A Study of Security Vulnerabilities on Docker Hub

Rui Shu, Xiaohui Gu and William Enck
North Carolina State University
Raleigh, North Carolina, USA
{rshu, xgu, whenck}@ncsu.edu

ABSTRACT

Docker containers have recently become a popular approach to provision multiple applications over shared physical hosts in a more lightweight fashion than traditional virtual machines. This popularity has led to the creation of the Docker Hub registry, which distributes a large number of official and community images. In this paper, we study the state of security vulnerabilities in Docker Hub images. We create a scalable Docker image vulnerability analysis (DIVA) framework that automatically discovers, downloads, and analyzes both official and community images on Docker Hub. Using our framework, we have studied 356,218 images and made the following findings: (1) both official and community images contain more than 180 vulnerabilities on average when considering all versions; (2) many images have not been updated for hundreds of days; and (3) vulnerabilities commonly propagate from parent images to child images. These findings demonstrate a strong need for more automated and systematic methods of applying security updates to Docker images and our current Docker image analysis framework provides a good foundation for such automatic security update.

Keywords

Docker Images; Security Vulnerabilities; Vulnerability Propagation

1. INTRODUCTION

The container abstraction has become a popular technique for running multiple application services on a single host. Similar to system virtualization, containers provide an isolated runtime environment and easy methods to package and deploy many instances of an application. However, in contrast to system virtualization, containerized applications on the same host share the host operating system kernel and services. Containers wrap system libraries, files, and code that are needed to support the target application. In doing so, containers become significantly more lightweight than system virtualization, leading to its recent popularity.

Docker is one of the most widely used container-based technologies. Docker distributes applications (e.g., Apache, MySQL) in the form of *images*. Each image contains the target application software as well as its supporting libraries and configuration files. As a result, Docker images provide a convenient way to store and deliver applications. New images need not to start from scratch. Rather, a new image can extend existing images, creating a parent-child relationship between images. At the roots of these inheritance trees are a set of base (or root) images that provide bare-bones functionality for a specific platform (e.g., Ubuntu).

A community has been developed around the creation and sharing of Docker images. Docker Hub,[1] introduced in 2014, is a cloud registry service for sharing application images. Images are distributed using *repositories*, which allow versioned image development and maintenance. Repositories can branch off of other repositories. For example, a maintainer can create an image `myimage:v1` in the `myimage` repository by building upon the `ubuntu:16.04` image in `ubuntu` repository. After installing application softwares, the maintainer can tag the working image as `myimage:v2`. Later, after applying some security updates, the image can be tagged `myimage:v3`.

Docker Hub contains two types of public repositories: official and community. Official repositories contain public, certified images from vendors (e.g., Canonical, Oracle, Red Hat, and Docker). In contrast, community repositories can be created by any user or organization. At the time of writing, there were nearly 100 official repositories. While there is no list of community repositories, our study has identified about 100,000 public community repositories.

In January 2015, a Forrester survey [14] of enterprises indicated that security was a top concern when deciding whether to deploy containers. The survey found that of the various security concerns, the Vulnerabilities & Malware concern was the greatest. Therefore, we hypothesize that the complexity of software configuration in Docker Hub images, combined with a large number of images built by various parties, results in a significantly vulnerable landscape. This intuition leads us to the primary research question of this work: *what is the state of security vulnerabilities in Docker Hub images?*

In this paper, we provide an evaluation of security vulnerabilities in both official and community images that are

CODASPY'17, March 22-24, 2017, Scottsdale, AZ, USA
© 2017 ACM. ISBN 978-1-4503-4523-1/17/03...$15.00
DOI: http://dx.doi.org/10.1145/3029806.3029832

[1]https://hub.docker.com/

publicly available on Docker Hub. Particularly, we aim at answering three key research questions:

RQ1 What is the composition of security vulnerabilities in official and community images based on the number and severity of Common Vulnerabilities and Exposures (CVEs) [4]?

RQ2 How much time has lapsed since images were last updated by their repository maintainers?

RQ3 Does creating images based on other images on Docker Hub lead to the propagation of security vulnerabilities, and to what extent?

To answer those questions, we build a framework that automatically discovers, downloads, and analyzes Docker images. With this tool, we analyze over 300,000 image versions from over 85,000 unique image repositories. Our major findings include: (1) both official and community images contain more than 180 vulnerabilities on average when considering all versions, and more than 80% of both official and community images include at least one high severity vulnerability; (2) a large number of both community and official images have not been updated for hundreds of days, but the latest version of official images are better maintained; and (3) vulnerabilities commonly propagate from parent images to child images.

We make the following contributions:

- *We build a scalable Docker Image Vulnerability Analysis (DIVA) system that automatically discovers, downloads, and analyzes images from Docker Hub.* We note that while Docker Hub is searchable, there is no prior enumeration of available community images. Our system supports parallel image analysis and extracts inter-image inheritance relationships among a large number ($> 300,000$) of image versions.
- *To the best of our knowledge, we perform the first systematic study of public community images on Docker Hub.* Our analysis demonstrates the significant need for more automated methods of applying security updates to Docker images.

We are not the first to study vulnerabilities in Docker Hub images. Prior studies have focused on official images on Docker Hub. For example, BanyanOps [24] reported that over 30% of official images include software with high-priority security vulnerabilities. However, the study was limited to official images and a small random sampling of community images. Additionally, Docker Inc. has worked with the Center for Internet Security (CIS) to release a Docker Security Benchmark to recommend best security practices for deploying Docker [5]. In May 2016, Docker Inc. also announced Docker Security Scanning [20] service (formerly called "Project Nautilus") to analyze security risks in Docker images. However, this service is currently limited to official repositories and some private repositories on Docker Hub.

The remainder of this paper proceeds as follows. Section 2 describes DIVA system design. Section 3 describes experimental evaluations. Section 4 discusses our findings. Section 5 focuses on our future work discussion. Section 6 overviews related work. Section 7 concludes.

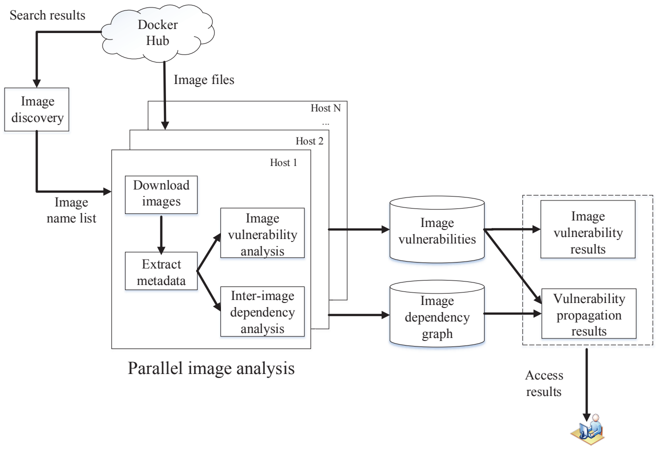


Figure 1: Docker Image Vulnerability Analysis (DIVA) System Framework.

2. DIVA SYSTEM DESIGN

In order to study the broader collection of community and official images on Docker Hub, we must overcome the following challenges:

C1 *There is no public list of community repositories or images on Docker Hub.* While Docker Hub lists around 100 official repositories, community images can only be discovered through keyword-based search.

