Understanding the Security Risks of Docker Hub

Peiyu Liu[1], Shouling Ji[1,(✉),⋆], Lirong Fu[1], Kangjie Lu[2], Xuhong Zhang[1], Wei-Han Lee[3], Tao Lu[1], Wenzhi Chen[1,(✉),⋆], and Raheem Beyah[4]

[1] Zhejiang University, Hangzhou, China
{liupeiyu,sji,fulirong007,lutao,chenwz}@zju.edu.cn,
xuhongnever@gmail.com
[2] University of Minnesota Twin Cities, Minneapolis, USA
kjlu@umn.edu
[3] IBM Research, Yorktown Heights, USA
wei-han.lee1@ibm.com
[4] Georgia Institute of Technology, Atlanta, USA
rbeyah@ece.gatech.edu

Abstract. Docker has become increasingly popular because it provides efficient containers that are directly run by the host kernel. Docker Hub is one of the most popular Docker image repositories. Millions of images have been downloaded from Docker Hub billions of times. However, in the past several years, a number of high-profile attacks that exploit this key channel of image distribution have been reported. It is still unclear what security risks the new ecosystem brings. In this paper, we reveal, characterize, and understand the security issues with Docker Hub by performing the first large-scale analysis. First, we uncover multiple security-critical aspects of Docker images with an empirical but comprehensive analysis, covering sensitive parameters in run-commands, the executed programs in Docker images, and vulnerabilities in contained software. Second, we conduct a large-scale and in-depth security analysis against Docker images. We collect 2,227,244 Docker images and the associated meta-information from Docker Hub. This dataset enables us to discover many insightful findings. (1) run-commands with sensitive parameters expose disastrous harm to users and the host, such as the leakage of host files and display, and denial-of-service attacks to the host. (2) We uncover 42 malicious images that can cause attacks such as remote code execution and malicious cryptomining. (3) Vulnerability patching of software in Docker images is significantly delayed or even ignored. We believe that our measurement and analysis serves as an important first-step study on the security issues with Docker Hub, which calls for future efforts on the protection of the new Docker ecosystem.

1 Introduction

Docker has become more and more popular because it automates the deployment of applications inside containers by launching Docker images. Docker Hub, one

⋆ Shouling Ji and Wenzhi Chen are co-corresponding authors.

of the most popular Docker image registries, provides a centralized market for users to obtain Docker images released by developers [4, 31, 35]. Docker has been widely used in many security-critical tasks. For instance, Solita uses Docker to handle the various applications and systems associated with their management of the Finnish National Railway Service [9]. Amazon ECS allows users to easily run applications on a managed cluster of Amazon EC2 instances in Docker containers [1]. In addition, millions of Docker images have been downloaded from Docker Hub for billion times by users for data management, website deployment, and other personal or business tasks.

The popularity of Docker Hub however brings many high-profile attacks. For instance, on June 13, 2018, a research institute reported that seventeen malicious Docker images on Docker Hub earned cryptomining criminals $90,000 in 30 days [8]. These images have been downloaded collectively for 5 million times in the past year. The report also explained the danger of utilizing unchecked images on Docker Hub: *"For ordinary users, just pulling a Docker image from the Docker Hub is like pulling arbitrary binary data from somewhere, executing it, and hoping for the best without really knowing what's in it"*. Therefore, a comprehensive and in-depth security study of Docker Hub is demanded to help users understand the potential security risks.

The study of Docker Hub differs from the ones of other ecosystems such as App store and virtual-machine image repositories [28, 32, 20, 30, 15] in the following aspects. (1) Docker images are started through run-commands. They are executed through special instructions called run-commands which are security-critical to the created containers. (2) The structures of Docker images are more complex than traditional applications. A single image may contain a large number of programs, environment variables, and configuration files; it is hard for a traditional analysis to scale to scan all images. (3) Docker images can bring new risks to not only the container itself but also the host because the lightweight virtualization technology leveraged in containers allows the sharing of the kernel. (4) The vulnerability-patching process of Docker images is significantly delayed because the programs are decoupled from the mainstream ones, and developers are less incentivized to update programs in Docker images. All the aforementioned differences require a new study for the security of Docker Hub.

The unique characteristics of Docker Hub call for an urgent study of its new security issues. However, a comprehensive and in-depth study entails overcoming multiple challenges. (1) It is not clear how to analyze the security impacts of various categories of information on Docker Hub. For example, run-commands are security-critical to Docker containers while a method for measuring the security impacts of run-commands is still missing. This requires significant empirical analysis and manual effort. (2) Obtaining and analyzing Docker images and the associated meta-information in a scalable manner is non-trivial. For example, it is difficult to perform a uniform analysis on Docker images, since a Docker image contains a large number of files in a broad range of types (e.g., ELF, JAR, and Shell Scripts).

In this paper, we perform the first security analysis against Docker Hub. Based on the unique characteristics of Docker Hub, we first empirically identify three major security risks, namely sensitive parameters in run-commands, malicious docker images, and unpatched vulnerabilities. We then conduct a large-scale and in-depth study against the three security risks. (1) Run-commands. We carefully analyze the parameters in run-commands to discover sensitive parameters that may pose threats to users. Moreover, we develop multiple new attacks (e.g., obtaining user files in the host and the host display) in Docker images to demonstrate the security risks of sensitive parameters in practice. We also conduct a user study to show that users, in general, are unaware of the risks from sensitive parameters. (2) Malicious executed programs. To study malicious Docker images efficiently, we narrow down our analysis to only the executed programs. We implement a framework to automatically locate, collect, and analyze executed programs. By leveraging this framework, we scan more than 20,000 Docker images to discover malicious executed programs. (3) CVE-assigned vulnerabilities. We provide a definition of the life cycle of vulnerability in Docker images and manually analyze the length of the time window of vulnerabilities in Docker images. To enable the analysis, we collect a large number of images and their meta-information from Docker Hub. Our collected dataset contains all the public information of 2,227,244 images from 975,858 repositories on Docker Hub.

