Harbormaster: Policy Enforcement for Containers

Mingwei Zhang
Stony Brook University
mizhang@cs.stonybrook.edu

Daniel Marino Petros Efstathopoulos
Symantec ResearchLabs
Daniel_Marino@symantec.com
Petros_Efstathopoulos@symantec.com

***Abstract*—Lightweight virtualization, as implemented by application container solutions such as Docker, have the potential to revolutionize the way multi-tier applications are developed and deployed, especially in the cloud. The success of application containers can be partly attributed to their ability to share resources with the underlying platform that hosts them. As such, the isolation provided by such containers is not as strict as with traditional VMs. These very characteristics that have contributed to the success of application containers can also be seen as factors that limit their widespread commercial adoption, since enterprise IT administrators cannot implement the various–and often fine-grained–security policies they are required to abide by. This problem is of limited consequence when a host is running a single user's application containers. But sharing compute resources among multiple users is an important benefit of containers and cloud-based deployment. In this paper we present a preliminary discussion of the challenges associated with enterprise security policy management for application containers deployed in multi-user environments. Furthermore, we present Harbormaster, a system that addresses some of these challenges by enforcing policy checks on Docker container management operations and allowing administrators to implement the *principle of least privilege*.**

I. Introduction

Application container technologies, such as Docker, are revolutionizing the way that applications are deployed on enterprise infrastructure in the cloud [1], [2]. Rather than manually installing individual components, or packaging them into a heavyweight VM, all of the required components for an application can be encapsulated in a lightweight container that can easily be deployed on any platform that supports the container technology. This new style of deployment enables flexible and efficient use of cloud compute resources.

To take full advantage of the infrastructure utilization efficiency provided by container-based deployment, multiple users will need to access a server that hosts containers, or a *container host*. Unfortunately, Docker and other lightweight virtualization solutions do not provide the same degree of isolation that traditional virtual machine (VM) infrastructure provides. In fact, the efficiency gains that Docker enjoys over traditional VMs are largely attributable to allowing containers to access and share the resources of the underlying operating system. This complicates efforts to secure a container host, since some container management commands can impact other containers on the server or can grant application code within the container privileged access to the underlying operating system. Traditional OS-level user permissions can be easily subverted on a container host since any user with the ability to issue Docker container management commands can freely launch a container that enjoys unrestricted, root access to the underlying operating system.

In addition to the lack of multi-user privilege management, the move to container-based application deployment creates other concerns for a security administrator. The increasing availability of ready-to-run application containers available on public repositories amplifies the existing concern that users may run unvetted code from untrusted sources and unwittingly provide a vector for attack. This is of particular concern when the container host may have access to sensitive data. In order for enterprise IT administrators to confidently allow container-based application deployment in the cloud, we must provide them with a privilege management tool they can use to implement security policies and maintain regulatory compliance.

Our goal is to investigate mechanisms that will address some of the security concerns related to the usage of application containers by enterprises. To that end, in this paper we tackle the security and management gap that currently exists between lightweight containers and traditional VMs. In addition, the application-centric nature of containers can enable useful policies that would be difficult to enforce for traditional VMs. We describe our prototype implementation of Harbormaster, a tool that allows a container host administrator to:

- securely empower multiple users to issue management commands without allowing them to abuse the root privileges required by the Docker daemon
- implement the principle of least privilege by granting users fine-grained privileges to perform only those tasks that they need to
- secure container hosts and the sensitive data they have access to by controlling what images may be installed on enterprise infrastructure and how they can be operated.

The rest of this paper is organized as follows: Section II provides some background on Docker and the challenges in question, Section III discusses the principles behind the proposed Harbormaster system, Section IV briefly describes our prototype implementation, Section V presents a preliminary evaluation, while Section VI concludes the paper.

II. Background

Lightweight, or OS-level, virtualization refers to a variety of techniques used to sandbox, constrain, or simply modify the resource namespace of a process or a group of processes. It has a long history stretching back at least to the introduction of the `chroot` system call in Version 7 Unix. Notable implementations include BSD Jails, OpenVZ, and Linux Containers (LXC) which provide varying degrees of isolation [3], [4], [5]. These approaches avoid the overheads associated with traditional virtual machines. The resulting runtime performance improvement has been amply documented in recent research in a variety of application areas from high performance computing (HPC) to application security monitoring [6], [7], [8].

