Isolation in Docker through Layer Encryption

Ioannis Giannakopoulos, Konstantinos Papazafeiropoulos, Katerina Doka, Nectarios Koziris
Computing Systems Laboratory,
National Technical University of Athens, Greece
{ggian, kpapazaf, katerina, nkoziris}@cslab.ece.ntua.gr

***Abstract*—Containers are constantly gaining ground in the virtualization landscape as a lightweight and efficient alternative to hypervisor-based Virtual Machines, with Docker being the most successful representative. Docker relies on union-capable file systems, where any action performed to a base image is captured as a new file system layer. This strategy allows developers to easily pack applications into Docker image layers and distribute them via public registries. However, this image creation and distribution strategy does not protect sensitive data from malicious privileged users (e.g., registry administrator, cloud provider), since encryption is not natively supported. We propose and demonstrate a mechanism for secure Docker image manipulation throughout its life cycle: The creation, storage and usage of a Docker image is backed by a data-at-rest mechanism, which maintains sensitive data encrypted on disk and encrypts/decrypts them on-the-fly in order to preserve their confidentiality at all times, while the distribution and migration of images is enhanced with a mechanism that encrypts only specific layers of the file system that need to remain confidential and ensures that only legitimate key holders can decrypt them and reconstruct the original image. Through a rich interaction with our system the audience will experience first-hand how sensitive image data can be safely distributed and remain encrypted at the storage device throughout the container's lifetime, bearing only a marginal performance overhead.**

I. Introduction

The advent of virtualization technologies has been a key enabler of Cloud Computing, providing the necessary abstraction that allows multiple independent virtual systems to share the same pool of physical resources [1]. In the last years, *containers* have gained ground as a lightweight virtualization solution which acts at the operating system level, where multiple containers, i.e., isolated user-space processes, may run on top of the kernel shared with the host machine.

By virtue of their design, containers incur significantly less overhead than Virtual Machines (VMs) - the traditional hypervisor-based counterparts - while enjoying better performance, reaching that of native applications [2]. Thus, containers are ideal as application hosting environments in the fields of Cloud Computing and Software Engineering, with modern resource schedulers and Cloud-based IDEs currently supporting them [3][4][5][6].

Docker [7] is one of the most prominent implementations of Linux containers. Docker is designed on the principle that each container should execute a single application component; If an application consists of more than one components, different Docker containers should be allocated.

Docker relies upon union-capable file systems. This means that a container image, which is used to instantiate containers, consists of a series of layers on top of a base OS image, so that each layer contains only the updates to the previous one. Thus, when making a change to a container (e.g., installing a new software package, etc.), the additional content is written on a new layer, created on top of the existing ones (which remain read-only). This mechanism facilitates the creation of new Docker images, since they can easily be built upon other existing Docker images, available in online repositories. Thus, an image and its derivatives share the same base layers (e.g., Ubuntu OS) and differ only in the topmost layers, which represent the additional files stored (e.g., MySQL installation).

The aforementioned property has given birth to a delicate and efficient distribution mechanism which reduces the sharing of images to exclusively sharing the specific image layers missing (e.g., the MySQL installation layer). This is particularly useful for Continuous Delivery and other DevOps tasks that require automation, since Docker layers can be stored in a central repository, i.e., the Docker Image Registry [8], and pulled by different clients concurrently. Furthermore, since each update can be formulated as a new layer, changing a Docker container is straightforward and completely safe: An unstable or faulty image layer can be easily removed, leaving a completely functional container.

Docker Image Registries support public as well as private images: The former are accessible by any user whereas the latter are only accessible by users with specific privileges. In both cases though, privileged users, such as the registry's administrator or people with physical access to the registry host, can obtain access to the images, since they are not encrypted by the client. Therefore, users distributing images with sensitive data do not rely on the registry: They either (a) commit and push to the registry new images *without* the sensitive data, which they add manually when the destination container is launched, or they (b) extract and encrypt the entire image, forfeiting the layered image philosophy and failing to fully exploit its advantages. Indeed, although the latter solution guarantees that all image layers are retrievable when the decrypted image is inserted into a new Docker host, decryption is a cumbersome process, and thus the overhead of decrypting the whole image for just a handful of sensitive files is prohibitively high.