The comprehensive analysis enables us to have multiple insightful findings. First of all, we find that the run-commands with sensitive parameters presented in Docker Hub may introduce serious security risks, including the suffering of denial-of-service attack and the leakage of user files in the host and the host display. Moreover, we observe that each recommended run-command in the repository description contains one sensitive parameter on average. Unfortunately, our user study reveals that users are not aware of the threats from sensitive parameters—they will directly execute run-commands specified by developers without checking and understanding them. Second, our analysis shows that malicious images are hidden among common ones on Docker Hub. Using our analysis framework, we have discovered 42 malicious images. The malicious behaviors include remote execution control and malicious cryptomining. Finally, we observe that the vulnerability patching for the software in Docker images is significantly delayed or even ignored. In particular, almost all the images on Docker Hub suffer from unpatched software vulnerabilities. In extreme cases, a single image may contain up to 7,500 vulnerabilities. In addition, vulnerabilities in the software of Docker images tend to have a much longer life cycle due to the lack of image updates. More critically, we find that the in-Docker vulnerabilities can even cause harms to the host machine through Docker, e.g., crashing the host.

Our analysis and findings reveal that the Docker ecosystem brings new security threats to users, contained software, and the host machine as well. To mitigate these threats, we suggest multiple potential solutions (see Section 7) such as automatically fixing vulnerabilities in images, detecting malicious images in Docker Hub, etc. We have reported all the security issues uncovered in this paper to Docker Hub and they are investigating to confirm these issues.

In summary, our work makes the following contributions.

- We empirically identify three major sources of security risks in Docker Hub, namely sensitive parameters in run-commands, malicious docker images, and unpatched vulnerabilities. We then conduct a large-scale and in-depth study against the three security risks based on all the public information of 2,227,244 images collected from 975,858 repositories on Docker Hub. We have open-sourced this dataset to support reproducibility and motivate future work in Docker security analysis [13].
- We uncover many new security risks on Docker Hub. 1) Sensitive parameters in run-commands can expose disastrous harm to users and the host, such as the leakage of host files and display, and denial-of-service attacks to the host. 2) We uncover 42 malicious images that can cause attacks such as remote code execution and malicious cryptomining. 3) Vulnerability patching of software in Docker images is significantly delayed by 422 days on average.
- Our analysis calls for attention to the security threats posed by the new Docker ecosystem. The threats should be addressed collectively by Docker image registry platforms, image developers, users, and researchers. Moreover, we have reported all the security issues uncovered in this paper to Docker Hub and suggest multiple mitigation approaches.

2 Background and Threat Model

In this section, we provide a brief introduction of Docker Hub and its critical risk resources. Then, we describe the threat model of our security analysis.

2.1 Critical Risk Sources in Docker Hub

Docker Hub is the world's largest registry of container images [5]. Images on Docker Hub are organized into repositories, which can be divided into official repositories and community repositories. For each image in a Docker Hub repository, besides the image itself, meta-information is also available to the users, such as repository description and history, Dockerfile [5], the number of stars and pulls of each repository, and developer information. To perform a risk analysis against Docker Hub, we first need to identify potential risk sources. We empirically identify risk sources based on *which major components control the behaviors of a Docker image*.

Run-command and sensitive parameter. In order to run a Docker container, users need to execute an instruction called run-command. A run-command mainly specifies the image and parameters used to start a container. For instance, a developer may specify a recommended run-command on Docker Hub, such as *"Start container with: docker run --name flaviostutz-opencv2 --privileged -p 2222:22 flaviostutz/opencv-x86"*. For users who have never used the image before, the recommended run-commands can be helpful for deploying their containers. However, it is unclear to what extent users should trust the run-commands posted by the developers, who can publish run-commands without

any obstruction because Docker Hub does not screen these content. In addition, a run-command may contain a variety of parameters that can affect the behavior of the container [6]. Some of these parameters are sensitive since they control the degree of isolation of networks, storage, or other underlying subsystems between a container and its host machine or other containers. For example, when users run an image with the parameter of `--privileged`, the container will get the root access to the host. Clearly, the misuse of run-commands containing sensitive parameters may lead to disastrous consequences on the container as well as the host (see Section 4).

Executed programs. Previous work already shows that a large amount of software in Docker images is redundant [31]. Hence, when analyzing the content of a Docker image, we should focus on the executed programs that are bound up with the security of the image. Based on our empirical analysis, we find that the entry-file (an executable file in Docker images, specified by a configure file or run-commands) is always the first software triggered when a container starts. Besides, the entry-file can automatically trigger other files during execution. Therefore, the executed programs (the entry-file and subsequently triggered files) are key factors that directly affect the safety of a container. Furthermore, in general, it is less common for users to run software other than executed programs [4]. Therefore, in the current study, we choose to analyze executed programs to check malicious images for measurement purposes.

Vulnerabilities in contained software. A Docker image is composed of a large number of software packages, vulnerabilities in these software packages bring critical security risks for the following reasons. Vulnerabilities can be exploited by attackers to cause security impacts such as data leakage. Additionally, Docker software programs are often duplicated from original ones, and Docker developers lack incentives to timely fix vulnerabilities in the duplicated programs. As a result, the security risks with vulnerabilities are elevated in Docker because vulnerabilities take a much longer time to be fixed in Docker images.

2.2 Threat Model

As shown in Figure 1, there are two different categories of threats that Docker Hub may face. (1) Vulnerable images. Developers upload their images and the associated meta-information to Docker Hub, which may contain vulnerabilities. If users download and run the vulnerable images, they are likely to become the targets of attackers who exploit vulnerabilities. Additionally, the run-commands announced by developers may contain sensitive

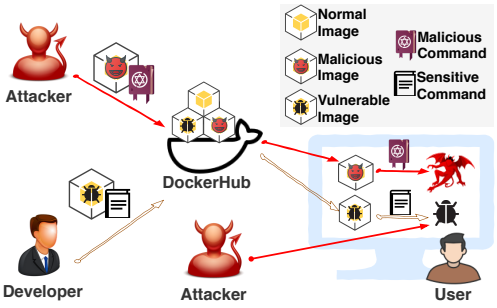


Fig. 1: The threat model of our security analysis.

parameters, which may bring more security concerns such as giving containers root access to the host (see Section 4). (2) Malicious images. Attackers may upload their malicious images to Docker Hub, and sometimes together with malicious run-commands in the description of their images. Due to the weak surveillance of Docker Hub, malicious images and metadata can easily hide themselves among benign ones (see Section 5) [8]. Once users download a malicious image and run it, they may suffer from attacks such as cryptomining. On the other hand, malicious run-commands can also lead to attacks such as host file leakage. In this study, since it is mainly for measurement purposes, we assume that attackers are not aware of the techniques we employ to analyze Docker images. Otherwise, they can hide the malicious code by bypassing the analysis, leading to an arms race.