Docker [9] has reinvigorated interest in lightweight virtualization by providing an easy-to-use interface for accessing the virtualization primitives built in to the Linux kernel, adding support for application image repositories, and leveraging an efficient copy-on-write file system for packaging application images with minimal space overhead. Due to its increasing popularity, we introduce our approach in the context of Docker.

A. Docker Terminology

Docker applications are packaged as *images*. A Docker image is essentially a read-only snapshot of a file system image containing the resources required by the application along with some metadata that, among other things, may indicate what executable within the image should be used to run the application. [1] Images are used to launch *containers*, which are dynamic instances of the application. The same image can be used to launch multiple containers. Once created from an image using a `run` command, containers can be stopped and started, and changes made to the container's internal file system persist within the container.

New images are created by downloading an existing image (using a `pull` command) from a public or private Docker registry to use as a base, and then modifying it. The modification can be done interactively by launching a container from the image and manually issuing commands, or via a script known as a Dockerfile. The new image is then `push`ed to a Docker registry where it is stored.

B. Docker architecture

Docker employs a client/server program architecture. Each container host runs an instance of the Docker daemon. Docker clients issue container management commands to the daemon using an HTTP-based protocol known as the Docker Remote API. The daemon processes these commands and performs the requested actions, such as downloading an application image from a repository or launching a container from an image.

The parameters passed to the `run` command used to launch a Docker container control the degree to which the container is isolated from the underlying operating system. Oftentimes it is desirable to give the container access to resources on the container host. For instance, an application image for an Apache HTTP Server can be launched and granted access to a directory on the host file system that contains a particular web application's resources (HTML files, PHP scripts, etc.), and be allowed to open network port 80.

Docker even provides a `--privileged` option which gives the application running within the container nearly the same access to the host as an uncontained process running with root privileges. While this may seem to defeat the primary purpose of a container, this flexibility can be useful. Consider deploying a security tool, with a complex set of library dependencies, on a container host. By packaging such a tool as a Docker image, we can exercise control over the dependencies and ease deployment. But, by running the image as a container with the `--privileged` option, we grant the security application low level access to all storage devices on the host, allowing it to perform system-wide scanning. Privileges can also be granted to a container in a more fine-grained manner by specifying an allowed set of Linux capabilities [10].

In order to access the kernel's lightweight virtualization primitives, the Docker daemon must run with `root` privileges. Each container process, which is forked from the daemon, initially runs as `root` and eventually relinquishes privileges leaving itself with the level of access specified by the parameters used to launch it.

C. Third Party Tools

By using the Docker Remote API, third parties can build orchestration tools to help manage complex container deployments. For instance, Google's Kubernetes and Amazon's Elastic Beanstalk are tools for scheduling, replicating, and load balancing between Docker containers [11], [12]. In our work, we also leverage the public availability of the Docker Remote API to provide a policy enforcement layer for Docker container hosts, but we do so in a way that maintains compatibility with higher level orchestration tools built for Docker. In this way, our privilege management tool can fit seamlessly into the growing Docker ecosystem.

D. Docker Security and Related Work

The Docker community has recently put serious effort into improving its security. This includes exploring how mandatory access control for contained apps can be implemented through SELinux and Apparmor policies, system call filters such as seccomp [13], and by running containers with minimal Linux capabilities [14]. These efforts help to protect an underlying platform (and other containers running on it) if, for instance, a containerized application is compromised by exploiting an unknown vulnerability. In addition, a more constrained environment has been introduced for some operations such as `docker pull` [15]. Most recently, Docker has introduced image signing and verification which helps ensure that an image comes from the expected publisher and has not been tampered with [16]. Each of these efforts aims to protect against external threats; they do nothing to restrict the actions of users who have permission to issue commands to the Docker daemon. This paper describes a way to enforce policies on legitimate users of Docker, enabling administrators to implement the principle of least privilege.

