DOCKER SECURITY IN WEB SIMULATION TOOLS: A LAYERED APPROACH

[1]MARCOS BARRANQUERO, [2]JUAN CASADO, [3]JOSEFA GÓMEZ, [4]ABDELHAMID TAYEBI, [5]JOSÉ A. JIMÉNEZ

[1,3,4]Computer Science Department, University of Alcalá, Alcalá de Henares, Spain
[2]Starleaf, Building 7, Hatters Lane Watford WD18 8YN, United Kingdom
[5]Department of Electronics, University of Alcalá, Alcalá de Henares, Spain
E-mail: [1]marcos.barranquero@edu.uah.es, [2]juan.ballesteros@starleaf.com, [3]josefa.gomezp@uah.es, [4]hamid.tayebi@uah.es, [5]jose.jimenez@uah.es

Abstract - In this work, container security issues and strengths are studied using Docker as the main implementation. First, the container infrastructure is described and compared against the traditional approach of virtual machines. Secondly, the Docker containers security is discussed by the different infrastructure layers that compose them and different solutions are proposed to try to decrease the attack surface over this kind of applications.

Keywords - Container Security, Cybersecurity, Docker, Docker Cybersecurity.

I. INTRODUCTION

In the last years, the popularity of containers has increased due to necessities such as the need of a more agile application deployment or one that provides better performance needs compared against other alternatives like VMs (Virtual Machines). When it comes to security, as other papers have studied [1], container applications have a wide attack surface, that ranges from the container image and its possible vulnerabilities, to the container daemon [2], including the applications and processes being executed in the container and the host that runs it. Recent years have seen a huge increase in the development and use of this technologies, as it can be seen in [3]. Alongside this increase, security concern towards the deployment and usage in production of containerized applications has increased too, leading to multiple security studies with different approaches and focuses, like from a Platform-as-a-Service point of view [4] or providing a framework and metrics like [5]. The aim of this work is to provide a structured guide of security concerns and good practices for the reader interested in deploying a secure dockerized application with safety. For this task, a set of good practices and resources are listed in this work, classified by layers. On section II, the evolution leading from VM to containers is briefly discussed. Section III describes the docker container infrastructure, regarding the docker daemon and host. Section IV presents different security concerns, examples and consideration related to each level of the docker infrastructure. The conclusions and future work are included in Section V.

II. FROM VIRTUAL MACHINES TO CONTAINERS

Traditionally, virtual machines have been used for the purpose of emulating the hardware and software of a real machine. Each virtual machine constitutes a piece of software that emulates a real machine's hardware. It uses the host real hardware to simulate an environment exactly identical to a real machine with the designated operating system, and other programs already installed and ready to use. Virtual machines are used in multiple use cases: to simulate a multiple-machine interconnected infrastructure to provide a service, to execute software from an isolated and sandboxed perspective, or to provide compatibility with different software that cannot be run on the host machine operating system.

However, VMs have some drawbacks:

- It takes a lot of resources from the host machine in a blocking form in most of the VM clients. For example, the RAM or storage will be designated before launching the virtual machine, and will be a fixed value even though not all the RAM or storage is being actually used.
- VMs are less efficient than real hardware, since they are accessing the resources in an indirect way or simulating them by software.
- VM's portability is limited and difficult: sharing a VM image involves large files and usually includes vendor data [6]. Additionally, environment replication is difficult to manage, although there are paid applications that can do it.

Containers came as a solution to these problems, looking to provide better performance, decrease the storage and power usage, process isolation and easier portability.

A container is a software unit that contains one or more applications and all the requirements and libraries needed to execute them. It is a lightweight and isolated package that is ensured to work with independence from the platform where it is executed.

Originally, Docker ran over Linux containers, known as LXC, but then moved to libcontainer, running in the same OS (Operating System) as the host machine. This allows containers to share most of the host operating system resources and run using the host kernel, providing a more efficient approach than the virtual machines, especially when running multiple virtualized services.

Docker provides additional features over LXC or libcontainer, like the automatic build feature that allows developers to define the commands to be executed when the container is launched, or the possibility to share different container images over the Docker registry.

In the end, the weight of the advantages and disadvantages of virtual machines and containers is determined by the use case. For isolated multiple process that are deployed as microservices, containers should be the initial choice. Dynamic resource allocation is possible in containers and on VMs.

Some orchestrators, like Kubernetes, provide dynamic control on the resource assignation. However, if a process needs complete isolation and guaranteed resources, a VM may be the best solution.

III. DOCKER CONTAINER ARCHITECTURE

Docker is composed by three main parts: the docker client, the Docker host and the Docker registry.

The Docker client is the interface through which the developer can interact with the docker host. When a command like docker run is executed, the Docker client communicates to the docker daemon using the Docker API. This allows to deploy an environment where the Docker client is separated from the Docker host. This approach could be useful to have more control and flexibility over Docker clients, being able to monitor them from the interface and allocating more or less resources depending on the use.