3 Analysis Framework and Data Collection

In this section, we provide an overview of our analysis framework. Then we introduce our methods of data collection and provide a summary for the collected dataset.

3.1 Overview of Our Analysis Framework

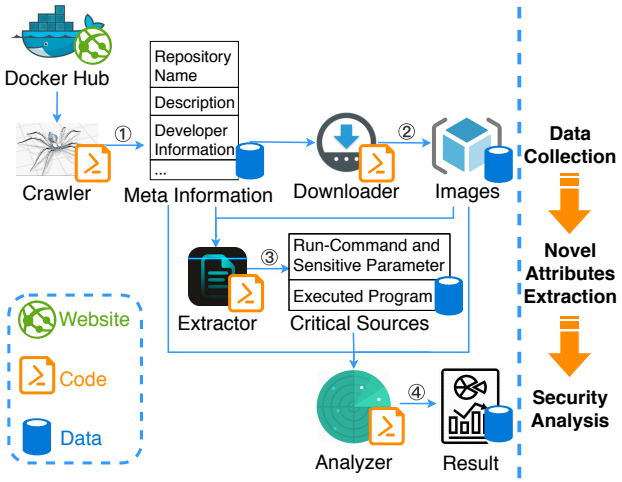


Fig. 2: The framework of our analysis.

In our security analysis, we focus on the key risk sources (run-commands, sensitive parameters, entry-files and vulnerabilities) in Docker Hub, as uncovered in Section 2.1. Our analysis framework for the study is outlined in Figure 2. We first collect a large-scale dataset, including the images and all the public information of a Docker image such as image name, repository description, and developer information, from Docker Hub (①,②). After data collection, the extractor utilizes a set of customized tools to obtain several previously-ignored essentials such as sensitive parameters and executed programs from the raw data (③). After obtaining the set of essentials, we perform systematic security analysis on Docker Hub from various aspects by leveraging `Anchore` [2], VirusTotal intelligence API [11], and a variety of customized tools (④).

3.2 Data Collection and Extraction

The abundant information (described in Section 2.1) of Docker images on Docker Hub is important in understanding the security of Docker Hub. However, most of this information has never been collected before, leading to incomprehensive analysis. Hence, we implement a customized web crawler that leverages Docker Hub API described in [3] to collect the Docker images and their associated meta-information from Docker Hub. All the information we obtain from Docker Hub is publicly available to anyone and it is legal to perform analysis on this dataset.

Summary of Our Collected Data. Our dataset, as shown in Table 1, contains all the public information of the top 975,858 repositories on Docker Hub. For each repository, the dataset contains the image files and the meta-information described in Section 2.1. Furthermore, to support in-depth analysis, we further extract the following data and code from the collected raw data.

Table 1: Data Collected in Our Work.

	No. Repositories	No. Images	No. Developers
Official	147	1,384	1
Community	975,711	2,225,860	349,860
Total	975,858	2,227,244	349,861

Collecting run-command and sensitive parameter. As discussed in Section 2.1, run-commands and sensitive parameters can make a great impact on the behavior of a container. In order to perform security analysis on them, we collect run-commands by extracting text contents that start with ``` ``docker run'' ``` from the repository descriptions and further obtain sensitive parameters from the run-commands through string matching.

Collecting executed program. As discussed in Section 2.1, the executed program is a key factor that directly affects the security of a container. Therefore, we develop an automatic parser to locate and extract the executed program. For each image, our parser first locates the entry-file according to the Dockerfile or the manifests file. Once obtaining the entry-file, the parser scans the entry-file to locate the files triggered by entry-file. Then, the parser scans the triggered file iteratively to extract all executed programs in the image. For now, our parser can analyze ELF files and shell scripts by leveraging `strings` [10] and a customized script interpreter, respectively.

4 Sensitive Parameters

In this section, we (1) identify sensitive parameters; (2) investigate the user awareness of sensitive parameters; (3) propose novel attacks exploiting sensitive parameters; (4) study the distribution of sensitive parameters on Docker Hub.

4.1 Identifying Sensitive Parameters

As described in Section 2.1, the parameters in run-commands can affect the behaviors of containers. However, among the over 100 parameters provided by Docker, it is unknown which parameters can cause security consequences. Therefore, we first obtain all the parameters and their corresponding descriptions from the documentation provided by Docker. Then, we manually identify the sensitive parameters by examining if they satisfy any of the following four proposed criteria: (1) Violate the isolation of file systems; (2) Violate the isolation of networking; (3) Break the separation of processes; (4) Escalate runtime privileges.

Next, we explain why we choose these criteria. (1) From the security perspective, each Docker container maintains its own file system isolated from the host file system. If this isolation is broken, a container can gain access to files on the host, which may lead to the leakage of host data. (2) By default, each Docker container has its own network stack and interfaces. If this isolation is broken, a container can have access to the host's network interfaces for sending/receiving network messages, which, for example, may cause denial of service attacks. (3) Generally, each Docker container has its own `process-tree` separated from the host. If this isolation is broken, a container can see and affect the processes on the host, which may allow containers to spy the defense mechanism on the host. (4) Most potentially dangerous Linux capabilities, such as loading kernel modules, are dropped in Docker containers. If a container obtains these capabilities, it may affect the host. For example, it is able to execute arbitrary hostile code on the host.