Docker aims to keep its core functionality lean, but to allow external tools to build additional functionality through its APIs. So, for instance, docker-swarm [17] and Google Kubernetes [11], [18] have stepped in to simplify complex deployments of interconnected Docker containers. In Kubernetes, deployments are described by a so-called `pod` file. Kubernetes reads this deployment description and automatically issues appropriate commands to create the containers on available

[1]The image actually consists of a series of file system layers each of which contains a set of differences from the previous layer. A union file system is used to combine these layers into a single file system that contains all of the application's resources.

Docker container hosts and to connect them to each other as needed. But again, anyone who has access to Kubernetes, or the Docker daemons to which it issues commands, essentially has unconstrained access to the container host servers. Our work, in contrast, provides a policy enforcement mechanism that allows an administrator to provide various, fine-grained levels of access to multiple users on a container host. In keeping with Docker's principle of maintaining a minimal core set of functionality, we accomplish this without modifying Docker itself.

E. Lack of multi-user support

If a single user is responsible for deploying applications on a Docker container host, then one could argue that there is no need for additional policy controls. However, we believe that containers have the potential to replace traditional VMs for many uses, allowing for more efficient use of cloud resources. Although Docker does not yet provide the level of isolation desired in many multi-tenant environments, it can serve as an appropriate lightweight replacement for VMs on a server (or a cloud VM) that is shared among multiple users within an organization. But, while hypervisors such as VMWare ESXi [19] offer robust support for controlling the privileges that multiple users have for managing their VMs, we know of no analogous solution for Docker or other lightweight containers.

Good security practice dictates that administrators should adhere to the principle of least privilege, granting users only the privileges needed to accomplish their job. Unfortunately, once a user is able to issue commands to a Docker daemon, they are free to download and launch any container (potentially from an untrusted source) with arbitrarily high privileges. We believe that providing administrators with a tool to constrain these commands and enforce all necessary enterprise-specific policies is an essential prerequisite to the widespread adoption of container technologies. In this spirit, we introduce Harbormaster.

III. Harbormaster Design

Harbormaster protects container hosts by enforcing fine-grained policies on the container management commands issued by users. It accomplishes this by *proxying* all commands destined for the Docker daemon and evaluating them for policy compliance before passing them along. In this section we describe the design of Harbormaster's architecture and the types of policies it can enforce.

A. Harbormaster system architecture

The basic system architecture is shown in Figure 1. Container users issue commands to the Harbormaster gateway which, in turn, issues commands to the Docker daemon. The Harbormaster gateway completely mediates all communication with the Docker daemon; no other user or process is given a communication channel to the daemon. [2]

From the point of view of the user, the Harbormaster gateway exposes exactly the same interface as the Docker daemon. We chose this proxy-based architecture in order to ensure that Harbormaster would easily fit into the growing Docker ecosystem. Whether Docker Remote API commands are issued directly by a user, or by a third-party orchestration tool such as Kubernetes, our gateway transparently proxies the commands and ensures policy compliance.

In our prototype implementation, policies are stored locally on the machine that hosts the Harbormaster gateway, but we envision an architecture where policies for many container hosts are stored on a central policy server. Individual gateway instances would contact the policy server to request applicable policies. In our prototype, an administrator is responsible for defining all aspects of the policy for all images. But, we envision that in a more complete implementation, non-admin users could be granted the privilege to adjust certain aspects of the policy for new images that they create.

B. Harbormaster Policies

We will now describe some policies that can be expressed in Harbormaster. Our prototype implementation, which accepts policies written in an ad-hoc XML format, can constrain a wide variety of options to the Docker Remote API. In this section, we highlight the most important operations that Harbormaster can constrain. We use the simplified, declarative rule language shown in Figure 2 for presentation purposes.

We divide the policy rules that Harbormaster enforces into two categories.