C2 *The entire registry of Docker Hub images is too large to mirror locally.* While the exact number of images is unknown, our experiments indicate there are hundreds of thousands of images on Docker Hub, and the number continues to grow. The size of images ranges from hundreds of megabytes to several gigabytes. It is impractical to store all images locally before analysis. Thus, our system must support stream-based image analysis, that is, extracting needed information continuously as new images are loaded into the memory and old images are deleted to make space for the new images.

C3 *The number of images prohibits sequential processing.* Our initial experiments indicated an average downloading and processing time of two minutes per image. Therefore, hundreds of thousands of images require tens of months of analysis time. For this reason, our system must support parallel processing to complete the analysis of hundreds of thousands of images within reasonable amount of time.

Figure 1 depicts the architecture of our Docker Image Vulnerability Analysis (DIVA) framework. There are three main components: 1) *the image discovery module* generates random strings to search Docker Hub to identify image names and retrieves images from Docker Hub; 2) *the image vulnerability analysis module* extracts useful metadata and detects vulnerabilities in different images; and 3) *the inter-image dependency analysis* module identifies the inheritance relationships between images.

We now describe these components in detail.

Table 1: Data collected from Docker images.

Data field	Description
Image ID	A 256 bits long ID for each unique image
Image Name	An identifier for each image that follows certain name policy
Last Update Time	Exact date and time of last update to the images
Layer ID	Unique ID of each layer and the relationship between layers
Commands	The history of building the image

2.1 Image Discovery

Our first challenge (**C1**) is to discover Docker Hub repositories and their corresponding images.

Official images are built by using an automated system called bashbrew,[2] which is composed of a set of scripts to clone, build, tag and push official images into Docker Hub. We collect names of official images from the recipes which are available in the docker library in github [32].

There is no public list of community repositories or images on Docker Hub. Instead, Docker Hub provides a case-insensitive, keyword-based search interface to discover repositories [9]. Search strings match repository name, user name, and words in the image description. The search results include: (1) the repository name, (2) a description of the repository, (3) the community rating for the repository in the form of number of stars, (4) whether the repository is official or not, and (5) whether or not the repository is built automatically from github. Each search query to Docker Hub returns at most 25 results.

We discover repository names by creating a dictionary of search keyword strings. Similar to PlayDrone [38], we generate random strings with lengths between 1 and 20 characters[3]. Our resulting dictionary includes 5,000,000 unique strings. The name crawler queries Docker Hub for each string and records the matched repository names. Duplicated names are removed. As we report in Section 3, we discovered 99,843 unique repository names using this method.

Once the repository names are known, we must determine the images within the repositories. For each repository, we perform an additional search to Docker Hub to enumerate all of the tags (e.g., 16.10, latest, trusty). We then combine the repository name with the tag to create the list of image names. Using this method, we discovered 440,524 unique image names. However, between the time of image name discovery and image analysis, a number of repositories and images were not downloadable. We discuss this reduction further in Section 3. Note that our approach discovers both official and community images. We further separate our results into two lists: official image names and community image names based on their image name format (i.e., official image names follow a format of *repo-name:tag* while community image names follow the format of *hub-user/repo-name:tag*).

We note that a Web search engine such as Google could have also been used to discover Docker Hub repository names. For example, the Google search query: *site:hub.docker.com* *"short description" "full description" "official repository"* returns a list of official image repository names. However, when using Google search to identify community image repositories, we were limited by the search results, identifying only a few hundred repositories.

[2]https://github.com/docker-library/official-images#bashbrew

[3]We limit our name string length to 20 because we observe that most of the image names include less than 20 characters. Our framework is generic, which can be configured with longer string length easily.

Table 2: Data collected from Clair.

Data field	Description
Timestamp	Exact time of analysis by Clair
Vulnerability ID	Unique CVE identifier to identify vulnerability
Severity Ranking	Severity of each vulnerability
Description of CVE	Description of each identified vulnerability
Associated Packages	Name and exact version of the package that associates with each vulnerability
Layer ID	Flag the specific image layer where the vulnerability resides

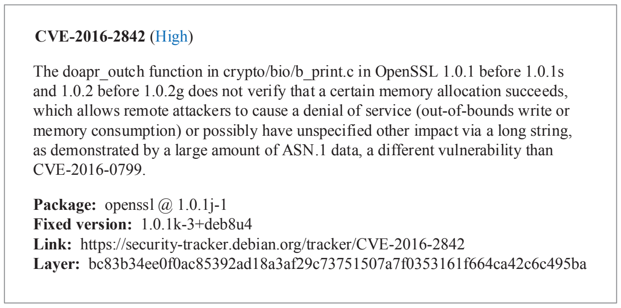
2.2 Image Vulnerability Analysis

Once the image names are identified, we need to download the corresponding image files for analysis. Since it is impractical to download all the images from the Docker Hub to our local hosts, we need to adopt a stream-based parallel image analysis approach. Specifically, each host fetches a set of image names from the name list and downloads those images using the Docker daemon's `docker pull` command (e.g., *"docker pull hub-user/repo-name:tag"* for community images). Next, we perform the image analysis. Once the analysis completes for the image set, all of those images are deleted. We iterate the above process over sets of new images on each host. We can scale up the processing by performing the analysis on a large number of hosts concurrently. We also found that images from the same repository often share common layers and therefore the Docker daemon can avoid pulling a layer again if the layer already exists on the local host. This observation can lead to further speedup by always retrieving the images of the same repository together.

To analyze the security vulnerability of each image, we first extract metadata about each image, such as its name, IDs, and layer information. Specifically, for each downloaded image, we collect five data fields, shown in Table 1. Note that for the last update time, we use `docker inspect` to fetch the details of each docker image and store the results in an array. The creation time is the date of the latest `docker build`, therefore, we use this timestamp to denote the latest update to images.

We then leverage Clair [16] to detect vulnerabilities in each image. Clair is an open-source tool from CoreOS designed to identify known vulnerabilities in container images. Clair has been primarily used to scan images in CoreOS's private container registry, Quay.io, but it can also analyze Docker images.

We collect several types of vulnerability information using Clair, as shown in Table 2. Clair uses static analysis to extract: 1) the version of all installed software packages, and 2) the operating system metadata in each layer of an image. Clair identifies insecure packages by matching the metadata against the Common Vulnerabilities and Exposures (CVE) vulnerability database[4] and similar databases such as Ubuntu CVE Tracker [37], Debian Security Bug Tracker [18], Red Hat Security Data [34], etc. Note that Clair only identifies the presence of packages with known



CVE-2016-2842 (High)

The doapr_outch function in crypto/bio/b_print.c in OpenSSL 1.0.1 before 1.0.1s and 1.0.2 before 1.0.2g does not verify that a certain memory allocation succeeds, which allows remote attackers to cause a denial of service (out-of-bounds write or memory consumption) or possibly have unspecified other impact via a long string, as demonstrated by a large amount of ASN.1 data, a different vulnerability than CVE-2016-0799.