The docker registry is a storage and distribution system for different Docker images. It allows the developers to push and pull images, working as a repository. By default, the Docker registry used is Docker Hub, a repository of public and private container images.

A Docker image is a binary file that includes all of the requirements for running a Docker container: stores the dependencies, tools, libraries and source code needed for an application to run. The image works as a template, similar to a snapshot for a virtual machine. The Docker image is usually divided in layers, where the first layer contains the base image of an operating system, and the container layers that describe the commands and executable files to be executed.

IV. DOCKER SECURITY BY LEVELS

4.1. Host Level

The host machine is where the docker daemon and the containers run. It is important to configure and harden the operating system of the host in order to secure it against possible attackers on a production environment.

It is also important to configure docker properly. Some good practices are:

- Docker containers should be run with the least privilege possible. By default, Docker requires root permissions to be executed, so a good practice is to add the user to the docker group.
- Docker has a feature that allows to add and remove capabilities to the containers, similarly to SecComp which is discussed later. Only the needed capabilities should be used in order to reduce the attack surface of the deployed container.
- By default, containers are allowed to escalate privileges when required. There is an optional security policy that denies the possibility to acquire new privileges once the container is running.
- Docker installation files should be secured. It is a good idea to review and restrict the file permissions and verify that the owner is the root user.
- Whenever possible, latest software versions should be used, since the latest version will have most of the known vulnerabilities patched. This goes for the docker package, the host operating system or mostly every software layer that interacts with the containers.

4.2. Application Level

With regard to application development, the design must integrate security by default applying the different known principles of secure development, without neglecting those sections that make the container interact with the outside world: input verification, secure APIs, etc.

An interesting approach is the distroless images [7] that, excluding the operating system, seek to include only applications and their runtime dependencies. They have neither a shell nor a package manager, nor the vast majority of packages that are usually included by default in Linux distributions.

This allows to deploy debloated containers, which do not contain anything installed beyond what is necessary, significantly reducing the attack surface and therefore providing a hardened container.

4.3. Container Operating System Level

Different hardening techniques have been suggested in papers like [8], where hardening tools and implementations are used. An overview of these tools would be described next.

1) SELinux: As described in [9], SELinux is a security architecture for Linux systems that allows administrators to have more control over who can access the system, and defines access controls for the applications, processes, and files on a system.

SELinux has different running modes. Enforcing, which is the default mode, will to enforce security policies over the application requests.

On the other hand, at the moment of running a container, Docker supports different options such as running a mounted volume over a host directory. Even though the container image has SELinux in enforcing mode, since the volume is shared between the host and the container, the host files will be accessible from the container and vice versa.

This is due to the fact that the Docker daemon has SELinux disabled by default. It is possible to enable it by overwriting the daemon settings with a new configuration file, as instructed in [10]:

```
[root@marcos]$ docker info | grep Security -A3
Security Options:
  seccomp
     Profile: default
# SELinux is not enabled.
[root@marcos]$ cat /etc/docker/daemon.json
{
      "selinux-enabled": true
}
# Docker service must be restarted
[root@marcos]$ systemctl restart docker
[root@marcos]$ docker info | grep Security -A3
  Security Options:
     seccomp
       Profile: default
     selinux # Selinux is enabled now
```

With this configuration it is still possible to mount and access the volume files, but the unmounted host files will not be writable or readable from the container anymore.

2) AppArmor: AppArmor is another Linux security module that restrains the access and permissions of applications, similarly to SELinux. However, AppArmor allows to define different security configurations for each program.

Docker allows to run containers loading different AppArmor profiles with the running command. By default, it runs the dockerdefault policy profile.

AppArmor profiles can be very flexible: the profiles use a globbing syntax that allows to define rules to accept or deny network traffic by protocol or IP. They can also define the directories that are writable or mountable, and allow or deny certain capabilities.

3) SecComp: SecComp is a kernel module that provides additional security with different profiles. Unlike AppArmor or SELinux,

SecComp allows to define profiles that limit the system calls and allow to manage the available call from within the Docker containers to the host's kernel.

Once again, if there is no SecComp profile specified, Docker will run the default profile. As it can be read in [11], by default, the SecComp profile limits system calls like the CAP SYS BOOT reboot system call, that would allow the containers to reboot the host.

However, in a production environment, maybe it is interesting to allow or deny certain system calls that could interfere within the service continuity, blocking commands like chmod or mkdir that contains potentially dangerous system calls.

The SecComp filters are written in a JSON file format, and loaded at the time the container is launched.
A simple example of applying a SecComp profile the "hello-world" image would be:

```
[root@marcos]$ cat chmod.json
{
    "defaultAction":"SCMP_ACT_ALLOW",
    "syscalls":[
         {
"name":"chmod",
"action":"SCMP_ACT_ERRNO"         }    ] }
[root@marcos]$ docker run hello-world
--security-opt seccomp:chmod.json
```

For this simple example, the profile works as a blacklist: the default action for any system call is to allow it with the SCMP ACT ALLOW tag. But for the chmod call, the action to be taken is to deny the call, with the SCMP ACT ERRNO tag.