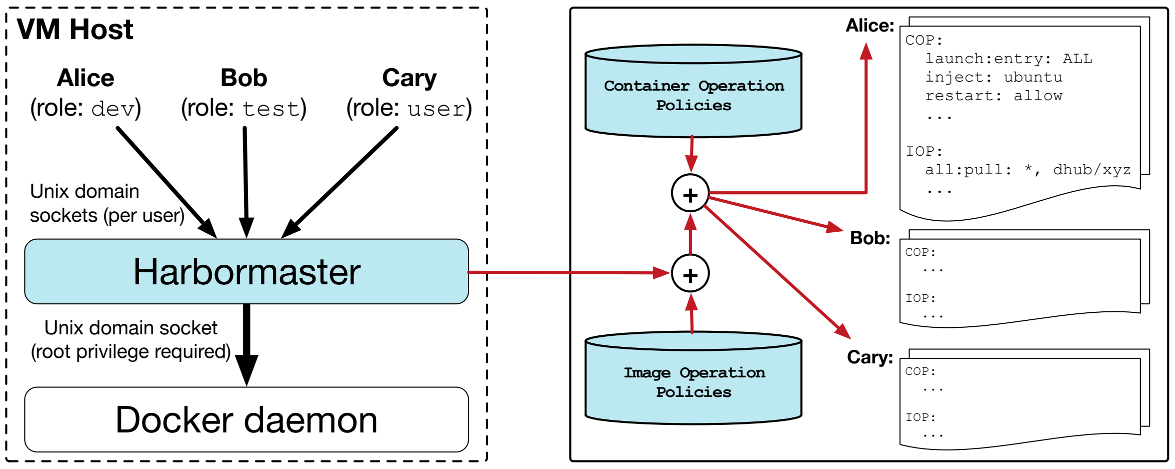
- *Image Operation Policy (IOP)* These policies constrain Docker commands such as `pull`, `build`, and `push` that fetch, create, and store images.
- *Container Operation Policy (COP)* These policies constrain Docker commands that create new containers and operate on existing containers. Perhaps the most important of these policies are those that limit the Docker `run` command which is used to launch a new container from an image and specifies which process is to be run within the container, which host resources will be made available to the container, and any special privileges that processes within the container will enjoy.

1) Image Operation Policy: Harbormaster allows an administrator to specify which images can be downloaded from registries, and by which users. For instance:

- `ALLOW ALL PULL hub.docker.com/*`
  Grant all users the ability to pull any image from the official public Docker registry.
- `DISALLOW ALL PULL hub.docker.com/vulnerable-app`
  But, you may want to disallow anyone from downloading certain apps (e.g., a web server) with known vulnerabilities.

Normally, we imagine that an administrator would create a global whitelist of images that all users are allowed to download, but more fine-grained policies are possible. For instance, a trusted developer may be given permission to pull arbitrary images from the docker hub, but operations personnel

[2]Preventing users from directly accessing the daemon can be done in a variety of ways. In our prototype, we bind the Docker daemon to a unix domain socket and restrict access with file permissions.



```
COP:
  launch:entry: ALL
  inject: ubuntu
  restart: allow
  ...
IOP:
  all:pull: *, dhub/xyz
  ...
```

```
COP:
  ...
IOP:
  ...
```

```
COP:
  ...
IOP:
  ...
```

Fig. 1.   Architecture of Harbormaster

Policy Rules		
Image Operation Rule	:=	[DIS]ALLOW ⟨*user_spec*⟩ ⟨*image_action*⟩ ⟨*image_spec*⟩
Container Operation Rule	:=	[DIS]ALLOW ⟨*user_spec*⟩ ⟨*container_action*⟩ FOR ⟨*container_spec*⟩
Policy Objects		
user_spec	:=	Linux user name or group name or ALL
image_spec	:=	Docker repository name plus a regular expression for name matching
container_spec	:=	regular expression matching a container name, or an ⟨*image_spec*⟩ :: ⟨*user_spec*⟩ which will match all containers launched from a matching image by a matching user
file_spec	:=	Linux pathname with regular expression support
port_spec	:=	list of port ranges or ALL
capability_spec	:=	list of Linux capabilities (e.g., CAP_NET_RAW) or ALL
resource_constraint_spec	:=	a condition (<,>,==) on any of the resource weighting priorities supported by Docker (e.g., CPU usage, memory usage, I/O bandwidth)
Image Policy Action (*image_action*) **Descriptions**		
PULL		download and store an image
EXTEND FROM		create a new image using the specified image as a base
REMOVE		delete an image
Container Policy Action (*container_action*) **Descriptions**		
MAP HOST FILE		takes a ⟨*file_spec*⟩ as an argument and allows user to launch the specified container with access to the given host file(s)
MAP HOST PORT		takes a ⟨*port_spec*⟩ as an argument and allows user to launch the specified container with access to the given host port(s)
GRANT CAPABILITY		takes a ⟨*cap_spec*⟩ as an argument and allows user to launch specified container with the given capabilities
SPECIFY ENTRYPOINT		allows user to use a non-default entrypoint to launch the specified container
INJECT PROCESS		allows user to execute arbitrary file in an already running container
STOP/START/RESTART		allows user to control container lifecycle
MODIFY RESOURCE USAGE		takes a ⟨*resource_constraint_spec*⟩ as an argument and allows user to launch specified container with an altered resource priority