Package: openssl @ 1.0.1j-1
Fixed version: 1.0.1k-3+deb8u4
Link: https://security-tracker.debian.org/tracker/CVE-2016-2842
Layer: bc83b34ee0f0ac85392ad18a3af29c73751507a7f0353161f664ca42c6c495ba

Figure 2: A sample output of Clair for CVE-2016-2842 from image ruby:2.0.0-p594-onbuild.

vulnerabilities. It does not determine if those packages are actually used by container instances. Similarly, it does not detect dynamic behavior in running instances, e.g., installing vulnerable package versions at runtime.

Clair identifies the package versions based on the file system view that is observable at runtime. If the image is built from a Dockerfile, which specifies a set of instructions to produce a local image, Clair is executed on the resulting image. As discussed further in Section 2.3, Docker images are based on layers. Each layer stores copy-on-write information to produce a file system view. For example, we define the base layer to be a scratch image (used before Docker version 1.5.0 [28]) or created from a Dockerfile instruction (e.g., ADD). The layers above the base layer are the results of installing additional packages via installing commands or upgrading commands such as `apt-get install` or `apt-get upgrade`, or operations on existing files (e.g., add, modify, delete) in running containers. In addition, executing instructions specified in Dockerfiles (e.g., ADD, COPY) also creates new layers. Since Clair operates statically, it must process all the layers in one image to identify any vulnerable packages. However, it must take care not to report a vulnerable package in a lower layer if it is superseded by a patched version of the package in a higher layer. We experimentally confirmed that Clair does not report a vulnerable package in a lower layer when a higher layer upgrades the package. For example, we ran Clair on the ubuntu:14.04 image and observed that vim 2:7.4.052-1ubuntu3 is identified as a vulnerable package. We performed an `apt-get upgrade` to upgrade vim to version 2:7.4.052-1ubuntu3.1 and committed the result to a new image. When running Clair on the new image, the vulnerability for the upgraded vim package was no longer present and was not reported by Clair.

Figure 2 shows a sample output from Clair for CVE-2016-2842 from image ruby:2.0.0-p594-onbuild. For each CVE entry, Clair collects the unique CVE identifier with the vulnerability severity rating. In Clair version 1.0 (used for our study) the analysis outputs specific advice for security flaws. In most cases, Clair recommends upgrading specific packages to a more recent version. We also note that CVE identifiers are unique IDs for known security vulnerabilities (e.g., CVE-2016-1977). Red Hat Security Advisories (RHSA) uses a different format of identifier (e.g., RHSA-2016:0176), which must be mapped to CVE identifiers [34, 10]. When Clair identifies a package with a vulnerability, it outputs a URL for the corresponding CVE, along with the layer ID that contains the package.

Each CVE's severity is ranked by the National Vulnerability Database (NVD) [6] using the Common Vulnerability Scoring System (CVSS) [8]. CVSS assigns a severity score based on a formula including exploitability and impact metrics. The NVD also provides a qualitative severity rating of "Low", "Medium" and "High" based on the CVSS score (Low: 0.0-3.9; Medium: 4.0-6.9; High: 7.0-10.0). We use these qualitative scores to report statistics in our study.

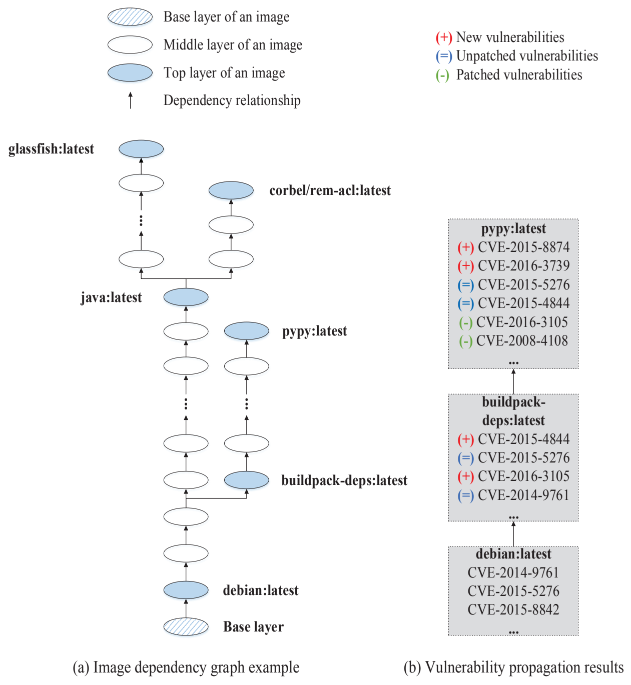


Figure 3: Inter-image dependency analysis example.

2.3 Inter-Image Dependency Analysis

Basing new images on existing images on Docker Hub minimizes effort. However, it also propagates any software vulnerability to the new image, if care is not taken apply security updates. In this section, we design an algorithm to investigate the dependency relationship between images, as well as identify vulnerability propagation patterns (**RQ3**).

Each Docker image is composed of a list of read-only layers. On a Docker host, each layer is stored as a tar file within a unique directory. Layers are stacked hierarchically, the order of which is specified in a JSON configuration file. The configuration file references a layer ID, which is unique throughout Docker Hub. Prior to Docker version 1.10, the layer ID was a randomly generated 256-bit UUID. However, for versions 1.10 and later, the layer ID is the SHA256 hash of the tar file content. Commonly, the first 12 hex characters are used as a short identifier for a layer. Note that our study uses Docker version 1.9.0, which was the stable version during our experiment. There are some differences between these two versions, e.g., the way how images are stored in the host; however, the changes would only require minor modification in the DIVA source code.

To study dependency relationships between images, we represent all layers in all images on Docker Hub using one *directed graph* $G = (V, E)$, where the set of vertices V represents the layer IDs, and the set of edges E represents relationships between layer IDs, as specified in the JSON configuration files of images. We call G the *image dependency graph*. In our representation, we label vertices with the set of image names that have the corresponding layer ID as the topmost layer. We represent G as an adjacency list. We maintain the lists of vertices and edges separately in order to label vertices when they are the topmost layer in an image.

To construct the image dependency graph, we process each image using three key steps: 1) updating the set of vertices V with newly discovered layer IDs, 2) updating the set of edges E with newly discovered edges based on the inter-layer relationships specified in the JSON configuration file (e.g., if the layer l_j is placed on top of the layer l_i in one image, we add an edge $l_i \rightarrow l_j$ in the image dependency graph), and 3) annotating a vertex corresponding to the topmost layer with the image name. Note that if an image only has one layer, no edges are added, but the vertex corresponding to that layer is annotated with the image name. Since multiple images may have the same topmost layer, the vertex annotation is a set. An example graph containing six Docker Hub images is shown in Figure 3 (a), which contains both official images (e.g., debian:latest) and community image (e.g., corbel/rem-acl:latest). In this example, these images share the same base layer.

We use the image dependency graph to determine the propagation of vulnerabilities between images on Docker Hub. To do this, we perform a depth-first search on G and compare the vulnerabilities of each image to its direct children. Let $\mathcal{V}(\cdot)$ be a function that returns the set of CVEs for an image, as reported by Clair (Section 2.2). We can then define the set of new vulnerabilities ($\mathcal{V}^+$), patched vulnerabilities ($\mathcal{V}^-$), and unpatched vulnerabilities ($\mathcal{V}^=$) for each pair of parent and child images (i_p, i_c) as follows:

$$\begin{aligned}\mathcal{V}^+(i_p, i_c) &= \mathcal{V}(i_c) \setminus \mathcal{V}(i_p)\\ \mathcal{V}^-(i_p, i_c) &= \mathcal{V}(i_p) \setminus \mathcal{V}(i_c)\\ \mathcal{V}^=(i_p, i_c) &= \mathcal{V}(i_p) \cap \mathcal{V}(i_c)\end{aligned}$$

Figure 3 (b) shows an the vulnerability propagation for the rightmost branch of the graph.

