Docker-sec: A Fully Automated Container Security Enhancement Mechanism

Fotis Loukidis – Andreou, Ioannis Giannakopoulos, Katerina Doka and Nectarios Koziris
Computing Systems Laboratory, National Technical University of Athens, Greece
{flouk, ggian, katerina, nkoziris}@cslab.ece.ntua.gr

***Abstract*—The popularity of containers is constantly rising in the virtualization landscape, since they incur significantly less overhead than Virtual Machines, the traditional hypervisor-based counterparts, while enjoying better performance. However, containers pose significant security challenges due to their direct communication with the host kernel, allowing attackers to break into the host system and co-located containers more easily than Virtual Machines. Existing security hardening mechanisms are based on the enforcement of Mandatory Access Control rules, which exclusively allow specified, desired operations. However, these mechanisms entail explicit knowledge of the container functionality and behavior and require manual intervention and setup. To overcome these limitations, we present Docker-sec, a user-friendly mechanism for the protection of Docker containers throughout their lifetime via the enforcement of access policies that correspond to the anticipated (and legitimate) activity of the applications they enclose. Docker-sec employs two mechanisms: (a) Upon container creation, it constructs an initial, static set of access rules based on container configuration parameters; (b) During container runtime, the initial set is enhanced with additional rules that further restrict the container's capabilities, reflecting the actual application operations. Through a rich interaction with our system the audience will experience first-hand how Docker-sec can successfully protect containers from zero-day vulnerabilities in an automatic manner, with minimal overhead on the application performance.**

I. Introduction

In the last years, Cloud computing has prevailed over traditional on-premise environments as a means of executing applications and/or offering services for a wealth of reasons, including reduced costs, seemingly infinite resources purchased in a pay-as-you-go manner, scalability, ease of maintenance, etc. One of the key enablers of Cloud Computing is virtualization, since it can provide the necessary abstraction that allows multiple independent virtual systems to share the same pool of physical resources [1]. Recently, *containers* have gained ground as a lightweight virtualization solution that offers a plethora of benefits compared to Virtual Machines (VMs), the traditional hypervisor-based alternatives.

Most importantly, containers incur significantly less overhead than VMs, since they run as user-space processes on top of the kernel, which they share with the host machine. Moreover, they provide the ability to enclose application components in lightweight units, simplifying their distribution and deployment. As a result, large-scale applications can be managed in an automated manner.

As their popularity rises, containers have been successfully used in various use cases, while technologies around them enjoy active development [2], [3]. Despite that, a low adoption rate of container technology has been observed according to the Cloud Foundation [4] and many researches designate security concerns as a determining factor [5]. Indeed, containers were not designed with security in mind. Albeit providing isolation to certain resources such as processes, file system, network, etc. and enforcing quotas to CPU, RAM and disk usage, containers are much more prone to attacks compared to VMs due to the absence of a *hypervisor*, which adds an extra layer of isolation between the applications and the host. Since containers and host share the same kernel, compromised or malicious containers can more easily escape out of their environment and allow attacks on the host kernel.

The most effective way to harden the security of Linux containers is to enforce Mandatory Access Control (MAC) at the kernel level to prevent undesired operations both on the host and the container side, using tools like AppArmor [6] or SELinux [7]. However, this is a cumbersome process which requires knowledge of the characteristics and requirements of the application running inside the container and manual creation of the specific security rules to be applied. A recent attempt to automate the extraction of MAC rules [8] operates on a per image rather than a per container instance basis, leaving room for cross-container attacks.