Fig. 2.   Syntax of our simplified presentation policy language. Our implementation uses a less succinct XML policy specification and supports all of these constraints and more.

may only download and launch a few production images from an internal registry.

An administrator may also want to constrain who is able to modify images in order to create new images. For instance:

- `ALLOW developers EXTEND FROM hub.docker.com/*`
  Grants users in the "developers" group the permission to build new images based on containers launched from official Docker registry images.

In our current implementation, we use operating system users and groups for authentication and to establish identity. This could easily be replaced by another authentication or access control mechanism.

2) Container Operation Policy: Creating a container from an image via the Docker `run` or `create` command is an operation that is essential to constrain. As mentioned in Section II, container processes are forked from the Docker daemon which runs with root privileges. The options included in the `run` or `create` command line determine what privileges, if any, the newly launched process relinquishes. Harbormaster's container launch policies constrain these sensitive commands, as well as other commands affecting the operation of an existing container.

Harbormaster allows users to run any image that the administrator has permitted them to download. In addition, once a user creates a new image, he is able to run it. In a more complete implementation, we envision that users would be able to specify other users or groups of users that are allowed to run the images that they create, rather than relying on an administrator to update the policy.

By default, Harbormaster ensures that a user is not able to launch a container with more privileges than the user is granted by the operating system. For instance, Harbormaster enforces that a user can mount host data into a container only if the user has file system permissions to that data. Similarly, Harbormaster checks any attempt to launch a container with a specific operating system privilege to ensure that the user initiating the operation has the privilege. An administrator may, however, grant additional privileges to specific users when launching specific images. For instance:

- `ALLOW qa MAP HOST FILE /private/website FOR apache`
  Allow users in the "qa" group to launch a container from the apache web server image and give it access to directory /private/website.
- `ALLOW qa GRANT CAPABILITY CAP_NET_BIND_SERVICE FOR apache`
  Grants users in the "qa" group the permission to launch a container from the apache image with a specific operating system capability, in this case the Linux CAP_NET_BIND_SERVICE which allows binding to a privileged port.

The administrator in the previous example has granted the QA users the ability to launch Apache with access to a privileged directory containing the website. But, following the principle of least privilege, she may not want these users to have access to some private PHP scripts within that directory. Unfortunately, Docker allows launching a container from an image with a non-default entry point. For instance, the command

```
$ docker run -v /private/website:/var/www \
> -v ~:/myhome apache cp -r /var/www /myhome/
```

launches the apache image, gives it access to both the private website directory and the user's home directory. But then, instead of executing the default `httpd` process, it copies the private website directory into the user's home directory. In order to prevent circumventing the intended policy in this way, Harbormaster disallows launching images with non-default entrypoints unless it is explicitly allowed by a policy. A similar issue arises with the Docker `exec` command, which allows a user to run a new process within an existing container. Privileges to use these Docker features can be granted to specific users for specific images. For instance:

- `ALLOW developer SPECIFY ENTRYPOINT FOR ALL`
  Grants users in the "developer" group the ability to specify a non-standard entrypoint for all images which they are permitted to run.
- `ALLOW developer INJECT PROCESS FOR ubuntu`
  Grants users in the "developer" group the ability to issue `exec` commands to inject processes into containers launched from the ubuntu image.

Finally, Harbormaster policies can grant privileges on the ability to stop and start containers launched by others. For instance:

- `ALLOW ops RESTART FOR ALL::ALL`
  Grants users in the "ops" group the ability to stop and start containers launched by all other users.

IV. IMPLEMENTATION

As mentioned in Section III, the Harbormaster prototype uses an XML file for policy specification. The Harbormaster gateway process reads this file, starts the Docker daemon and binds it to a socket accessible only to Harbormaster (which, like the Docker daemon, runs with root privileges), and awaits container management commands.