3. EXPERIMENT

To identify the names of community images, we generated 5,000,000 random strings. During the month of February 2016, we queried Docker Hub for each string. After removing duplicates, the search query process identified 99,843 different repository names, including all 98 official repositories. Querying Docker Hub for repository tags produced a list of 440,524 unique image names, composed of 436,722 community images and 3,802 official images.

We did not start to download and analyze images immediately after we generated the image name list. Instead, we randomly selected a sample of 20,000 images, downloaded and analyzed them to further test and improve our analysis framework between March and April. When we performed our image analysis in late April 2016, not all repositories and images were still available. We found that some repositories were purely deleted by users, and we also detected deletions of tags within repositories. Our final dataset consisted of 86,066 repositories, containing 356,218 images, including 3,802 images from the 98 official repositories.

We performed the image metadata extraction using our university's cloud computing infrastructure called the Virtual Computing Lab (VCL) [7]. We reserved 100 virtual machines, each with 4GB memory and 40GB storage, and configured with Ubuntu version 14.04, Docker version 1.9.0, Clair version 1.0. We dedicated one processing node for the official images. The remaining 99 processing nodes were used to analyze community images. The list of community image was split up into 99 sublists, taking care to ensure that images within the same repository were on the same sublist and processed by the same host to avoid repeated downloading of the same layers shared among different images in the same repository.

As for image vulnerability detection, we ran Clair as a container instance on each virtual machine. The Clair instance uses a PostgreSQL container instance to periodically update local vulnerability database (e.g., Ubuntu vulnerabilities database, Debian vulnerabilities database and Red Hat vulnerabilities database). Both the Clair instance and the PostgreSQL instance kept running and waiting for analysis requests throughout the entire experiment. In the end, we aggregated the raw results from Clair for analysis.

4. RESULTS

We now return to our motivating research questions:

RQ1 What is the composition of security vulnerabilities in official and community images based on the number and severity of CVEs?

RQ2 How much time has lapsed since images were last updated by their repository maintainers?

RQ3 Does creating images based on other images on Docker Hub lead to the propagation of security vulnerabilities, and to what extent?

This section presents our experimental results.

4.1 Vulnerabilities per Image

The number of vulnerabilities per image characterizes the Docker Hub vulnerability landscape. Each Docker Hub repository is a collection of related images. Images refer to repository tags, which are commonly different versions of an application or a distribution. Since older, potentially more vulnerable, images may not ever be updated, it is useful to consider both the vulnerabilities per image, as well as the vulnerabilities in the latest version of that repository. To identify the latest image in a repository, we leverage the Docker Hub convention to use the tag "`:latest`" to indicate the latest version. The `:latest` tag is also automatically assigned if a maintainer does not specify any tag when creating a repository. However, if the user specifies any other tag but the `:latest` tag, the repository does not include the `:latest` tag, which is not included in our results about the latest versions. In our dataset, we found that 10,435 out of 85,968 community repositories and 5 out of 98 official repositories did not have a `:latest` tag.

Table 3 reports the number of vulnerabilities for all versions of images, as well as only the latest images. The table includes the mean, median, max, min, and standard deviation of vulnerabilities for the 352,416 community images and 3,802 official images that we analyzed. Interestingly, the number of vulnerabilities per community image does not significantly differ when considering all images verses latest images. In contrast, there is a significant difference between

Table 3: Number of Vulnerabilities per Image.

Image Type	Total Images	Number of Vulnerabilities				
		Mean	Median	Max	Min	Std. Dev.
Community	352,416	199	158	1,779	0	139
Community `:latest`	75,533	196	153	1,779	0	141
Official	3,802	185	127	791	0	145
Official `:latest`	93	76	76	392	0	59

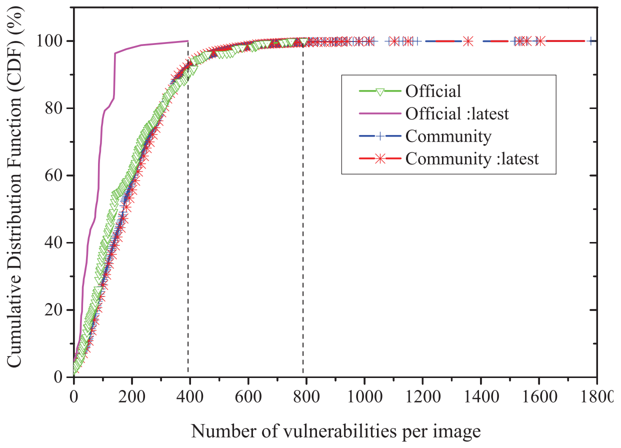


Figure 4: Cumulative distribution function (CDF) of the number of vulnerabilities per image.

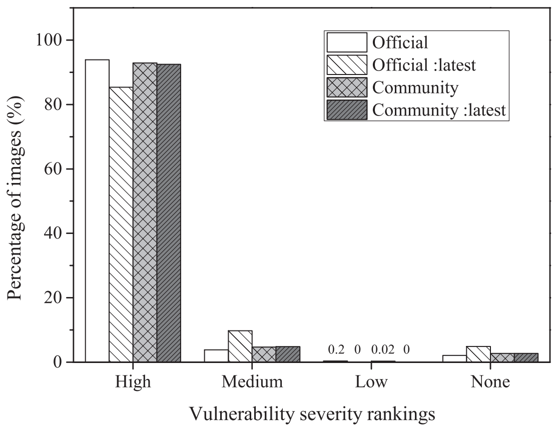


Figure 5: Distribution of images based on most severe vulnerability.

the two classes for official images. This phenomenon is likely the result of better maintenance for official images.

Figure 4 depicts the cumulative distribution function (CDF) for these same images classes. Note that the dashed vertical lines indicate the maximum number of vulnerabilities per image for that class. The CDF corroborates our take-aways from Table 3. The CDF also shows that both classes of community images track the CDF of the vulnerabilities in the class containing all official images. One possible explanation is that many community repositories are based off of old versions of official images, and the maintainers have not applied security updates to the latest image in the repository. We consider vulnerability propagation further in Section 4.5.

4.2 Vulnerability Severity

Clair provides five types of security rankings for vulnerabilities: "Negligible", "Low", "Medium", "High", "Critical". However, we chose to use the more standard NVD severity ranking: "Low", "Medium" and "High". To identify the severity of a vulnerability, we crawled the vulnerability type and CVSS score from the CVE Details database[4] for each CVE vulnerability. We then mapped the score to the NVD ranking based on their thresholds.

Figure 5 categorizes community and official images into four groups: high, medium, low and none. An image is placed in the group corresponding to the highest severity ranking of its most severe CVE. For example, if an image contains at least one "High" severity ranking CVE, it is placed in the "High" group.