To overcome these limitations we demonstrate *Docker-sec*, an open-source[1], automatic and user-friendly mechanism for securing Docker and generally OCI[2] compatible containers. Docker-sec protects containers from attacks against both the host and the container engine, restricting the container access to the resources that are truly necessary for the operation of the encompassed application. Docker-sec uses AppArmor to enforce access policies to all critical components of the Docker architecture by applying secure profiles to each of them. Container profiles are constructed based on (a) the *static analysis* of the container execution parameters and (b) the *dynamic monitoring* of the container behavior during runtime. More specifically, Docker-sec offers users the ability to automatically generate initial container profiles based on configuration parameters provided during container initialization (e.g., allowing only specific folders and files to be mounted). If a stricter security policy is required, Docker-sec can dynamically

[1]https://github.com/FotisLouk/docker-sec

[2]The Open Container Initiative (OCI) is a Linux Foundation project to design open standards for operating-system-level virtualization, most importantly Linux containers.

enhance the initial profile with rules extracted through the monitoring of real-time container execution during a pre-production period. By virtue of its two mechanisms, Docker-sec can protect containers since their very creation from zero-day vulnerabilities, incurring only a minimal overhead on the application performance.

Our demonstration of Docker-sec will showcase its ability to i) automatically derive initial access rules that restrict container capabilities to the very essential ones for its operation (via our static analysis mechanism) and ii) enhance the initial set of rules with additional ones that better reflect the enclosed application operations (via our dynamic monitoring mechanism). Both mechanisms will be demonstrated for Docker containers hosted in a private Opestack IaaS cluster. The participants will be able to interact with Docker-sec through an enhanced Docker Web Management UI, choosing from a number of different application containers and simulated attacks.

II. Docker-sec Architecture

Docker-sec is a container protection mechanism based on Docker, the most popular Linux container implementation, although it can easily be applied to any container abiding the OCI standard. In a nutshell, Docker uses a client-server architecture that consists of the *Docker client* and the *Docker host* (Figure 1). The Docker client is the user interface to Docker and interacts with the Docker host through the *Docker Engine*, a daemon responsible for building, running, and distributing the containers residing in the host machine. In order to manage the container's lifecycle, Docker Engine uses *containerd* a lightweight daemon that can handle multiple concurrent requests. Containerd, in turn, relies on *runC*, a CLI tool, to handle low-level container operations. RunC is usually executed by *containerd-shim*, a process which is used to manage headless containers.

The basic sandboxing mehcanism of Docker is Linux namespaces, which are able to virtualize and isolate various components of the system. However, in order to provide the required functionality, namespaces are usually tied to resources of the host system that cannot be virtualized. For example, although `mount` namespace offers the container a different view of the file system hierarchy, usually various essential file systems (such as cgroups and sysfs) are shared with the host. Through them, a container can access sensitive information and settings. Consequently, we need to identify the resources that Docker allows the container to access, determine the ones that are sensitive and protect them via Apparmor. It is also important to guard the processes through which these resources are assigned to containers, so as to allow only legitimate access to them.

Docker-sec adds an additional security layer on top of Docker's security defaults by automatically creating per-container AppArmor profiles. The system is protected from malicious or undermined containers that try to take control of the host or the containers running on it, since containers cannot communicate with other processes via signals, ptrace or D-Bus. Furthermore, Docker-sec enhances container security through generating dynamic security profiles, given an application workload. This way the privileges of a container, (e.g., capabilities, network access, etc.) are confined to the bare minimum that is needed for the specific workload. As a result, users of Docker-sec can gain the benefits of a MAC system, without having to deal with the complexity of maintaining it.