4.2 User Awareness of Sensitive Parameters

We find that nearly all the default container isolation and restrictions enforced by Docker can be broken by sensitive parameters in run-commands, such as `--privileged`, `-v`, `--pid`, and so on. We describe the impact of these sensitive parameters in Section 4.3. However, the real security impacts of these sensitive parameters on the users of Docker Hub in practice is not clear. Therefore, the first question we aim to answer is "are users aware of the sensitive parameters when the parameters in run-commands are visible to them?" To answer this question, we conduct a user study to characterize the behaviors of users of Docker Hub, which allows us to understand user preferences and the corresponding risks.

Specifically, we survey 106 users including 68 security researchers and 38 software engineers from both academia and industry fields. For all the 106 users, 97% of them only focus on the functionality of images and have never raised doubts about the descriptions, e.g., the developer identification, the run-commands, on Docker Hub. It is worth noting that, even for 68 users who have a background on security research, 95% of them trust the information provided on Docker Hub. 90% of security experts run an image by exactly following the directions provided by image developers. Only 10% of security experts indicate that they prefer to figure out what the run-commands would do. Indeed, the study can be biased due to issues such as the limited number of the investigated users,

imbalanced gender and age distribution, and so on. However, our user study reveals that users, even the ones with security-research experience, do not realize the threats of sensitive parameters in general. Nearly 90% of users exactly execute run-commands specified by developers without checking and understanding them. Hence, we conclude that sensitive parameters are an overlooked risk source for Docker users. More details of the user study are deferred to Appendix A.1.

4.3 Novel Attacks Exploiting Sensitive Parameters

To demonstrate the security risks of sensitive parameters in practice, we develop a set of new attacks in Docker images that do not contain any malicious software packages. Our attacks rely on only run-commands with sensitive parameters to attack the host. Note that we successfully uploaded these images with our "malicious" run-commands to Docker Hub without any obstruction, confirming that Docker Hub does not carefully screen run-commands. However, to avoid harm to the community, we immediately removed the sensitive parameters in the run-commands after the uploading, and performed the attacks in our local lab machines only.

The leakage of host files. As described in Figure 3, we show how to leak user files in the host using sensitive parameters. Specifically, `--volume` or `-v` is used to mount a volume on the host to the container. If the operator uses parameter, `-v src:dest`, the container will gain access to `src` which is a volume on the host. Exploiting this parameter, attackers can maliciously upload user data saved in the host volume to their online repository, such as GitHub. It is important to note that this attack can be user-insensitive, i.e., the attacker can prepare a configure file in the malicious image to bypass the manual authentication.

```
git clone https://Attacker/data.git
cd data
git pull
cp -r usr-data .
git add .
cur_date=`date`
git commit -m "$cur_date"
git push
```

Fig. 3: Code example to implement the leakage of host files.

Other attacks are delayed to Appendix A.2. Overall, these novel attacks we proposed in this paper demonstrate that sensitive parameters can expose disastrous harm to the container as well as the host.

4.4 Distribution of Sensitive Parameters

Next, we study the distribution of sensitive parameters used in real images on Docker Hub. We observe that 86,204 (8.8%) repositories contain the recommended run-commands in their descriptions on Docker Hub. Moreover, as shown in Table 2, there are 81,294 sensitive parameters in these run-commands—on average, each run-command contains one sensitive parameter. Given the common usage of sensitive parameters and their critical security impacts, it is urgent to improve users' awareness of the potential security risks brought by sensitive parameters and propose effective vetting mechanisms to detect these risks.

Table 2: Distribution of sensitive parameters.

Criteria	No. Sensitive Parameters
Violate the isolation of file systems	33,951
Violate the isolation of networking	43,278
Break the separation of processes	56
Escalate runtime privileges	4,009
Total	81,294

5 Malicious Images

As reported in [8], high-profile attacks seriously damaged the profit of users. These attacks originate from the launching of malicious images such as electronic coin miners. In order to detect malicious images, manual security analysis on each software contained in a Docker image yields accurate results, which is however extremely slow and does not scale well for large and heterogeneous software packages. On the other hand, a dynamic analysis also becomes impractical since it is even more time-consuming to trace system calls, APIs, and network for millions of Docker images. Compared to the two methods above, a static analysis seems to be a potential solution to overcome these challenges. However, the number of software packages contained in each image varies from hundreds to thousands. It is still challenging to analyze billions of software packages using static analysis. Fortunately, we found that a majority of software packages in Docker images are redundant [31] which thus can be filtered out for efficient analysis. However, it is unclear which software deserves attention from security researchers. As stated in Section 2.1, we propose to focus on the executed programs. As long as a malicious executed program is detected, the corresponding image can be confirmed as malicious. Considering that our research goal is to characterize malware for measurement purposes instead of actually detecting them in practice, we focus on the executed programs only rather than other software for discovering malicious images in this study. We will discuss how to further improve the detection in Section 8.

5.1 Malicious Executed Programs

Detecting malicious executed programs. By leveraging the parser proposed in Section 3.2, we can obtain all the executed programs in the tested images. We observe that the file types of the extracted executed programs could be JAR written by JAVA, ELF implemented by C++, Shell Script, etc. It is quite challenging to analyze many kinds of software at the same time, since generating and confirming fingerprints for malware are both difficult and time-consuming. Therefore, we turn to online malware analysis tools for help. In particular, VirusTotal [11] is a highly comprehensive malware detection platform that incorporates various signature-based and anomaly-based detection methods employed by over 50 anti-virus (AV) companies such as Kaspersky, Symantec.

Therefore, it can detect various kinds of malware, including Trojan, Backdoor, and BitcoinMiner. As such, we employ VirusTotal to perform a primary screening. However, prior works have shown that VirusTotal may falsely label a benign program as malicious [14, 22, 29]. To migrate false positives, most of the prior works consider a program malicious if at least a threshold of AV companies detects it. In fact, there is no agreement in the threshold value. Existing detection has chosen two [14], four [22], or five [29] as the threshold. In this paper, to more precisely detect malicious programs, we consider a program as malicious only if at least five of the AV companies detect it. This procedure ensures that the tested programs are (almost) correctly split into benign and malicious ones.