Furthermore, since Docker stores the container's data in plain text format in the host file system, data confidentiality may be compromised by a malicious storage provider who has physical access to the storage medium or privileged (e.g., root equivalent) access to the host machine.

To overcome these limitations, in this work we propose and demonstrate *IDLE* (**I**solation in **D**ocker through **L**ayer **E**ncryption), a tool that runs on top of Docker and allows the user to extract and encrypt a specific image layer, transfer it to the destination host and decrypt in a secure way, ensuring that even if the protected layer becomes publicly available, only the key-holder can obtain access to it. Furthermore, to protect the confidentiality of already shared and deployed Docker images against a malicious storage provider and ensure that sensitive data remain encrypted at the storage device they belong to throughout the container's lifetime, we enhance the Docker storage engine with a mechanism that allows access to stored container data an "encryption/decryption" mapping that translates the operations performed over the original data into operations over encrypted data. The experimental evaluation of our system's prototype demonstrates that our mechanism causes marginal overhead to the container's performance.

Our demonstration of the IDLE system will showcase its ability to i) securely distribute/migrate Docker images that contain sensitive data (via our *IDLE-on-the-move* mechanism) and ii) preserve the confidentiality of sensitive data stored on disk even during container usage (via our *IDLE-at-rest* mechanism). The demonstration platform will showcase both IDLE mechanisms for Docker containers hosted in a private Openstack IaaS cluster. The participants will have the opportunity to interact with IDLE through an enhanced Docker Web Management UI, controlling the amount and type of data that will be securely transferred between Docker hosts and enforcing the IDLE mechanisms on top of dynamically allocated Openstack volumes.

II. IDLE System Architecture

The IDLE system architecture consists of two modules, the *IDLE-on-the-move* and the *IDLE-at-rest* modules. The former implements the mechanisms that handle the secure distribution/migration of container images with sensitive data among hosts, while the latter is responsible for ensuring the confidentiality of stored data throughout the container's lifetime. Both modules rely on the capabilities offered by the underlying *union* file system - OverlayFS [9] in our case - which supports the Docker layered image architecture.

Layered images allow the user to build her images in an incremental manner, with each extra layer adding a new feature to the existing ones. Assuming a single-layer base image (e.g., Ubuntu 14.04), a user can boot it, execute a set of commands (e.g., install java) and finally commit her updates. This is performed by creating a new image layer that contains exclusively the newly appended/updated files and lies on top of the base image. Upon further updates (e.g., tomcat installation), additional new layers are created and placed on top of the previous ones following the same process. The final image will be of the form depicted in Fig. 1(a). We should note at this point that upon instantiation of a container, Docker by default creates a new layer on top of the existing image layers so as to store any future changes. This top layer is mounted

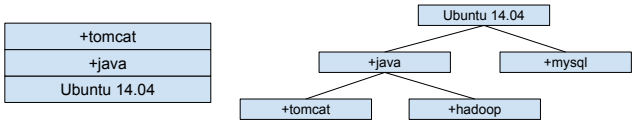

Fig. 1. Docker (a) image example and (b) layer tree for different images

with read/write permissions, unlike the rest of the image layers which remain read-only.

This layered structure results in the creation of a tree-hierarchy among different image layers, as depicted in Fig. 1(b). In this example, existing image layers (e.g., Ubuntu 14.04, +java) can serve as a basis for the generation of new container images (e.g., Ubuntu 14.04, +java, +hadoop). This scheme encourages the re-usability of the various layers and enables Docker's lightweight images, since only layer updates need to be propagated. Docker layers can be stored in central image registries and made available for downloading by different clients. Thus, a user in possession of a base image (e.g., Ubuntu 14.04) that wants to add an extra application (e.g., MySQL) does not need to pull a whole new image from the registry, but rather download the desired, application-specific layer, as opposed to the monolithic VM images, which need to contain the whole underlying software stack even for a minor update.