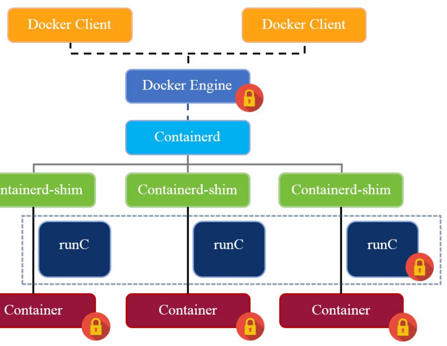

Fig. 1. Docker components protected with AppArmor in Docker-sec

Docker-sec creates secure AppArmor profiles for all Docker components that require protection to render the environment more secure. First of all, Docker-sec creates and enforces AppArmor profiles on containers, which serve as an entry point of an attacker, since they run arbitrary code and are accessible by users of the virtualized applications. The goal is to construct a separate profile per container, placing each one in a separate security context in order to restrict the sharing of resources among containers. Second, Docker-sec creates AppArmor profiles to protect `runC`, since it is the only process that can directly interact with containers via signals. Thus, Docker-sec is able to protect the entire process of launching the container, i.e., from the moment that `runC` starts initializing the container until it hands the control over to the process running inside it. Finally, Docker Engine is protected with a separate AppArmor profile, since users that can access it have full control over containers, images, volumes and network. The components of Docker that are automatically protected via AppArmor profiles via Docker-sec are designated with red locks in Figure 1.

►**Container Profile:** Container profiles are created using rules extracted from the configuration of each container and enhanced with rules based on the behavior of the contained application. To that end, Docker-sec employs two mechanisms: (a) *Static analysis*, which creates initial profiles from static Docker execution parameters and (b) *dynamic monitoring* which enhances them through monitoring the container workflow during a user-defined testing period.

The *Static Analysis* mechanism collects important static information about the container and its accesses. This information, which is either provided by the user as command line argumets or generated by Docker and obtained through Docker-specific commands, is used to derive initial security rules and construct the appropriate profiles under which the new container will be launched. More specifically, when the user executes `docker create` or `docker run`, commands with which the Docker Engine constructs the requested containers, Docker-sec collects from the command line arguments important information such as the container volumes, i.e., the files and folders of the host filesystem mapped to the container, as well as the container user, root or non-root, and the accompanying privileges. Moreover, through `docker info`, the command that displays system wide information regarding the Docker installation, Docker-sec obtains information like the ID of the container, which is a SHA256 checksum, and the mount point of the container's root file system. By knowing this information, Docker-sec can enforce runC to transition to a temporary AppArmor profile, which is designed for the initialization phase of the specific container. After this phase ends and before handing the control over to the container process, runC transitions to the AppArmor profile which will be used (and possibly enhanced by the Dynamic Monitoring mechanism) during the container's runtime.

Dynamic Monitoring allows the user to specify a training period for a specific container, during which Docker-sec will collect data about the behavior of the container. After initiating the training session, the user utilizes the part of the application she is interested in, making use of all the required application functionality, so that Docker-sec can determine the privileges (e.g., network access, file-system access, capabilities) that are necessary for the container to function properly. At the end of the training period Docker-sec analyzes the audit log that records the legitimate container accesses and adds the corresponding rules to the containers runtime profile, possibly discarding unnecessary privileges that were initially granted to it by the runtime profile generated by the static analysis phase. The training process can be repeated, if necessary, until all the required functionality is captured and imprinted in the container profile. Of course, during the training of the container runtime profile, it is important that only authorized and trusted users have access to the container and the container application. Otherwise, it is possible to record and grant access to system resources that are not needed by the container, undermining system security. It is worth noting, that while one container is under training mode, the rest of the containers are still protected.

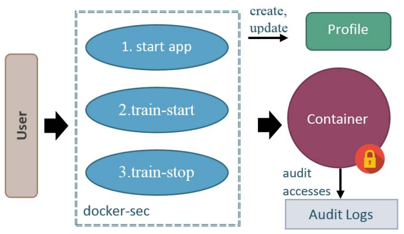

Fig. 2. Training process for container runtime profile

To provide the above functionality, Docker-sec uses AppArmor's capabilities for auditing certain accesses that are required by a process. AppArmor can set a profile in either *enforce mode*, where all the profile rules are enforced and no violations are allowed, or in a *complain mode* where violations of the rules are recorded but allowed for the execution of the corresponding system calls. In addition to the above, it is possible, through appropriate rules, to mix these two modes, providing greater flexibility. In particular, by maintaining a profile in enforce mode, we can choose to monitor and log the set of accesses governed by the rule, while the remaining rules of the profile continue to be enforced protecting the system. Therefore, by utilizing this capability, we can monitor the container's access to specific resources.