We adopt a simple approach to authenticating and identifying multiple users in our prototype by leveraging Linux users and file permissions. A dedicated named pipe is created for each user to communicate with the Harbormaster gateway. Each user's environment is modified to set the DOCKER_HOST variable to their personal IPC pipe to the Harbormaster gateway. File permissions are used to prevent any other user from sending commands via another user's IPC channel. In this way, the Harbormaster gateway can easily identify the user who is issuing a command, and apply the appropriate policy before possibly forwarding the command to the Docker daemon.

Our simple prototype allowed us to experiment with the feasibility of intercepting and modifying Docker commands, and with a variety of policies for safely sharing a container host among users. But, the design could be extended in many ways to create a more practical, production-level system. Most notably, policies could be hosted on a central server for use by multiple hosts. A level of discretionary access control could be added for users to grant certain privileges on containers

TABLE I. HARBORMASTER POLICY PARSING AND EVALUATION PERFORMANCE.

Number of Rules	Time to Evaluate
5	0.07 sec
50	0.1 sec
10,000	0.5 sec
100,000	6 sec

they own. In fact, a full XACML-based access control policy mechanism could be put in place to allow policies to be based on rich attributes flexibly assigned to users, resources, and the current environment [20].

V. EVALUATION OF PRELIMINARY PROTOTYPE

In this section, we describe an initial evaluation of our Harbormaster gateway which serves as a proxy for all communication with the Docker daemon. In our effort to assess our current prototype, we perform the following steps: 1) we study the runtime overhead added to Docker operations by the proxy, and confirm that Harbormaster transparently proxies commands even when they are issued by third-party tools, such as Kubernetes, 2) we verify the effectiveness of the prototype in stopping attempts to break the enforced policy, and 3) we attempt to quantify the impact of Harbormaster's presence on addressing a specific class of potential threats related to Docker entry point misuse.

A. Runtime performance

The most crucial Harbormaster policies are checked and enforced during Docker container launch operations. We used a particularly popular Docker container image—the official `MySQL` image available on the Docker registry—and a simple policy in order to measure the overhead that Harbormaster introduces by proxying communication with the Docker daemon. Our testbed is a Linux x86-64 Ubuntu 14.04 System running on an Intel Core-i5 4200 CPU, with 12 Gigabytes of RAM and an 128GB SSD. We measured the impact on launch time both when launching a single container and when launching up to fifty containers from the image simultaneously. Figure 3 shows that the median overhead introduced by Harbormaster is 11.2%, which we find acceptable. The policy we are using for this experiment is very simple. It contains only two rules, of which the first requires that the container should not be started in privileged mode and the second requires that the data mapped into the container can only be at location `/home/public`. More complicated policies increase the overhead somewhat, as shown in Table I, but keep in mind that this overhead applies only to container management operations and that these times reflect the performance of our unoptimized prototype. Harbormaster does not introduce any overhead to the runtime performance of the contained application.

a) Integration with Kubernetes.: We have also tested Harbormaster with a popular container management system, Kubernetes, which we described in Section II. In particular, we tested our system in the Google container cloud using a small deployment consisting of two VMs: a master and a slave node, both protected by Harbormaster. We exercised the system through Kubernetes for the duration of a week, successfully testing all Kubernetes' sample applications (pods)

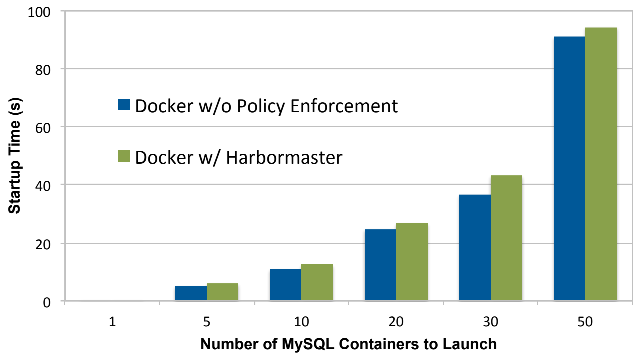


Fig. 3. Harbormaster container launch time overhead.

in `kubernetes/examples/guestbook`, with no issues. This simple experiment demonstrates that our prototype successfully parses the Docker Remote API commands whether they are issued directly by the Docker client, or by a third-party tool such as Kubernetes. This ability to expose the same interface as the Docker daemon and to transparently proxy all commands allows Harbormaster to seamlessly integrate with the Docker ecosystem.