IDLE-on-the-move module: This module encrypts and distributes the sensitive layers of a Docker image. Figure 2 depicts the proposed secure image distribution mechanism. We assume a container that consists of multiple layers (e.g., 7 layers, according to the figure) with the topmost layer (Layer 6) containing the confidential data.

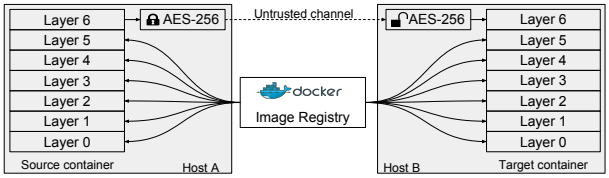

Fig. 2. Container migration scheme

As depicted in the figure, we decompose the container's image in two parts: The public part – formed by Layers 0-5 – and the private part that consists of Layer 6. The public part of the image is pushed to the Docker image registry as usual. The private part is not shared via the image registry, but the user can distribute it with a tool of their choice (e.g., through a public Web Server). The private image layer is encrypted using AES-256 and additionally signed with the ECDSA algorithm. The encrypted layer is transferred to the destination host. Our mechanism pulls the public part of the image and instantiates a new container; The private part is, in turn, decrypted and installed as the top layer of the newly allocated container.

To enforce the per-layer encryption, we have implemented a tool which orchestrates the previously described procedure. First, the user provides the container id she wishes to encrypt. For simplicity, we assume that only the topmost layer is to be encrypted. To identify which is the topmost layer for the specified container, we first parse the configuration file located in

`$DOCKER_PATH/image/overlay/layerdb/mounts/<container id>/init-id`, in which the ID of the topmost layer is stored. Then, the `layer_id` value is retrieved and the final layer path is identified, which is `$DOCKER_PATH/overlay/<layer id>/upper`. Afterwards, we create an archive with the data of the topmost layer and remove it from the container image. The archive contains the private layer's files and meta-files used internally by overlay, denoting file deletions, versioning, etc. The public part of the image is then pushed into the image registry. After being encrypted and signed, the encrypted layer is transferred to the destination host.

At the destination, the reverse procedure is followed: Upon successful validation of its signature, the private image layer is extracted from its archive. The public part of the image that must be pulled from the registry is designated by the id stored in the metadata file, included in the private part's archive. After downloading the public image, a new container is instantiated and the extracted private layer is placed in the topmost layer. At the end, the image is securely transferred from the source to the destination host, maintaining the confidential data encrypted throughout the migration procedure. Finally, we should note that during the aforementioned procedure, Docker remains agnostic of the described operations. This modularity means that our approach does not need sophisticated setups. On the contrary, it can operate in any Docker installation, assuming that it supports Overlay. One of our future targets is to port our methodology to different file systems.

IDLE-at-rest module: We now discuss the utilization of a data-at-rest encryption mechanism in order to save the image layer with the confidential data in an encrypted manner to the disk, through dm-crypt [10]. The main idea behind dm-crypt is that the data always remain encrypted on the disk, e.g., in the directory `/tmp/encrypted`, while a file system mapping allows on-the-fly data decryption. This mapping is accessible by the users as a directory in the filesystem, e.g., the directory `/tmp/unencrypted`. Every time a new file operation is issued on a file under the unencrypted directory, dm-crypt translates it into an operation on the encrypted directory and the encryption/decryption operations are executed on the fly. Obviously, a root user is able to access the data when the mapping is enabled, i.e., when the encrypted directory is mounted. However, when the directory is not mounted the encrypted data remain inaccessible even for privileged users.