►**RunC Profile:** Since runC directly interacts with container processes through commands like `docker run`, `docker exec` or `docker stats`, we have opted for a separate AppArmor profile for it. The runC profile contains the appropriate rules, one per container, that allow runC to set each container's root mount point through the `pivot_root` system call and assign it a separate temporary profile. This temporary profile, used during the initialization of the specific container, protects the container until its transition (via `aa_change_profile` or `aa_change_onexec` functions) to the final container profile, used during the container runtime as described earlier.

Thus, Docker-sec protects the whole container lifecycle, starting from the runC profile, continuing with the temporary profile, during container initialization, and ending up with the final container profile, used during application runtime. As a result, access to the containers via signals or ptrace is allowed only to the legitimate procceses of the host, and most importantly, containers cannot access or control host processes via these mechanisms, minimizing the attack surface and protecting from a variety of attacks (e.g., CVE-2016-9962).

►**Docker Daemon Profile:** To protect the Docker Daemon, Docker-sec adopts a modified version of the AppArmor profile, available from the Docker github repository, which restricts access exclusively to the resources and tools/binaries (e.g. ps, cat, ls, etc.) that the Docker Engine requires for its operation.

III. Preliminary Performance Results

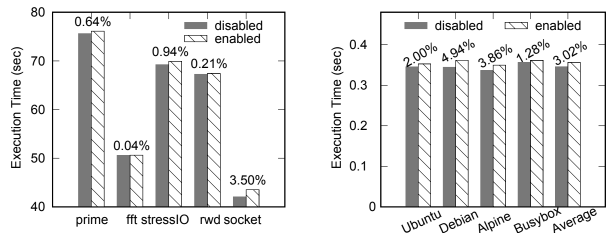

Fig. 3. Performance overhead of Docker-sec

We now evaluate the performance overhead introduced by Docker-sec during the launching of a container and the execution of the contained application due to the enforcement of the AppArmor profile. Our evaluation unfolds in two axes. First, utilizing *stress-ng*[3], a popular benchmarking tool used to stress a computer system, we execute different workloads and compare execution times, using an Ubuntu image drawn from its official Docker Hub registry[4]. We consider two types

[3]http://kernel.ubuntu.com/c̃king/stress-ng/

[4]Docker Hub is a cloud-based registry service for container image discovery, distribution and change management. https://hub.docker.com

of Docker containers: One that is secured with Docker-sec (referred to as "enabled") and one that runs without any security profile enabled (referred to as "disabled"). The selected workloads are i) *prime*, which calculates prime numbers, ii) *fft*, which executes the Fast Fourier Transformation, iii) *stress IO*, which executes sequential and random access reads/writes, iv) *rwd*, which reads, writes and deletes files and v) *socket*, which continuously opens and closes sockets. Second, we measure the time required for the container's bootstrap, using 5 different Docker images obtained from official repositories on Docker Hub. The choice of the specific images has been dictated by their bootstraping time: We chose images with small initialization time as our worst case scenario, so as to examine the maximum posssible relative overhead introduced by our mechanisms.

Our evaluation indicates that utilizing Docker-sec introduces a minimal overhead both during the container's lifetime (Figure 3, left) and during the container's bootstrapping (Figure 3, right). Specifically, in the former case, the observed overhead does not exceed 3.5%. For CPU-bound applications (e.g., *prime*, *fft*) the observed overhead is marginal, whereas benchmarks that stress file system resources (i.e., *stress I/O* and *rwd*) present slightly increased overhead that does not surpass 1%. Interestingly, the highest overhead is measured for the *socket* benchmark. This behavior is attributed to the fact that the enforcement of AppArmor rules in socket creation/destruction takes more time than in all other cases. Finally, when measuring the delay introduced in container bootstrapping for different Docker images, we notice that Docker-sec introduces a relatively constant overhead (between 2 – 4%) regardless of the image type.