B. Harbormaster policy enforcement scenarios.

To evaluate the effectiveness of Harbormaster, we evaluated a hypothetical scenario involving an organization with several users sharing a single Docker container host which is administered by a developer `Alice`. Our organization has another member `Cary` who is a developer in a different group, and `Bob` who is a tester in the QA group. We simulated a scenario where `Alice` builds a new containerized application and wants to allow `Bob` to access her host to test the application and to grant `Cary` access to her host in order to experiment with Docker and build her own containerized applications. `Alice` doesn't want either of the others to have access to the private data located on her container host.

We found that Harbormaster is able to effectively enforce policy in this scenario. In Table II, we list several unauthorized commands that `Cary` attempts on `Alice`'s host, but that Harbormaster is able to reject based on the policy `Alice` has put in place. For instance, `Cary`'s attempts to download an unauthorized image is rejected, as is her attempt to copy the file with `Alice`'s passwords. Note that even if we trust Cary not to act maliciously, the principle of least privilege dictates that she should be prevented from performing sensitive actions that she doesn't need to. That way, even if Cary's account is compromised by a malicious user, the damage is contained. Harbormaster allows the implementation of such a policy on a container host.

C. Automatic Docker image analysis

To better understand the potential impact of Harbormaster and the opportunity to enforce policies that prevent undesirable behaviors, we used the Docker registry API [21] in order to analyze a large number of Docker images publicly available in the Docker hub registry [22]. The goal of our analysis was to assess the degree to which these images contain configuration

TABLE II. UNINTENDED BEHAVIOR PREVENTED BY HARBORMASTER

Unintended Behavior	Action	Policy Type
`docker run pull unauthorized_img`	Reject	IOP
`docker rmi alice_image`	Reject	IOP
`docker rm alice_container`	Reject	COP
`docker run -v/data/alice:/data cary/ftp scp /data myweb`	Reject	COP
`docker run -v/etc/shadow:/tmp/pass ubuntu scp /tmp/pass cary_web`	Reject	COP

oversights that may lead to unintended behaviors. In particular, for the purposes of this paper, we will focus on our findings related container entry point specification, or lack thereof.[3]

We crawled 84,897 Docker repositories (images) from 24,066 users by recursively calling the API query functions (similar to `docker search`). This corresponds to the majority of the all repositories in the public Docker registry, including 72 official repositories as well as 31,790 automatically built repositories connected with `github`.

a) Entry point analysis.: Of the 84,897 images that we analyzed, 12,845 (15.1%) of the images have an explicit entry point specified in their metadata. These images were generated from a Dockerfile containing an `ENTRYPOINT` statement. The rest have a default entry point specified in their metadata which will only be used if another command is not specified on the `docker run` command line. These images were built from a Dockerfile containing a `CMD` statement.

The existence of an `ENTRYPOINT` in the metadata is a strong indication that the container is intended to run only the specified command. Harbormaster can be used to ensure that users do not use special Docker command line switches to override this intended entry point. Tables III and IV demonstrate the top-10 entry points as indicated by `ENTRYPOINT` and `CMD` specifications respectively.

TABLE III. TOP-10 ENTRY POINTS (FOR CONTAINERS WITH NO ENTRYPOINT STATEMENTS)

Entry point	Number
/entrypoint.sh	780
/docker-entrypoint.sh	347
/start.sh	179
/run.sh	172
/usr/local/bin/jenkins.sh	165
/bin/bash	151
/bin/sh -c usr/bin/mongod	140
/usr/bin/redis-server	132
/app/init	107
/usr/sbin/apache2	75
total	12845

In Table III we observe that 151 images (0.9% of all images) have entry point programs such as`/bin/bash` or `/bin/sh`. The protection afforded by enforcing these shell entry points is of limited value, since users can launch any shell command in interactive mode. But, one could imagine identifying these images and either forbidding them altogether, or simply enforcing a policy that sensitive data can never be mounted to containers launched from these images.