4.4. Communication Level

If a Docker container is deployed with the client and the daemon running on different machines, it would be desirable to secure the docker API communication with TLS or SSH [12]. In addition, it is possible to run the docker daemon and client in different modes, where the client and the host authenticate each other.

4.5. Image and Registry Level

As it is shown in [13], the images of the most popular docker registry, Dockerhub, do not always provide adequate security for the deployment. Recently, Docker has included a vulnerability scanning tool built into the docker client and Dockerhub, which allows to identify potential vulnerabilities in the container's images. Additionally, there are external tools such as Snyk [14], which can scan and monitor container images at different stages of the deployment. The tool allows the user to scan a Dockerfile, a Git repository or a Docker image, looking for potential vulnerabilities like outdated dependencies or configuration vulnerabilities, and presenting alternatives and suggestions on to fix them. A good practice to keep in mind is the use of multi-staged builds: a way to build the container by selecting to load only specific elements from several different previously built images. This creates a small image with just the commands and dependencies to run, reducing the attack surface and providing flexibility in development and deployment. This could be useful, for example, in a use case where there are two container images: a large one with the SDK and the needed compilations tools to compile the source code of an application, and a small base image with only the needed runtime dependencies for running the compiled application, producing a smaller final image.

V. CONCLUSIONS

As the use of containers and Docker grows, concerns about container security increase. It is difficult to maintain an adequate level of security while keeping pace with software updates and use. More research and dissemination should be done on the different hardening techniques in order to increase the average level of safety. To do this, future work will study more methods and approaches to add security by default at different layers and levels, as well as performing security verification of container images and related software or real-time protection and integration with security systems.

ACKNOWLEDGMENTS

This work was supported by the University of Alcala through project CCG20/IA-045.

The authors would like to thank professor Javier Junquera Sánchez for providing multiple resources and different points of view on the matter of adding security to docker.

REFERENCES

[1] T. Combe, A. Martin, R. Di Pietro, "To Docker or Not to Docker: A Security Perspective", IEEE Cloud Computing, vol. 3, no. 5, pp. 54-62, Sept.-Oct. 2016.

[2] S. Sultan , I. Ahmad, T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead", IEEE Access, vol. 7, pp. 5297-52996, 17 April 2019.

[3] Sysig, "Container usage report of 2019". [Online]. Available from: https://sysdig.com/blog/sysdig-2019-container-usage-report/.

[4] A. R. Manu, J. K. Patel, S. Akhtar, V. K. Agrawal and K. N. B. Subramanya, "A study, analysis and deep dive on cloud PAAS security in terms of Docker container security", 2016 International Conference on Circuit, Power and Computing Technologies, pp. 1-13, 2016, doi: 10.1109/ICCPCT.2016.7530284.

[5] H. Jin, Z. Li, D. Zou and B. Yuan, "DSEOM: A Framework for Dynamic Security Evaluation and Optimization of MTD in Container-Based Cloud", IEEE Transactions on Dependable and Secure Computing, vol. 18, no. 3, pp. 1125-1136, 1 May-June 2021, doi:10.1109/TDSC.2019.2916666.

[6] D. Kargatzis, S. Sotiriadis and E. G. M. Petrakis, "Virtual machine migration in heterogeneous clouds: from openstack to VMWare", 2017 IEEE 38th Sarnoff Symposium, pp. 1-6, 2017, doi: 10.1109/SARNOF.2017.8080393.

[7] GoogleContainerTools Organization, "Distroless Docker Images". [Online]. Available from: https://github.com/GoogleContainerTools/distroless.

[8] Amith Raj MP, A. Kumar, S. J. Pai and A. Gopal, "Enhancing security of Docker using Linux hardening techniques," 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), pp. 94-99, 2016, doi: 10.1109/ICATCCT.2016.7911971.

[9] RedHat, "What is SELinux?". [Online]. Available from: https://www.redhat.com/en/topics/linux/what-is-selinux.

[10] Docker, "Docker daemon documentation". [Online]. Available from: https://docs.docker.com/engine/reference/commandline/dockerd/.

[11] Docker, "Seccomp security profiles for Docker". [Online]. Available from: https://docs.docker.com/engine/security/seccomp/.

[12] Docker, "Protect the Docker daemon socket". [Online]. Available from: https://docs.docker.com/engine/security/protect-access/

[13] Snyk, "Top ten most popular docker images each contain at least 30 vulnerabilities". [Online]. Available from: https://snyk.io/blog/top-tenmost-popular-docker-images-each-contain-at-least-30-vulnerabilities/

[14] Snyk, "Snyk open source security management tool". [Online]. Available from: https://snyk.io/product/open-source-security-management/

★★★