security on Docker Hub registry and propose that Docker Hub should provide security rating to the image repositories apart from the reviews and star rating. We also identify certain features of malicious image repositories that could come useful if we are to build a machine learning model for malicious image detection.

Our findings on vulnerability severity and composition on both official and community-based image indicates a need of more automated and intelligent techniques to counter new and upcoming security threats to container technology.

Future work in this area could definitely do a more comprehensive study on registries apart from official ones and a more extensive study on Docker Hub covering most if not all the community-based images. A major contribution in this study could be made if one can get real data on malicious container images and verify if our classification model does really fit with that data.

REFERENCES

[1] Docker Bench for Security. https://github.com/docker/docker-bench-security.

[2] CoreOS Clair. https://github.com/coreos/clair.

[3] http://www.cvedetails.com/cve/

[4] CVE: Common Vulnerabilities and Exposures. https://cve.mitre.org/.

[5] http://www.cvedetails.com/browse-by-date.php/.

[6] National Vulnerability Database. https://nvd.nist.gov/home.cfm.

[7] Banyan Collector. https://github.com/banyanops/collector.

[8] Shu, Rui, Xiaohui Gu, and William Enck. 2017. "A Study of Security Vulnerabilities on Docker Hub." *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy - CODASPY 17.* doi:10.1145/3029806.3029832.

[9] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[10] A. Bettini. Vulnerability exploitation in Docker container environments. https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments-wp.pdf, 2015.

[11] S. Bugiel, S. N¨urnberger, T. P¨oppelmann, A.-R. Sadeghi, and T. Schneider. AmazonIA: When elasticity snaps back. *In Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 389–400, New York, NY, USA, 2011. ACM.

[12] Debian Security Bug Tracker. https://security-tracker.debian.org/tracker.

[13] B. DeHamer. Docker Hub Top 10. https://www.ctl.io/ developers/blog/post/docker-hub-top-10/, August 2015.
[14] Docker Security Scanning. https://docs.docker.com/docker-cloud/builds/image-scan/.
[15] https://thenewstack.io/ml-beyond-algorithmic-detection-security/
[16] IBM's Vulnerability Advisor. http://www-03.ibm. com/press/us/en/pressrelease/47165.wss.
[17] K. Hashizume, N. Yoshioka, and E. B. Fernandez. Three misuse patterns for cloud computing. *Security engineering for Cloud Computing: approaches and Tools*, pages 36–53, 2012.
[18] J. Gummaraju, T. Desikan, and Y. Turner. Over 30% of official images in docker hub contain high priority security vulnerabilities. Technical report, BanyanOps, 2015.