This figure shows that even though the latest version of official repositories generally has less vulnerabilities, the vulnerabilities it contains generally include at least one that is high severity. Although it is difficult to determine whether the packages with high severity vulnerabilities are used in running containers, they are still important to address. For example, they may be exploited by attacks that chain together multiple vulnerabilities.

[4]http://www.cvedetails.com/

4.3 Image Age

Many Docker Hub repositories are well maintained, whereas others remain unmaintained. Intuitively, an image that has not been updated in a long time is more likely to contain more vulnerabilities. Therefore, we seek to characterize the age of images at the time of analysis. We determine the age by subtracting the last update timestamp from the time of our analysis for that image. For example, we analyzed the `clojure:lein-2.5.3-onbuild` image on May 17, 2016 and its last update time was March 24, 2016. Therefore, its age is 54 days.

Figure 6 shows the CDF of the age of images at the time of analysis for the four classes of images. As depicted in the figure, for images of all versions, official images are somewhat similar to community images: about 70% of both types of images are updated in less than 400 days at the time of analysis. There is some difference in the percentage of very recently updated images: approximately 20% for all official images verses approximately 10% for all community images. In contrast, nearly 86% of the latest official images are recently updated. This result suggests that official images, particularly the latest official images, are much more frequently maintained on Docker Hub than community images. Finally, we note that the CDF of the latest community images nearly matches the CDF of all community images.

There are several possible explanations for the significant number of images that have not received updates for a long time. For example, some images may deliberately not be updated in order to reproduce bugs in specific experimental environments. Another explanation is that image maintain-

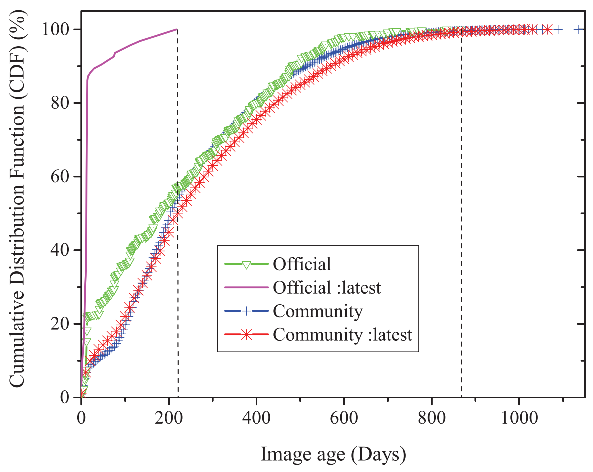


Figure 6: Cumulative distribution function (CDF) of percentage distribution of the age of images at the time of analysis.

ers do not update images to ensure software compatibility. Finally, images not marked as `:latest` may be intentionally unmaintained to provide snapshots of runtime environments.

4.4 Vulnerability Composition

Thousands of new vulnerabilities are discovered each year. In this subsection we consider the composition of security vulnerabilities that exist in Docker Hub images. We first look at the composition of unique vulnerabilities. Next we consider the composition of vulnerability types. Finally, we report the packages contributing to the most vulnerabilities.

Number of Unique Vulnerabilities: Figure 7 compares the total number of CVE vulnerabilities discovered between 2008 to 2015 [1] to the corresponding CVEs that exist in our dataset of Docker Hub images. The figure shows that the number of CVEs per year remained approximately the same between 2008 and 2013, with a steep increase in 2014, and then a decrease in 2015. In contrast, the CVEs found in our dataset of images grows steadily. We found 6,845 unique CVE vulnerabilities in the set of all community images and 1,554 unique CVE vulnerabilities in the set of all official images from the year 2008 to 2015. Since our dataset reports vulnerabilities from the images state in 2016, this trend is to be expected, as some, but not all images are patched over time. However, Docker Hub was not published until 2014, and the existence of CVEs from prior years suggests that some images have included very old software packages.

Types of Vulnerabilities: The CVE Details database taxonmizes CVEs into several vulnerability types. Most of CVE vulnerabilities are associated with one or more vulnerability types. For example, CVE-2015-1781 [2], which is a buffer overflow vulnerability that can be exploited in DNS services and causes denial of service or arbitrary code execution, can fall into three types: denial of service, execute code, and overflow. However, some CVE vulnerabilities are not categorized with any type, e.g., CVE-2015-4000 [3] (a Logjam vulnerability that allows a man-in-the-middle attacker to downgrade the cipher suites used for TLS connections). Furthermore, a small portion of the CVEs in our dataset belong to reserved CVE entries, which are not included in the CVE Details database. On the whole, we were able to categorize 5,116 of 6,845 unique CVEs for community images and 1,069 of 1,554 unique CVEs for official images.

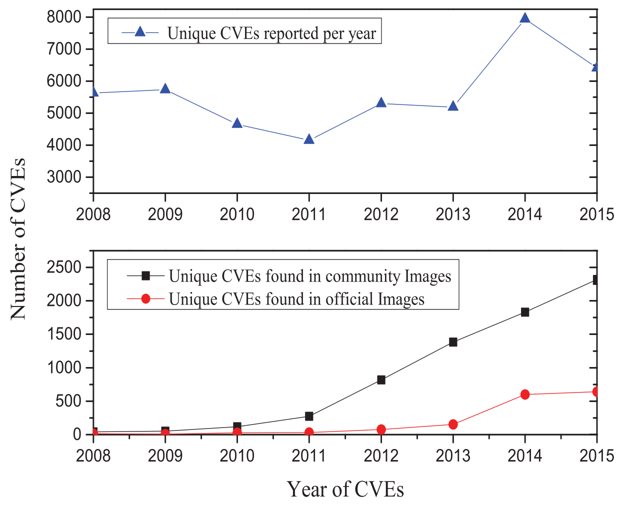


Figure 7: Comparison between CVEs discovered in CVE database and CVEs found in community images and official images from 2008 to 2015.

Tables 4 and 5 show the prevalence of CVE types in the latest version of official and community images. We focus on the latest version, because these images are most likely to represent the most recent version offered by the maintainers. The tables report vulnerability type ranked by the number of images that contain at least one vulnerability of that type discovered in that year. For example, Table 4 shows that 66 of the 93 official images contains an overflow vulnerability from 2010 in its *latest* version. Specifically, this high prevalence of overflow vulnerabilities from 2010 is caused by 2 unique CVEs (i.e., CVE-2010-3192, CVE-2010-4051) found in 2 packages (i.e., eglibc, glibc). The most significant vulnerability was CVE-2010-4051, which was related to a "RE_DUP_MAX overflow", which can lead to denial of service. This vulnerability can be exploited in some applications, e.g., ProFTPD. Finally, comparing official images (Table 4) to community images (Table 5), we see that trends are fairly similar, but community images have more variety in vulnerabilities. One explanation is that the number of studied community images is much larger than the number of official images.

We also observe that a significant portion of the latest community images are impacted by vulnerabilities from 2012 and 2013. However, the latest official images are not. This phenomenon correlates with our previous finding for image age, since a large number of community images, even of the latest version, are not as well-maintained as official images. For example, CVEs from some previous years do not receive enough attention.

Most Vulnerable Packages: Finally, we investigate which packages most frequently cause Docker images to contain

Table 4: Vulnerability types ranked per year by the number of impacted `:latest` official images.