The results of the primary screening only report the type of each malware provided by anti-virus companies. It is hard to demonstrate the accuracy of primary screening results, let along understanding the behavior of each malware. Therefore, after we obtain a list of potentially malicious files from the primary screening, a second screening is necessary to confirm the detection results and analyze the behavior of these files. Specifically, we dynamically run the potentially malicious files in a container and collect the logs of system call, network, and so on to expose the security violations of such files.

Finally, we implement a framework to finish the above pipeline. First, our framework utilizes our parser to locate and extract executed programs from the Docker images. Second, it leverages ViusTotal API to detect potentially malicious files. Third, we implement a container which contains a variety of tools such as `strace` and `tcpdump` for security analysis. Our framework leverages this container as a sandbox for automatically running and tracing the potentially malicious files to generate informative system logs. Since most benign images are filtered out by the primary screening, system logs are generated for only a few potentially malicious images. This framework greatly saves manual efforts and helps detect malicious images rapidly.

5.2 Distribution of Malicious Images

We first study the executed programs in the latest images in 147 official repositories, and we find that there are no malicious executed programs in these official images. Then, to facilitate in-depth analyses on community images, we extract the following subsets from the collected dataset for studying the executed programs. 1) The latest images in the top 10,000 community repositories ranked by popularity. 2) According to the popularity ranking, we divide the rest community repositories into 100 groups and randomly select 100 latest images from each group. In this way, we obtain 10,000 community images.

Results. On average, our parser proposed in Section 3.2 takes 5 and 0.15 seconds in analyzing one image and one file, respectively. The parser locates 693,757 executed programs from the tested images. After deduplication, we get 36,584 unique executed programs, in which there exist 13 malicious programs identified by our framework. The 13 malicious programs appear in 17 images. Moreover, we notice that all the malicious programs are entry-files in these

malicious images. This observation indicates that it is common for malicious images to perform attacks by directly utilizing a malicious entry-file, instead of subsequently triggered files.

Intuitively, the developer of a malicious image may release other malicious images on Docker Hub. Therefore, we propose to check the images that are related to malicious images. By leveraging the metadata collected in Section 3.2, once we detect a malicious image, we can investigate two kinds of related images: (1) the latest 10 images in the same repository and (2) the latest images in the most popular 10 repositories created by the same developer. We obtain 48 and 84 of these two types of related images respectively, in which there are 27 images contain the same malicious file found in the previously-detected malicious images. After analyzing all the related images by leveraging the framework developed in Section 5.1, we further obtain 186 new executed programs, in which there are 20 new malicious programs in 25 images. This insightful finding indicates that heuristic approaches, such as analyzing related images proposed in this work, are helpful in discovering malicious images and programs effectively. We hope that the malicious images and insights discovered in this paper can serve as an indicator for future works. The case study of the detected malicious images is deferred to Appendix A.3.

6 CVE Vulnerabilities

In this section, we evaluate the vulnerabilities in Docker images which are identified through Common Vulnerabilities and Exposures (CVE) IDs because the information of CVEs is public, expansive, detailed, and well-formed [34]. First, we leverage `Anchore` [2] to perform vulnerability detection for each image and study the distribution of the discovered vulnerabilities in Docker images. The results demonstrate that both official and community images suffer from serious software vulnerabilities. More analysis about vulnerablities in images are deferred to Appendix A.4. Then, we investigate the extra window of vulnerability in Docker images.

Defining of the extra window of vulnerability. To understand the timeline of the life cycle of a vulnerability in Docker images, accurately determining the *discovery* and *patch* time of a vulnerability, the *release* and *update* time of an image is vital. However, several challenges exist in determining different times. For instance, a vulnerability may be patched multiple times. In addition, different vendors might release different *discovery* times. Hence, we propose to first define these times motivated by existing research [34].

- *Discovery-time* is the earliest reported date of a software vulnerability being discovered and recognized to pose a security risk to the public.
- *Patch-time* is the latest reported date that the vendor, or the originator of the software releases a fix, workaround, or a patch that provides protection against the exploitation of the vulnerability. If the vulnerability is fixed by the upgrade of the software and the patch is not publicly available, we record the date of the upgrade of the software instead.

- For a vulnerability, *release-time* is the date that the developer releases an image that first brings in this vulnerability.

- For a vulnerability, *upgrade-time* is the date that the developer releases a new edition of the image, which fixes this vulnerability contained in the previous edition.

Suppose that a vulnerability of software S is discovered at T_d. Then after a period of time, the developer of S fixes this vulnerability at T_p. Image I first brings in this vulnerability at T_r. It takes extra time for the developer of I to fix the vulnerability and update this image at T_u. The developer of Image I is supposed to immediately fix the vulnerability once the patch is publicly available. However, it usually takes a long time before developers actually fix the vulnerability in an image. Therefore, we define W_e, the extra window of vulnerability in images, to measure how long it takes from the earliest time the vulnerability could be fixed to the time the vulnerability is actually fixed. Figure 4 presents three different cases of the extra window of vulnerability. In the first two cases, W_e is spreading from T_p to T_u. In the last case, even though the patch of a vulnerability is available, image I still brings in the vulnerability, so the W_e starts at T_r and ends at T_u. For all the cases, there is always an extra time window of vulnerability in image I before the developer updates I.

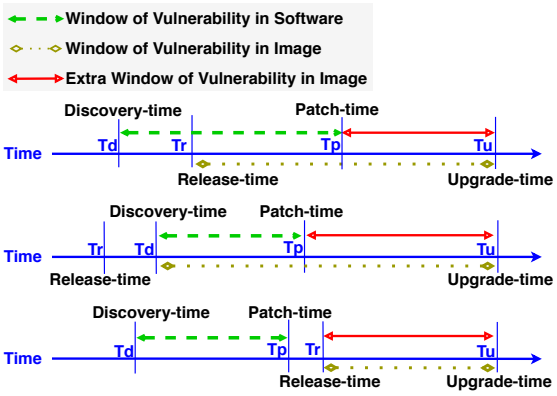


Fig. 4: Extra window of vulnerabilities in Docker images.