As mentioned above, this mechanism acts as an extra precaution measure to enhance the protection of confidential data against an adversary that has access to the storage service over which the container is executed. Taking into consideration that nowadays a common practice is to deploy Docker containers over a cloud infrastructure, it becomes apparent that the confidential container data could be stored over a cloud storage stack that consists of multiple layers and, possibly, non-trusted parties. The ability to maintain data confidentiality in such cases becomes a highly desirable feature which resembles the ability to boot a VM from an encrypted disk. Our tool manages at-rest Docker layer encryption through dm-crypt transparently for the user. In order to facilitate the cloud deployment, we provide a component able to leverage inherent mechanisms of different Cloud Computing platforms. Using an easily extensible implementation, IDLE-at-rest currently supports dm-crypt mappings based on: a) a file on the local filesystem or b) an Openstack Volume, the lifetime of which is managed by our module in an automated way.

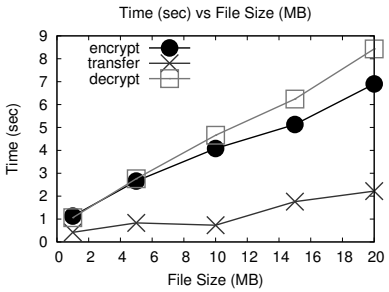

Fig. 3. Experimental Evaluation

IDLE overhead: We now evaluate the overhead inserted by the utilization of the IDLE on-the-move system, in order to securely transfer a container layer from a source to a destination Docker host. The key parameter that affects the execution time of our approach is the size of the data to be encrypted; To this end, we created a Docker container based on a vanilla Debian image, and populated the topmost layer with files of different sizes, varying from 1 to 20MB. For each size, we encrypted the topmost layer, transferred it to a destination Docker host and installed it into a new Debian-based container, measuring the time for each phase separately. In Fig. 3 we depict our findings. The experiments were conducted in a private Openstack installation and the Docker hosts (the source and the destination) run Ubuntu 16.04 with 1-core CPU and 1GB of RAM each. Each experiment was repeated 5 times and the presented times correspond to the average of those runs. The figure indicates that the time needed to encrypt and decrypt the image layer increases linearly with the size of the layer, a reasonable finding since the execution time is dominated by the encryption algorithm. Moreover, it is obvious that for a layer size of 10MB, the overhead inserted by our approach does not surpass 4 seconds; A time which is marginal when compared to the time needed to fetch the public part of an image from a Docker Image registry, that, according to the image size, may require several minutes.

III. Demonstration Description

Our system is controlled by a comprehensive web-based GUI that attendees will utilize. The basic interaction dimensions include container selection, authentication token input, inspection of container's status and verification of container's confidentiality. The GUI allows the user to manage the Docker hosts as well as the two IDLE mechanisms.

Use case scenarios We consider two pre-defined, common real world use cases that showcase the benefits of the proposed approach. Apart from those, the user will have the opportunity

to construct her own scenario, selecting among a plethora of pre-deployed Docker containers. More precisely:

►**Credential protection**: We assume a three tier application that consists of a Web Server that renders and serves web pages, an Application Server that implements the business logic of the application and a Database Server, each of which is deployed in a dedicated container, according to Docker's philosophy. The three containers communicate with each other through a user-management protocol. A common practice for the deployment of those components is to add authorization information inside the configuration file of each of them, which will later be used by the servers to establish connections with each other. For example, to the Application Server container one would append a few lines to the appropriate configuration file regarding the host, port and a username/password pair pointing to the Database Server. This file is confidential since the information it contains should not be publicized. The same applies to various other sensitive information: Secret API tokens, passwords, private SSH keys, etc. must be included in a container's image in order to be able to reach external services. This information is, of course, not publishable.

►**Log protection**: The second use case is that of a service that produces logs which imply user activity. Assuming the three tier application as above, the Web Server generates logs that relate the IP address of a client, the timestamp of her visit, the pages that she visited etc. This information should be considered confidential and thus not be made freely available.