Vulnerability Type	Rank (Number of impacted images)						
	2015	2014	2013	2012	2011	2010	2009
Overflow	1 (78)	1 (75)	3 (14)	5 (5)	2 (2)	1 (66)	1 (14)
Denial of service	2 (77)	1 (75)	1 (56)	1 (44)	2 (2)	1 (66)	4 (1)
Obtain information	2 (77)	7 (6)	5 (12)	6 (0)	5 (0)	4 (30)	5 (0)
Bypass a restriction or similar	4 (57)	4 (40)	6 (1)	2 (28)	1 (3)	1 (66)	2 (2)
Execute code	5 (56)	1 (75)	2 (34)	3 (22)	5 (0)	6 (0)	2 (2)
Gain privileges	6 (33)	10 (0)	6 (1)	4 (15)	5 (0)	6 (0)	5 (0)
Memory corruption	7 (4)	6 (7)	4 (4)	6 (0)	4 (1)	6 (0)	5 (0)
Cross site scripting	8 (2)	8 (4)	6 (1)	6 (0)	5 (0)	6 (0)	5 (0)
Directory traversal	9 (1)	5 (8)	6 (1)	6 (0)	5 (0)	5 (13)	5 (0)
Http response splitting	10 (0)	9 (2)	10 (0)	6 (0)	5 (0)	6 (0)	5 (0)

Table 5: Vulnerability types ranked per year by the number of impacted `:latest` community images.

Vulnerability Type	Rank (Number of impacted images)						
	2015	2014	2013	2012	2011	2010	2009
Denial of service	1 (60k)	1 (60k)	1 (54k)	1 (39k)	1 (5k)	1 (30k)	3 (2k)
Overflow	2 (60k)	2 (59k)	3 (38k)	5 (6k)	4 (3k)	2 (26k)	1 (7k)
Obtain information	3 (59k)	7 (23k)	4 (36k)	6 (4k)	8 (174)	4 (17k)	7 (2)
Bypass a restriction or similar	4 (58k)	4 (49k)	5 (15k)	3 (20k)	3 (3k)	3 (26k)	5 (277)
Execute code	5 (58k)	3 (59k)	2 (47k)	2 (20k)	2 (3k)	6 (1k)	2 (2k)
Gain privilege	6 (52k)	9 (5k)	8 (942)	4 (11k)	7 (255)	7 (94)	9 (0)
Memory corruption	7 (31k)	5 (40k)	6 (5k)	7 (871)	5 (2k)	9 (6)	6 (10)
Cross site scripting	8 (7k)	10 (4k)	7 (980)	8 (198)	6 (387)	8 (88)	4 (486)
Directory traversal	9 (4k)	6 (35k)	11 (69)	10 (94)	10 (4)	5 (14k)	9 (0)
Cross site request forgery	10 (2k)	11 (276)	9 (644)	12 (54)	10 (4)	10 (0)	9 (0)
Http response splitting	11 (466)	8 (9k)	12 (0)	11 (67)	9 (58)	10 (0)	9 (0)
Sql injection	12 (16)	12 (42)	10 (218)	9 (158)	10 (4)	10 (0)	8 (1)

vulnerabilities. Recall from Section 2.2 that Clair reports the vulnerable package name. Table 6 shows the top-ten packages for both community images (all and latest) and official images (all and latest). Note that the statistics are calculated across all versions of the package. For official images, `glibc` is the most frequent offender, affecting over 80% images in both all versions and the latest version. The `glibc` package is also the most significant offender for community images. Another observation is that some packages (e.g., util-linux, shadow, perl, openssl, etc.) appear in each category. Therefore, it is possible that a small number of vulnerable packages cause a significant impact on Docker Hub. These packages could be targeted specifically to improve the security of the Docker Hub ecosystem.

4.5 Image Dependency Relationship

Our third research question seeks to understand the relationship between image dependencies and vulnerability propagation. Child images can be created from both official and community images. There are two general ways to build child images from parent images. First, if a user updates a running image that was downloaded from Docker Hub, that image can be committed as a new image. Second, a Docker Hub repository maintainer can specify a *FROM* instruction in the Dockerfile of a new image. This instruction specifies the base image, which Docker automatically downloads to the Docker host when building the new image from the Dockerfile. Both of the methods may lead to vulnerability propagation. We study this relationship from two perspectives: (1) the degree of propagation from parent image to child image, and (2) the factors that promote propagation.

RQ3.1: *To what degree do child images add, inherit, or remove vulnerabilities?* In Section 2.3 we described an algorithm of identifying the CVEs relationships between a parent and child image. Figure 8 shows the average number of new, unpatched, and patched CVEs per edge between images in the dependency graph. Further, we distinguish between the types of inheritance: official to official, official to community, and community to community. The figure shows that child images inherit on average 80 or more vulnerabilities from their parents, regardless if the parent is official or community. Furthermore, child images frequently introduce new vulnerabilities. This is an interesting observation, because it suggests that when a child installs new software packages, the maintainer is not applying security updates (e.g., with `apt-get upgrade`). That said, Figure 8 does indicate the vulnerability propagation is slightly better for child images that are created from official images.

RQ3.2: *How does image popularity promote vulnerability propagation?* We answer this question in three stages. First, we identify the top most influential OS and non-OS base images, as determined by the number of descendant images. Tables 7 and 8 list the top 10 OS and non-OS base images along with the number of descendant images. Our results for top OS base images is consistent with an August 2015 study by CenturyLink [19]. Second, we look at the distribution of influential base images (Figure 9), we see that there are a relatively small number of very influential images. Finally, we correlate top ranked images with top vulnerable packages.

Tables 7 and 8 list the top vulnerable packages (from Table 6) for the top OS and non-OS base images. The tables show that many of the top vulnerable packages appear in the top influential base images. Thus, *it is highly likely that the root cause of pervasive vulnerabilities on Docker Hub is the result of propagation from a relatively small set of highly influential base images.* As such, future work should investigate methods of automatically pushing updates based on the dependency graph.

Table 6: Top ten packages causing images to contain vulnerabilities.

Rank	Package name (Percentage of impacted images)			
	Official	Official `:latest`	Community	Community `:latest`
1	glibc (89.81%)	glibc (81.91%)	glibc (84.24%)	glibc (84.82%)
2	util-linux (89.55%)	util-linux (81.91%)	openssl (78.32%)	openssl (78.51%)
3	shadow (89.55%)	shadow (81.91%)	util-linux (77.01%)	util-linux (77.24%)
4	perl (87.29%)	audit (77.66%)	shadow (77.01%)	shadow (77.24%)
5	apt (83.82%)	perl (73.40%)	perl (74.07%)	perl (73.05%)
6	openssl (83.79%)	tar (72.34%)	pam (70.92%)	pam (70.53%)
7	tar (83.58%)	apt (70.21%)	pcre3 (66.54%)	audit (67.10%)
8	openldap (76.85%)	openssl (67.02%)	audit (65.48%)	pcre3 (65.59%)
9	krb5 (76.06%)	systemd (67.02%)	krb5 (64.99%)	dpkg (64.36%)
10	audit (73.51%)	gcc (65.96%)	libidn (64.54%)	libidn (62.93%)

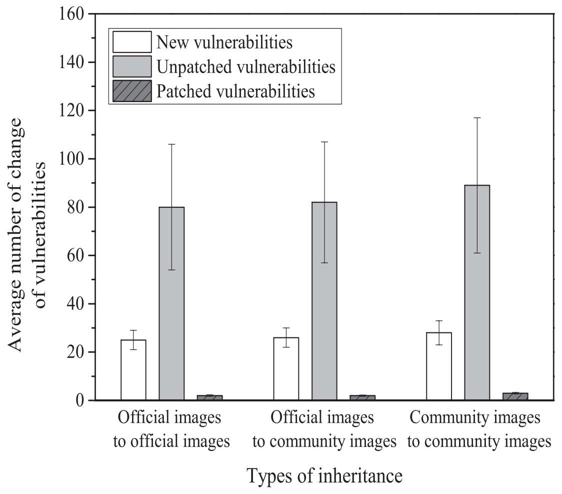


Figure 8: Statistics of the pattern of CVE propagation.

