Rootless containers

Aleksi Hirvensalo
aleksi.hirvensalo@aalto.fi

Tutor: Mario Di Francesco

Abstract

Containers are used to isolate applications at the operating system level. However, that isolation can be escaped and currently the threat is amplified by how containers are being run. Most containers run as root by default, which means that if they can escape the isolation, they are a root user of the host machine. To tackle the issue, rootless containers have been created. Rootless means to run the container and its runtime as non-root.

This paper introduces rootless containers and gives a general overview of the topic. Furthermore, a practical example is given on how to attack poorly configured containers and how to prevent the attack by enabling rootless containers.

KEYWORDS: container, container security, docker, rootless containers, sandboxing

1 Introduction

Containers realize application virtualization at the operating system level [6]. In particular, containers provide a lightweight mechanism to ensure isolation, allowing multiple applications to run on the same host. They also enable a specific application to always run the same, regardless of the infrastructure. On a higher level containers do offer the same kind of flexibility in deploying and running applications as virtual machines [19]. However, containers are different in a number of ways.

Virtual machines are an abstraction of the hardware whereas containers are an abstraction at the application layer [19]. In fact, containers share the kernel of the host which allows for multiple containers to run on the same operating system. By sharing the kernel, containers can use the same resources and functions as the host machine, for example, the filesystem and networking. However, the visibility and accessibility of these resources are controlled. This is how the aforementioned lightweight isolation is achieved.

Before defining containers more in depth, it must be noted that the focus of this paper is Linux containers. This simplification is made as most often containers run on a Linux environment [10]. Containers are essentially a technology that uses features of the Linux kernel, mainly control groups and namespaces. Containers are nothing more than processes running on a Linux operating system. Control groups limit the resources that a certain process is assigned to, for example, memory and CPU [10]. Namespaces limit the visibility of operating system resources for a given process [10]. These resources can for example be file system structure and process.

Containers have rapidly become popular in software companies due to their benefits in performance compared to other means of virtualization [12]. Containers allow for agile development and efficient operations [18] and have been largely adopted in DevOps practices and building microservices. One of the most popular tools for containers is Docker [1]. an open-source project that can be run on Linux, macOS and Windows. On Linux and macOS, Docker is mainly used through VMs. Moreover, on Windows the user can choose between Windows containers or Linux containers through Docker desktop [8]. Docker is an open platform that can be used throughout softwares lifecycle e.g. to develop, to deploy and to run the software [5]. Docker also enables users to separate the applications from the infrastructure, allowing quicker software delivery.

As new technologies are developed and introduced, new security concerns and vulnerabilities often arise. Containers are no exception [10]. Compared to virtual machines, containers provide a weaker level of isolation, as containers share the same kernel as the host machine. A stronger level of isolation is preferred because it mitigates the risk of malicious applications interacting with the host or other containers. This paper focuses on rootless containers, a method to mitigate the aforementioned threat. The goal is to introduce rootless containers to the reader and give a technical overview of the topic. Also, a tutorial is given on how to use rootless containers with Docker.

The paper starts with the background, where containers are described and a motivation for rootless containers is given. Then it covers how rootless containers are made and how to enable them in practice. Finally, sandboxing is briefly covered followed by concluding remarks.

2 Background

Operating containers involves three important elements, the container runtime, the container image and the container itself. The container runtime takes care of running the containers, it makes sure that the container is isolated accordingly and necessary resources are created for the container to use, for example the filesystem and the container user are configured. The runtime is able to create the needed resources by instructions that the container image provides. In fact, the container image's only responsibility is to describe how to run the container i.e. the container environment. Regarding Docker, the container runtime is called Docker daemon which is responsible for building, running and distributing containers [5]. Docker registry takes care of storing the images. A Docker registry takes care of storing the images; Docker Hub is a public registry, used by default [4].

In general, most containers run as root, which corresponds to the host machine's root [10]. This is the case in Docker as well [10]. In Docker, one can configure the container to run as non-root but the runtime will by default run as root.

Running containers as root is clearly a security risk; if the container user manages to escape its isolation, it will be in control of the host machine. Therefore, it would be convenient to run containers as non-root if possible. That is indeed possible through the so-called rootless containers, a term which broadly refers to running the containers and also the runtime as non-root [11].

At a first glance, rootless containers seem like an unnecessary concept. Why do containers even need to run as root by default? There are a few reasons. One reason is that they introduce some level of complexity, especially with networking. Rootless containers cannot bind to privileged ports and have limited access to the filesystem.