Obtaining different times. We choose the latest five editions of the 15 most popular images and randomly sample other 15 Docker images to investigate the extra window of vulnerabilities in images. By leveraging `Anchore` [2], we obtain 5,608 CVEs from these 30 images. After removing duplication, 3,023 CVEs remain, from which we randomly sample 1,000 CVEs for analysis. Then, we implement a tool to automatically collect *discovery*-, *patch*-time, and image *release*-, *update*-time for each CVE by leveraging the public information released on NVD Metrics [12]. We aim to obtain all the vital time of 1,000 CVEs from public information. However, the *discovery*- and *patch*-time of vulnerabilities are not always released in public. Therefore, we only obtain the complete information of vital time of 334 CVEs. It is worth noting that in some complex cases (e.g., there are multiple *discovery*-time for a CVE), we manually collect and confirm the complete information by reviewing the external references associated with CVEs.

Results. After analyzing the vital time of the 334 CVEs, we observe that it takes 181 days on average for common software to fix a vulnerability. However, the extra window of vulnerabilities in images is 422 days on average while the longest extra window of vulnerabilities could be up to 1,685 days, which allows sufficient time for attackers to craft corresponding exploitations of the vulnerabilities in Docker images.

7 Mitigating Docker Threats

In this section, we propose several possible methods to mitigate the threats uncovered in this paper.

Sensitive parameters. To mitigate attacks abusing sensitive parameters, one possible method is to design a framework which automatically identifies sensitive parameters and alerts users on the webpages of repository descriptions on Docker Hub. First of all, it is necessary to maintain a comprehensive list of sensitive parameters by manual analysis. After that, sensitive parameters in the descriptions of Docker images can be identified easily by leveraging string matching. Docker Hub should be responsible for displaying the detection results in the image description webpage and prompting the users about the possible risks of the parameters in run-commands. The above framework can be implemented as a backend of the website of Docker Hub or a browser plug-in. Additionally, runtime alerts, as adopted by iOS and existing techniques [33, 38], can warn users of potential risks before executing a run-command with sensitive parameters, which will be an effective mechanism to mitigate the abusing of sensitive parameters.

Malicious images. To detect malicious images, traditional static and dynamic analysis, e.g., signature-based method, system call tracing can be certainly helpful. However, many challenges do exist. For example, if the redundant files of an image cannot be removed accurately, it will be extraordinarily time-consuming to analyze all the files in an image by traditional methods. The framework proposed in Section 5.1 can be utilized to solve this problem. Furthermore, heuristic approaches, such as analyzing related images proposed in Section 5.2, could also be beneficial in discovering malicious images.

Vulnerabilities. Motivated by previous research [18], we believe that automatic updating is an effective way to mitigate the security risks from vulnerable images. However, software update becomes challenging in Docker. Because the dependencies among a large quantity of software are complicated and arbitrary update may cause a broken image. We propose multiple possible solutions to automated updating for vulnerable software packages. First, various categories of existing tools such as `Anchore` [2] can be employed to obtain vulnerability description information including the CVE ID of the vulnerability, the edition of the corresponding vulnerable software packages that bring in the vulnerability, the edition of software packages that repairs this vulnerability. Second, package management tools, e.g., apt, yum, can be helpful to resolve the dependency relationship among software packages. After we obtain the above information, we may safely update vulnerable software and the related software.

8 Discussion

In this section, we discuss the limitations of our approach and propose several directions for future works.

Automatically Identifying Malicious Parameters. We perform the first analysis on sensitive parameters and show that these parameters can lead to disastrous security consequences in Section 4. However, the sensitive parameters we discuss in this paper are recognized by manual analysis. Obviously, there are other sensitive parameters in the field that still need to be discovered in the future. Additionally, even though we discover many sensitive parameters on Docker Hub, it is hard to identify which parameters are published for malicious attempts automatically. Hence, one future direction to improve our work is to automatically identify malicious parameters in repository descriptions. One possible method is gathering images with the similar functionalities and employ statistical analysis to detect the deviating uses of sensitive parameters—deviations are likely suspicious cases, given that most images are legitimate.

Improving Accuracy for Detecting Malicious Images. In Section 5, we narrow down the analysis of malicious images to the executed programs. Although this method achieves the goal of discovering security risks in Docker Hub, it still has shortcomings for detecting malicious images accurately. For example, malware detection is a well-known arms-race issue [17]. First of all, VirusTotal, the detector we used in the primary screening may miss malware. Additionally, we cannot detect malicious images that perform attacks with other files rather than their executed programs. For example, our approach cannot detect images that download malware during execution, since our parser performs static analysis. Furthermore, after we obtain the system log of potential malicious images, we conduct manual analysis on these log files. Motivated by [21], we plan to include more automatic techniques to parse and analyzing logs as future work. Moreover, we plan to analyze more images in the future.

Polymorphic Malware. Malware can change their behaviors according to different attack scenarios and environments [25, 17]. Since our research goal is to characterize malware for measurement purposes instead of actually detecting them in practice, we employ existing techniques to find and analyze malware. However, it is valuable to understand the unique behaviors of Docker Malware. For instance, Docker has new fingerprints for malware to detect its running environments. How to emulate the Docker fingerprints to expose malicious behaviors can be an interesting future topic.

9 Related work

Vulnerabilities in Docker images. Many prior works focus on the vulnerabilities in Docker images [23, 36, 39, 27, 16]. For instance, Shu et al. proposed the *Docker Image Vulnerability Analysis (DIVA)* framework to automatically discover, download, and analyze Docker images for security vulnerabilities [36]. However,

these studies only investigate the distribution of vulnerabilities in Docker images, while our work uniquely conducts in-depth analysis on new risks brought by image vulnerabilities, such as the extra window of vulnerability.

Security reinforcement and defense. Several security mechanisms have been proposed to ensure the safety of Docker containers [26, 37, 35]. For instance, Shalev et al. proposed `WatchIT`, a strategy that constrains IT personnel's view of the system and monitors the actions of containers for anomaly detection [35]. There also exist other Docker security works that focus on defenses for specific attacks [24, 19]. For instance, Gao et al. discuss the root causes of the containers' information leakage and propose a two-stage defense approach [19]. However, these studies are limited to specific attack scenarios, which are not sufficient for a complete understanding of the security state of Docker ecosystems as studied in this work.