►**User-defined**: The user will be able to create her own custom use-case scenario, by selecting from a list of available container images and incorporating sensitive information of her choice.

Actions During the demonstration, the users will be able to interact with IDLE through an intuitive user interface and perform the following actions: (a) secure container migration between two Docker hosts and (b) encryption enforcement of existing Docker containers, utilizing storage volumes dynamically allocated from an Opestack IaaS Cluster.

Regarding the former action, which demonstrates the functionality of the IDLE-on-the-move component, the user will be able to select a specific container based on a list of pre-deployed containers running in an existing Docker host. Upon selection, the user will be able to extract the topmost layer, according to the discussed methodology, entering her passphrase for the symmetric key encryption and optionally providing an ECDSA key, utilized for signing the encrypted layer. The encrypted layer will be, then, downloaded to the client machine. With the reverse procedure, the user will be able to upload the encrypted layer into a separate Docker host, trough the same UI; After uploading it, the system will ask for a verification key and the secret key which was utilized for the encryption. Finally, upon successful decryption, the thin layer of the target container will be populated with the data from the source host, and the container will launch. The user will be able to verify the entire process through IDLE's UI, in which she will be able to attach a console for both the source and the destination containers. A screenshot of IDLE's UI is depicted in Figure 4. We should note that IDLE's functionalities are exposed through a UI based on the open source Weave Scope project [11], which is one of the most prominent tools used for the management of Docker containers. Through it, the user can easily monitor and manage her containers.

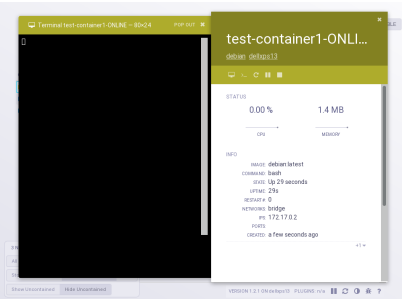

Fig. 4. IDLE's UI

The latter action, which demonstrates the IDLE-at-rest module functionality, will be showcased in a similar manner. The user will be able to initiate the at-rest encryption enforcement through the User Interface, providing the necessary passphrase for the encryption. In our demonstration, the encrypted data will lie on top of dedicated Openstack volumes, generated when the at-rest encryption process initiates. The user is responsible to provide the necessary user authentication tokens, such as usernames, passwords, endpoints, etc., in order to be authenticated with the cloud platform of their choice. After the authentication, the user will be able to inspect the encrypted layer which will be detached from the Docker host when the container is in stopped state and it will be re-attached when the container runs. Again, all the procedure will be verifiable through the container console, launched through IDLE's UI as indicated in Figure 4.

ACKNOWLEDGMENT

This work was supported by the TREDISEC project (G.A. no 644412), funded by the European Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme.

REFERENCES

[1] L. M. Vaquero *et al.*, "A break in the clouds: towards a cloud definition," *ACM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2008.
[2] W. Felter *et al.*, "An updated performance comparison of virtual machines and linux containers," in *ISPASS*. IEEE, 2015, pp. 171–172.
[3] V. K. Vavilapalli *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of SoCC*. ACM, 2013, p. 5.
[4] B. Hindman *et al.*, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." in *NSDI*, vol. 11, 2011, pp. 22–22.
[5] A. Verma *et al.*, "Large-scale cluster management at Google with Borg," in *Proceedings of Eurosys*. ACM, 2015, p. 18.
[6] "Eclipse Che," https://www.eclipse.org/che/.
[7] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.
[8] "Docker Hub," https://hub.docker.com/.
[9] "OverlayFS," https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.
[10] "dm-crypt," https://www.kernel.org/doc/Documentation/device-mapper/dm-crypt.txt.
[11] "Weave," https://www.weave.works/products/weave-scope/.