IV. Demonstration Description

Docker-sec implements a command line interface similar to Docker, appending the suffix `-sec`, to the existing docker commands. Our automated system is based on *AppArmor* and a wrapper utility written in `bash`, which is responsible for creating AppArmor profiles tailored to specific container instances and for interacting with Docker Engine to perform the necessary operations in order to enforce them.

The attendees will interact with Docker-sec through a comprehensive, web-based GUI. The basic interaction dimensions comprise container creation, new AppArmor profile creation for a given container image, executing well known exploits and training arbitrary container images with different workloads.

Our demonstration covers two use cases. In the first scenario, the attendees will be able to verify Docker-sec's efficiency through building an enhanced security profile tailored to a specific container instance and attempting to exploit it. In the second scenario, the attendees will be able to create a new security profile using an arbitrary container running any given workload. More precisely:

►**Exploiting containers:** In the first use case, the user will be able to start a new WordPress container using the Docker-sec CLI. The container will be launched using the profile created through the static analysis mechanism. After initializing the container, the user will define a monitoring period and use the container through the WordPress UI. During this period, she will observe how the profile is being modified through the dynamic monitoring mechanism, which audits specific system resources, while protecting the rest of the system, since the static profile is still being enforced. When the training phase completes we will compare the static profile with the dynamic one to determine the exact privileges required by the specific application and to understand how Docker-sec restricts container access.

Next, the attendees will be able to attack the host system through an undermined container and compare the effects of the attacks when the container uses (a) no profile (i.e., totally unprotected container), (b) a vanilla AppArmor profile, (c) the profile created through the static analysis phase of Docker-sec, and (d) the profile created by both the static and the dynamic mechanisms of Docker-sec. In this step, after gaining access to a shell inside the container, the users will be able to "act maliciously" through the execution of various simulated attacks, like modifying the SSH daemon, installing new utilities inside the container or exploiting a vulnerability of the container engine (e.g., CVE-2016-9962).

►**Constructing new profiles:** In the second use case, the attendees will be given the opportunity to run Docker-sec for a variety of Docker images and enclosed workloads. They will be then able to compare the profiles generated for containers of identical images but different workloads executing within them. Through this process they will be able to discover the different set of privileges required for each container and how Docker-sec adapts to them. To that end, various benchmarks will be available, including heavy application loads or stressing of specific parts of a computer system, like CPU and I/O. Moreover, due to the variety of benchmarks, users can experience first-hand the overhead imposed by Docker-sec and AppArmor for various application types and assess its performance both in real life and extreme case scenarios.

Acknowledgement

This work has been supported by the European Commission in terms of the E2DATA H2020 ICT Project (780245).

References

[1] L. Vaquero *et al.*, "A Break in the Clouds: Towards a Cloud Definition," *ACM SIGCOMM*, vol. 39, no. 1, pp. 50–55, 2008.
[2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Queue*, vol. 14, no. 1, p. 10, 2016.
[3] S. Newman, *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.
[4] "Hope Versus Reality, One Year Later An Update on Containers," https://www.cloudfoundry.org/wp-content/uploads/2012/02/Container-Report-2017-1.pdf.
[5] "Portworx Annual Container Adoption Survey 2017," https://portworx.com/wp-content/uploads/2017/04/Portworx_Annual_Container_Adoption_Survey_2017_Report.pdf.
[6] "AppArmor," https://wiki.ubuntu.com/AppArmor.
[7] "SELinux," https://selinuxproject.org.
[8] M. Mattetti *et al.*, "Securing the Infrastructure and the Workloads of Linux Containers," in *IEEE CNS*, 2015, pp. 559–567.