Registry security. Researchers have conducted a variety of works on analyzing the code quality and security in third-party code store and application registries, such as GitHub and App Store [28, 32, 20, 30, 15]. For instance, Bugiel et al. introduced the security issues of VM image repository [15]. Duc et al. investigated Google Play and the relationship between the end-user reviews and the security changes in apps [30]. However, Docker image registries such as Docker Hub have not been fully investigated before. This work is the first attempt to fill the gap according to our best knowledge.

10 Conclusion

In this paper, we perform the first comprehensive study of Docker Hub ecosystem. We identified three major sources for the new security risks in Docker hub. We collected a large-scale dataset containing both images and the associated meta-information. This dataset allows us to discover novel security risks in Docker Hub, including the risks of sensitive parameters in repository descriptions, malicious images, and the failure of fixing vulnerabilities in time. We developed new attacks to demonstrate the security issues, such as leaking user files and the host display. As the first systematic investigation on this topic, the insights presented in this paper are of great significance to understanding the state of Docker Hub security. Furthermore, our results make a call for actions to improve the security of the Docker ecosystem in the future. We believe that the dataset and the findings of this paper can serve as a key enabler for improvements of the security of Docker Hub.

Acknowledgements. This work was partly supported by the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, the National Key Research and Development Program of China under No. 2018YFB0804102, NSFC under No. 61772466, U1936215, and U1836202, the Zhejiang Provincial Key R&D Program under No. 2019C01055, and the Ant Financial Research Funding.

A Appendix

A.1 User Study on Sensitive Parameters

We design an online questionnaire that contains questions including "Do you try to fully understand every parameter of the run-commands provided on the Docker Hub website before running those commands?", "Do you make a security analysis of the compose.yml file before running the image?", etc. Our questionnaire was sent to our colleagues and classmates, and further spread by them. In order to ensure the authenticity and objectivity of the investigation results, we did not tell any respondents the purpose of this survey. We plan to conduct the user study in the official community of Docker Hub in the future.

Finally, we collected 106 feedback offered by 106 users from various cities in different countries. All of them have benefited from Docker Hub, i.e., they have experiences in using images from Docker Hub. Besides, they are from a broad range from both academia and industry fields, including students and researchers from various universities, software developers and DevOps engineers from different companies, etc.

As described in Section 4.2, the results of our user study show that 97% of users only care about if they can successfully run the image while ignoring how the images run, not to mention the sensitivity parameters in run-command and docker-compose.yml file. Even for 68 users who have a background in security research, only 10% of them indicate that they prefer to figure out the meaning of the parameters in run-commands.

A.2 Novel Attacks Exploiting Sensitive Parameters

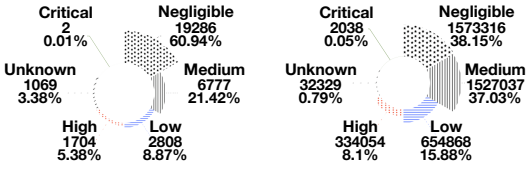
Obtaining the display of the host. `--privileged` is one of the most powerful parameters provided by Docker, which may pose a serious threat to users. When the operator uses command `--privileged`, the container will gain access to all the devices on the host. Under this scenario, the container can do almost anything with no restriction, which is extremely dangerous to the security of users. More specifically, `--privileged` allows a container to mount a partition on the host. By taking a step further, the attacker can access all the user files stored on this partition. In addition to accessing user files, we design an attack to obtain the display of a user's desktop. In fact, with `--privileged`, a one-line code, `cp /dev/fb0 user_desktop.txt`, is sufficient for attackers to access user display data. Furthermore, by leveraging simple image processing software [7], attackers can see the user's desktop as if they were sitting in front of the user's monitor.

Spying the process information on the host. `--pid` is a parameter related to namespaces. Providing `--pid=host` allows a container to share the host's PID namespace. In this case, if the container is under the control of an attacker, all the programs running on the user's host will become visible to the attacker inside the container. Then, the attacker can utilize these exposed information such as the `PID`, the `owner`, the path of the corresponding executable file and the execution parameters of the programs, to conduct effective attacks.

A.3 Case Study of Malicious Images

We manually conduct analysis on detected malicious images. For instance, the image `mitradomining/ffmpeg` on Docker Hub is detected as malicious by our framework. The entry-file of this image is `/opt/ffmpeg` [7]. According to the name and entry-file of the image, the functionality of this image should be image and video processing. However, our framework detects that the real functionality of the entry-file is mining Bit-coins. By leveraging the syscall log reported by our framework, we determine that the real identity of this image is a Bit-coin miner. Thus, once users run the image, their machines will become slaves for cryptomining.

A.4 Distribution of Vulnerabilities



(a) Official images. (b) Community images.

Fig. 5: Vulnerabilities existing in the latest images.

We investigate the distribution of vulnerabilities in the latest version of all official images. First, we observe that the latest official images contain 30,000 CVE vulnerabilities. Figure 5(a) categorizes these CVE vulnerabilities into 6 groups according to the severity levels assessed by the latest CVSSv3 scoring system [12]. Although only 6% of vulnerabilities are highly/critically severe, they exist in almost 30% of the latest official images. Furthermore, we conduct a similar analysis on the latest images in the 10,000 most popular community repositories. As shown in Figure 5(b), the ratios of vulnerabilities with medium and high severity increase to over 37% and 8%, respectively, which are higher than those of official images. In addition, it is quite alarming that more than 64% of community images are affected by highly/critically severe vulnerabilities such as the denial of service and memory overflow. These results demonstrate that both official and community images suffer from serious software vulnerabilities. Additionally, community images contain more vulnerabilities with higher severity. Hence, we propose that software vulnerability is an urgent problem which seriously affects the security of Docker images.