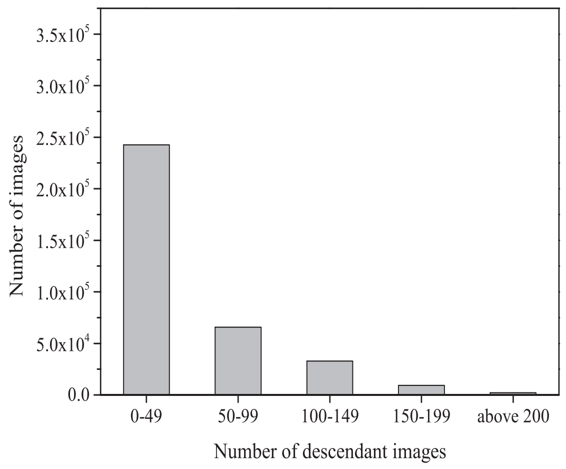


Figure 9: Distribution of the number of descendant images.

4.6 Summary

Our experimental study reveals a set of key findings about the security vulnerabilities of Docker Hub:

1. Both official and community images contain more than 180 vulnerabilities on average when considering all versions. Although the latest official images contain fewer vulnerabilities, the average number of vulnerabilities per image still reach more than 70. In contrast, the number of vulnerabilities contained in the latest community images shows little difference from that of all community images. In addition, more than 80% of both types of images have at least one high severity level vulnerability.

2. About 50% of both community and official images have not been updated in 200 days, and about 30% of images have not been updated in 400 days. There is some difference in the percentage of more frequently updated images (i.e., updated in 14 days) between official images and community images: approximately 20% for all official images verses approximately 10% for all community images. In contrast, nearly 86% of the *latest* official images have been updated in less than 14 days.

3. Child images bring in about 20 more new vulnerabilities on average, and they also inherit 80 vulnerabilities on average from their parent images. The vulnerability propagation is slightly better when child images are created from official images. In addition, there are a relatively small number of influential base images, and we also find top vulnerable packages mostly appear in all top influential base images.

5. FUTURE WORK DISCUSSION

First, our current architecture depends on Clair to statically identify vulnerabilities from installed packages. One possible enhancement for our work is to dynamically scan independent packages that are being installed in the running containers. As a result, we can achieve most timely detection of vulnerabilities introduced by the package update to running docker containers.

Second, we hope to patch the running containers when a vulnerability is detected. One possible approach is to upgrade packages to secure version in running containers, e.g., with `apt-get upgrade`. However, creating containers from images and committing patched containers into images incur resource overhead (e.g., CPU, disk) to the hosts. Moreover, applications or containers might require rebooting after patching, which would incur undesirable unavailability for server applications (e.g., a production web server). Therefore, it is challenging to develop an effective and practical security patching solution, which is also part of our future work.

Table 7: Top ten referenced OS base images. (✓: A package is included in the image; ✗: A package is not included in the image. *: These vulnerable packages appear in Table 6 in both all versions and the latest version of official images.)

Rank	Image name	Number of descendant images	Vulnerable packages (*)											
			glibc	util-linux	shadow	perl	apt	openssl	tar	openldap	krb5	audit	systemd	gcc
1	ubuntu:trusty-20150528	11440	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✓
2	ubuntu:trusty-20151001	10820	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
3	ubuntu:trusty-20150630	8781	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
4	debian:8.3	6642	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	✗
5	ubuntu:trusty-20151028	5862	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
6	ubuntu:trusty-20150730	4912	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
7	ubuntu:trusty-20160217	4755	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
9	ubuntu:trusty-20151218	4497	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
10	ubuntu:14.04.2	3328	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✓

Table 8: Top ten referenced non-OS base images. (✓: A package is included in the image; ✗: A package is not included in the image. *: These vulnerable packages appear in Table 6 in both all versions and the latest version of official images.)

Rank	Image name	Number of descendant images	Vulnerable packages (*)											
			glibc	util-linux	shadow	perl	apt	openssl	tar	openldap	krb5	audit	systemd	gcc
1	node:5.3	3935	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
2	ruby:2.2.4-alpine	3279	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
3	buildpack-deps: jessie-curl	3149	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
4	node:4.2.2-onbuild	2972	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
5	nginx:1.9.7	2887	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓
6	golang:1.5.2-alpine	2749	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
7	node:5.2	2691	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
8	node:4.2.3-onbuild	2597	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
9	nginx:1.9	2551	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓
10	node:5.1.1-onbuild	2544	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Third, the size of Docker Hub community continues to grow at a rapid pace, and more vulnerabilities are being discovered in the meantime. We encourage the participation from image publishers, image users, and repository maintainers to improve the whole ecosystem. For instance, image publishers and maintainers could eliminate security vulnerability risks by utilizing vulnerability assessing tools during image pushing, sharing, and maintaining. Image users should check security threats before running an image downloaded from Docker Hub repositories.

6. RELATED WORK

Docker vulnerability assessment: The first area of related work includes recent efforts in auditing and assessing the security of Docker. For example, Docker's Benchmark for Security [5] assesses the deployment environment and suggests best practices. However, many suggestions are general best practices for Linux. In May 2016, Docker Inc. announced the Docker Security Scanning service [20], formerly known as "Project Nautilus", which provides automated security analysis, validation and continuous monitoring for binary images that hosted on Docker Hub. Images are scanned before every push to Docker Hub, and users are notified when vulnerabilities are discovered. Unfortunately, this service is currently only available to Docker Cloud private repository customers.

There are also several analysis approaches providing vulnerability detection. Banyan Collector [12] can facilitate analysis by launching image containers and running scripts inside them to collect specific information, e.g., installed packages. OpenSCAP Container Compliance [33] provides multiple tools to assist administrators and auditors with assessment, measurement and enforcement of security baselines. Container Compliance provides vulnerabilities assessment of running containers and images (e.g., Red Hat Docker containers) against Common Vulnerabilities and Exposures (CVE) vulnerability database. Twistlock [36] is a closed-source utility that performs heuristics and dynamic profiling at runtime to identify potential risks. Twistlock runs as a dedicated privileged container on each host and looks at the resources being consumed by a container application, including API processes that are spawned, as well as ports being opened. IBM's Vulnerability Advisor [27] is specific to images hosted on IBM's Bluemix cloud. It monitors images pushed to its registry by inspecting features such as packages, configurations, and opened ports. It then compares installed packages against known vulnerability databases for security issues. Vulnerability Advisor also provides guidance for basic security policies.