TABLE IV. TOP-10 DEFAULT EXECUTABLES (FOR CONTAINERS WITH NO ENTRYPOINT STATEMENT)

Default startup	Number
/bin/bash	9723
/sbin/my_init	2111
run.sh	1244
bash	1015
/usr/bin/supervisord	866
/bin/sh	735
nginx -g daemon off	536
start.sh	482
/usr/sbin/sshd -D	365
/sbin/init	350
total	71528

Furthermore, when looking at images with no explicit entry point specified, we identify more opportunities for useful policy enforcement by Harbormaster. In particular, 73,386 (87%) of these images specify meaningful (non-shell) default programs in their `CMD` statement. Again, Harbormaster could be used to automatically enforce that only the specified default program may be run inside the container.

VI. CONCLUSIONS

Lightweight application containers such as Docker hold great promise for simplifying the development and deployment of cloud-based applications and making efficient use of cloud resources. But without a mechanism to constrain the sharing of resources on the target platforms, their adoption may be limited. Harbormaster takes the first steps toward defining such a mechanism, by allowing the definition of fine-grained policies that empower administrators to support multiple users while adhering to the principle of least privileges. In addition, policies that exercise fine-grained control over which images can be launched and what data they have access to helps administrators to secure their infrastructure and protect their sensitive data. Our prototype implementation demonstrates that Harbormaster's proxy-based architecture enables transparent integration with the Docker ecosystem and is able to enforce useful policies in a multi-user environment.

[3]Note that we also made other potentially threatening observations, such as 2280 application passwords in plain text hard-coded in the images, that could be addressed by Harbormaster, but are beyond the scope of our current prototype's policy language.

References

[1] Docker, “Build, Ship and Run Any App, Anywhere.” https://www.docker.com.

[2] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, “Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors,” in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys ’07. New York, NY, USA: ACM, 2007, pp. 275–287. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273025

[3] FreeBSD Handbook, “Jails.” http://bit.ly/1CSkzOq.

[4] OpenVZ, https://openvz.org.

[5] “LXC: Infrastructure for container projects,” https://linuxcontainers.org.

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” IBM, Tech. Rep., 2014.

[7] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments,” in *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, ser. PDP ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 233–240. [Online]. Available: http://dx.doi.org/10.1109/PDP.2013.41

[8] Y. Huang, A. Stavrou, A. K. Ghosh, and S. Jajodia, “Efficiently tracking application interactions using lightweight virtualization,” in *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, ser. VMSec ’08, 2008.

[9] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” Linux Journal, 2014.

[10] S. E. Hallyn and A. G. Morgan, “Linux capabilities: Making them work,” in *Linux Symposium*. Citeseer, 2008, p. 163.

[11] Google, “Kubernetes.” http://kubernetes.io/.

[12] Amazon, “AWS Elastic Beanstalk Documentation.” http://amzn.to/1FOPk99.

[13] W. Drewry, “dynamic seccomp policies (using bpf filters),” https://lwn.net/Articles/475019/.

[14] Docker, “Docker security: Linux kernel capabilities,” http://bit.ly/1d3gWQV.

[15] ——, “Advancing docker security: Docker 1.4.0 and 1.3.3 releases,” November 2014, https://blog.docker.com/2014/12/advancing-docker-security-docker-1-4-0-and-1-3-3-releases.

[16] ——, “Introducing docker content trust,” August 2015, https://blog.docker.com/2015/08/content-trust-docker-1-8/.

[17] ——, “Docker swarm,” https://docs.docker.com/swarm/.

[18] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *Cloud Computing, IEEE*, vol. 1, no. 3, pp. 81–84, Sept 2014.

[19] VMWare, “ESXi.” http://vmw.re/1GPwzrQ.

[20] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah, “First experiences using XACML for access control in distributed systems,” in *Proceedings of the 2003 ACM Workshop on XML Security*, ser. XMLSEC ’03. New York, NY, USA: ACM, 2003, pp. 25–37. [Online]. Available: http://doi.acm.org/10.1145/968559.968563

[21] Docker, “Docker registry API documentation,” https://docs.docker.com/reference/api/registry_api/.

[22] ——, “Docker hub public registry,” https://registry.hub.docker.com.