References

1. Amazon Elastic Container Servicen (August 2019), `https://aws.amazon.com/getting-started/tutorials/deploy-docker-containers`
2. Anchore (August 2019), `https://anchore.com/engine/`
3. API to get Top Docker Hub images (August 2019), `https://stackoverflow.com/questions/38070798/where-is-the-new-docker-hub-api-documentation`
4. Docker (August 2019), `https://www.docker.com/resources/what-container`

5. Docker Hub Documents (August 2019), https://docs.docker.com/glossary/?term=Docker\%20Hub
6. Docker Security Best-Practices (August 2019), https://dev.to/petermbenjamin/docker-security-best-practices-45ih
7. FFmpeg (August 2019), http://ffmpeg.org
8. Malicious Docker Containers Earn Cryptomining Criminals $90K (August 2019), https://kromtech.com/blog/security-center/cryptojacking-invades-cloud-how-modern-containerization-trend-is-exploited-by-attackers
9. Running Docker in Production (August 2019), https://ghost.kontena.io/docker-in-production-good-bad-ugly
10. strings(1) - Linux man page (August 2019), https://linux.die.net/man/1/strings
11. Virustotal Api (August 2019), https://pypi.org/project/virustotal-api/
12. Vulnerability Metrics (August 2019), https://nvd.nist.gov/vuln-metrics/cvss
13. Understanding the Security Risks of Docker Hub (July 2020), https://github.com/decentL/Understanding-the-Security-Risks-of-Docker-Hub
14. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket. In: NDSS. vol. 14, pp. 23–26 (2014)
15. Bugiel, S., Nürnberger, S., Pöppelmann, T., Sadeghi, A.R., Schneider, T.: Amazonia: when elasticity snaps back. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 389–400. ACM (2011)
16. Combe, T., Martin, A., Di Pietro, R.: To docker or not to docker: A security perspective. IEEE Cloud Computing **3**(5), 54–62 (2016)
17. Cozzi, E., Graziano, M., Fratantonio, Y., Balzarotti, D.: Understanding linux malware. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 161–175. IEEE (2018)
18. Duan, R., Bijlani, A., Ji, Y., Alrawi, O., Xiong, Y., Ike, M., Saltaformaggio, B., Lee, W.: Automating patching of vulnerable open-source software versions in application binaries. In: NDSS (2019)
19. Gao, X., Gu, Z., Kayaalp, M., Pendarakis, D., Wang, H.: Containerleaks: Emerging security threats of information leakages in container clouds. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 237–248. IEEE (2017)
20. Gorla, A., Tavecchia, I., Gross, F., Zeller, A.: Checking app behavior against app descriptions. In: Proceedings of the 36th International Conference on Software Engineering. pp. 1025–1035. ACM (2014)
21. He, P., Zhu, J., He, S., Li, J., Lyu, M.R.: Towards automated log parsing for large-scale log data analysis. IEEE Transactions on Dependable and Secure Computing **15**(6), 931–944 (2017)
22. Kotzias, P., Matic, S., Rivera, R., Caballero, J.: Certified pup: abuse in authenticode code signing. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 465–478 (2015)
23. Kudva, P.: Security analysis of container images using cloud analytics framework. In: Web Services–ICWS 2018: 25th International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings. vol. 10966, p. 116. Springer (2018)
24. Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., Zhou, Q.: A measurement study on linux container security: Attacks and countermeasures. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 418–429. ACM (2018)

25. Liu, B., Zhou, W., Gao, L., Zhou, H., Luan, T.H., Wen, S.: Malware propagations in wireless ad hoc networks. IEEE Transactions on Dependable and Secure Computing **15**(6), 1016–1026 (2016)
26. Loukidis-Andreou, F., Giannakopoulos, I., Doka, K., Koziris, N.: Docker-sec: A fully automated container security enhancement mechanism. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). pp. 1561–1564. IEEE (2018)
27. Martin, A., Raponi, S., Combe, T., Di Pietro, R.: Docker ecosystem–vulnerability analysis. Computer Communications **122**, 30–43 (2018)
28. Martin, W., Sarro, F., Yue, J., Zhang, Y., Harman, M.: A survey of app store analysis for software engineering. IEEE Transactions on Software Engineering **43**(9), 817–847 (2017)
29. Miller, B., Kantchelian, A., Tschantz, M.C., Afroz, S., Bachwani, R., Faizullabhoy, R., Huang, L., Shankar, V., Wu, T., Yiu, G., et al.: Reviewer integration and performance measurement for malware detection. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 122–141. Springer (2016)
30. Nguyen, D., Derr, E., Backes, M., Bugiel, S.: Short text, large effect: Measuring the impact of user reviews on android app security and privacy. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 155–169. IEEE (2019)
31. Rastogi, V., Davidson, D., Carli, L.D., Jha, S., Mcdaniel, P.: Cimplifier: automatically debloating containers. In: Joint Meeting on Foundations of Software Engineering (2017)
32. Ray, B., Posnett, D., Filkov, V., Devanbu, P.: A large scale study of programming languages and code quality in github. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 155–165. ACM (2014)
33. Ringer, T., Grossman, D., Roesner, F.: Audacious: User-driven access control with unmodified operating systems. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 204–216. ACM (2016)
34. Shahzad, M., Shafiq, M.Z., Liu, A.X.: A large scale exploratory analysis of software vulnerability life cycles. In: 2012 34th International Conference on Software Engineering (ICSE). pp. 771–781. IEEE (2012)
35. Shalev, N., Keidar, I., Weinsberg, Y., Moatti, Y., Ben-Yehuda, E.: Watchit: Who watches your it guy? In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 515–530. ACM (2017)
36. Shu, R., Gu, X., Enck, W.: A study of security vulnerabilities on docker hub. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. pp. 269–280. ACM (2017)
37. Sun, Y., Safford, D., Zohar, M., Pendarakis, D., Gu, Z., Jaeger, T.: Security namespace: making linux security frameworks available to containers. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 1423–1439 (2018)
38. Wijesekera, P., Baokar, A., Tsai, L., Reardon, J., Egelman, S., Wagner, D., Beznosov, K.: The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 1077–1093. IEEE (2017)
39. Zerouali, A., Mens, T., Robles, G., Gonzalez-Barahona, J.M.: On the relation between outdated docker containers, severity vulnerabilities, and bugs pp. 491–501 (2019)