Our study is the first systematic study of security vulnerabilities in both official and community images on Docker Hub. Compared to previous vulnerability detection techniques, our scalable framework leverages static analysis that provided by Clair, which enables the analysis of a large number of images in a reasonable time. Our findings reveal not only the security vulnerabilities of each image, but also the propagation of vulnerabilities between images.

Virtual machine image security: The second category of related work includes efforts that study virtual machine images, which in many ways parallel Docker images. For example, Amazon's EC2 platform provides customers with a community repository of pre-built Amazon Machine Images (AMIs). If attackers inject malicious code into images and publish them in public repositories, other users who retrieve these images may be compromised [26, 39, 25, 23, 11, 21]. In other cases, confidential information may accidentally leak due to template image cloning [23]. Bugiel et al. [15] provided a systematic analysis of security and privacy in AMIs on Amazon EC2. Their framework extracts sensitive information that can be used as a backdoors in virtual machines created from vulnerable AMIs.

Public virtual machines images are commonly customizable by consumers. Therefore vulnerabilities may propagate that similar to our findings in Docker. Zhang et al. [40] analyzed the cost and effectiveness of exploiting popular vulnerabilities in IaaS Cloud, and then used game theory to model attacks and defenses. Arun Thomas et al. [35] discussed the problem of *virtual machine image sprawl* or *image sprawl* for short. Simply put, the problem is that since creating or cloning an image is easy, the number of images is continuously growing. As a result, the storage and maintainance will become complicated.

To protect VM images against leaking sensitive data by publishers or running malicious images, Mirage [39] provides a set of management approaches (e.g., image filters, virtual scanners) to remove confidential information or detect malicious images. Similarly, *Nuwa* [41] enables automated offline image patching to reduce security threats.

Finding unpatched code in OS distributions: There are also parallels to the propagation of vulnerable code propagating within software packages themselves. For example, ReDeBug [29] is a scalable syntax-based pattern matching approach for finding unpatched copies in OS-distribution scale code bases. Some other work about the detection of cloned code [17, 13, 22, 30, 31] have applied to security. These works have conceptual similarity to vulnerability extrapolation in images. Both copied code and the reusable Docker images can lead to vulnerability propagation.

7. CONCLUSION

Docker Hub provides a public registry for users to store and share containerized-applications. In this paper, we studied the state of security vulnerabilities in these images. We proposed a scalable Docker Image Vulnerability Analysis (DIVA) framework for automatically discovering, downloading, and analyzing vulnerabilities in images from Docker Hub. DIVA also assesses vulnerability propagation between images. We used DIVA to analyze over 300,000 images and found significant and pervasive vulnerabilities in Docker Hub images. We also found strong correlations between top influential images and top ranked vulnerable packages, which implies that the widespread image vulnerabilities are likely the result of propagation from a small number of influential images. These findings demonstrate a strong need for more automated and systematic methods of applying security updates to Docker images and we believe DIVA provides a good foundation to meet the need with its stream-based Docker image processing framework.

8. ACKNOWLEDGMENTS

This work is supported by the NSA Science of Security Lablet at North Carolina State University, under Contract # H98230-14-C-0139. Any opinions, conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the funding agencies. The authors would also like to thank Adwait Nadkarni, our shepherd Dr. Florian Kelbert, and the anonymous reviewers for their valuable feedback during the writing of this paper.

9. REFERENCES

[1] Browse vulnerabilities by date from CVE Details. http://www.cvedetails.com/browse-by-date.php/.

[2] CVE-2015-1781. http://www.cvedetails.com/cve/CVE-2015-1781/.

[3] CVE-2015-4000. http://www.cvedetails.com/cve/CVE-2015-4000/.

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

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

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

[7] NCSU Virtual Computing Lab. https://vcl.ncsu.edu/.

[8] NVD Common Vulnerability Scoring System. https://nvd.nist.gov/cvss.cfm.

[9] Repositories on Docker Hub. https://docs.docker.com/docker-hub/repos/.

[10] RHSA to CVE and CPE mapping. https://www.redhat.com/security/data/metrics/rhsamapcpe.txt.

[11] M. Almorsy, J. Grundy, I. Müller, et al. An analysis of the cloud computing security problem. In *Proceedings of APSEC 2010 Cloud Workshop, Sydney, Australia, 30th Nov*, 2010.

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

[13] 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.

[14] 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.

[15] S. Bugiel, S. Nürnberger, T. Pöppelmann, 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.

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

[17] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. Xiao: tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 369–378. ACM, 2012.

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

[19] B. DeHamer. Docker Hub Top 10. https://www.ctl.io/developers/blog/post/docker-hub-top-10/, August 2015.

[20] Docker Security Scanning. https://docs.docker.com/docker-cloud/builds/image-scan/.

[21] D. A. Fernandes, L. F. Soares, J. V. Gomes, M. M. Freire, and P. R. Inácio. Security issues in cloud

environments: a survey. *International Journal of Information Security*, 13(2):113–170, 2014.

[22] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *ACM Sigplan Notices*, volume 45, pages 175–190. ACM, 2010.

[23] B. Grobauer, T. Walloschek, and E. Stocker. Understanding cloud computing vulnerabilities. *IEEE Security & Privacy*, 9(2):50–57, 2011.

[24] J. Gummaraju, T. Desikan, and Y. Turner. Over 30% of official images in docker hub contain high priority security vulnerabilities. Technical report, BanyanOps, 2015.

[25] K. Hashizume, D. G. Rosado, E. Fernández-Medina, and E. B. Fernandez. An analysis of security issues for cloud computing. *Journal of Internet Services and Applications*, 4(1):1, 2013.

[26] 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.

[27] IBM's Vulnerability Advisor. http://www-03.ibm.com/press/us/en/pressrelease/47165.wss.

[28] Is FROM scratch the root of all Docker Images? https://www.ctl.io/developers/blog/post/is-from-scratch-the-root-of-all-docker-images/.

[29] J. Jang, A. Agrawal, and D. Brumley. Redebug: finding unpatched code clones in entire os distributions. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 48–62. IEEE, 2012.

[30] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[31] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[32] Library of official images. https://github.com/docker-library/official-images/tree/master/library/.

[33] OpenSCAP Container Compliance. https://github.com/OpenSCAP/container-compliance.

[34] Red Hat Security Data. https://www.redhat.com/security/data/metrics/.

[35] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: using semantic information to combat virtual machine image sprawl. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120. ACM, 2008.

[36] Twistlock. https://www.twistlock.com/product/vulnerabilitymanagement/.

[37] Ubuntu CVE Tracker. https://launchpad.net/ubuntu-cve-tracker.

[38] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 221–233, New York, NY, USA, 2014. ACM.

[39] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 91–96. ACM, 2009.

[40] S. Zhang, X. Zhang, and X. Ou. After we knew it: Empirical study and modeling of cost-effectiveness of exploiting prevalent known vulnerabilities across iaas cloud. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 317–328, New York, NY, USA, 2014. ACM.

[41] W. Zhou, P. Ning, X. Zhang, G. Ammons, R. Wang, and V. Bala. Always up-to-date: scalable offline patching of vm images in a compute cloud. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 377–386. ACM, 2010.