Another reason is that the container image needs to install software using package managers. This is reasonable but only during the build phase of the image [10]. After the dependencies have been installed the container can be configured to run as non-root.

The last reason is that for some applications it makes sense to run as root outside of containers, for example apps that bind to privileged ports [10]. This has been then translated to containers as well where it does not make much sense as the application can bind to any port and then the port can be mapped to privileged ports. To conclude, the aforementioned reasons can be circumvented at the cost of increasing complexity and few restrictions with rootless containers. Besides complexity, performance can be an issue with rootless containers, it has been shown that creating a rootless container takes longer than a normal container [16].

3 Achieving rootless

How can one achieve rootless containers? It can be simply divided into two main categories, *creating* containers as non-root and *running* containers as non-root. As mentioned previously, the containers should be created and run as non-root on the host machine. In the rest of the paper, non-root user will specifically mean a non-root user on the host machine.

3.1 Creating rootless

Before going into creating rootless, Linux capabilities should be introduced, they are an essential feature when considering rootless containers at all as certain capabilities are required to create namespaces and control groups. Capabilities can be assigned to threads and they dictate what operations the thread can perform. Capabilities can also be assigned to files. Certain capabilities are also needed to create containers. For example, *CAP_SYS_ADMIN* capability allows to create mount namespaces [2]. Typically processes started by a non-root user do not have special capabilities [10], whereas processes started by a root user basically contains all of the capabilities [2]. To grant capabilities, one needs to have *CAP_SETFCAP* capability which is automatically granted to root. Essentially, capabilities offer a much more fine-grained access control compared to just being root or non-root.

One will notice a problem when trying to create containers as non-root because non-roots do not have the required capabilities. However, this can be circumvented with the use of user namespaces. The user namespace allows for a given process to have a different view of user and group IDs. In practice this means that a pseudo-root can be created inside a specific container. In this context pseudo-root means that the root user has admin rights inside a specific container but on the host machine, wherein it is actually a non-root user [10]. Within the user namespace the pseudo-root user can then apply capabilities to the processes as a typical root user would. However, these capabilities do not apply outside of the specific user namespace [15]. This means that even though the user has a wide range of privileges inside the namespace, it does not have them if it manages to escape.

However, using the pseudo-root introduces some complexity and limitations. One cannot expect an image that runs successfully as a root in a normal container to do the same in a rootless container, even though the rootless container might perceive itself to be running as root [10]. For example, the *CAP_NET_BIND_SERVICE* capability allows processes to bind to low-numbered ports but inside the rootless container the capability just does not work [10]. That would require the container to exercise the capability against host machines resources that it cannot see or have access to. Finally, it is good to realise that most of the applications running in normal containers will run successfully inside rootless containers as well [10].

3.2 Running rootless

How to run containers as non-root? A straightforward solution is to use rootless image and/or to specify a non-root user to run a container. Regarding rootless container images, Bitnami maintains a broad collection of non-root images [3]. Note that one can specify a pseudo-root to run the container as well. The important aspect is that the user is not root on the host machine.

As previously mentioned, running the containers as non-root is not rootless; if the container runtime is operated by root there is still a potential for outbreak. The running rootless section is quite irrelevant if the runtime operates as non-root. As the runtime is responsible for running the containers, it cannot elevate the containers to have root privileges if it is running as non-root itself. To summarise, if the runtime is non-root, one cannot run the container as root. Running the container as pseudo-root is still possible.

3.3 Rootless Docker

Docker required to run its daemon with root privileges. This has recently changed as Docker introduced rootless mode since version 19.03, and is considered stable since version 20.10 [13]. Docker still runs as rootful (as opposite to rootless) by default and just by running `docker` without `sudo` does not mean one is running rootless. Therefore, one needs to install the rootless mode and switch to it explicitly.

With rootless mode a new user namespace is created for the Docker daemon to operate in. As of Linux 3.8, user namespaces can be created by non-roots [15]. This effectively allows non-root users to work with Docker completely without the root users help.

Running containers as non-root in Docker can be achieved by using the USER directive to specify a non-root user to be used. The specified user is then used for RUN command which will start the container. One can also use the `-user` flag with `docker run` and then after the flag, specify a non-root user id or group id. If one does not specify the user, it will be the root. Rootless images can also be used through Docker Hub by searching, for example Bitnami images [4].

Rootless mode can be observed with Docker, for example by running `docker run -it alpine sh` and then in the terminal `whoami`. It will tell that the user is root. However, running `sleep 100` and then `ps -fC sleep` on the host machine will reveal the truth. From the output it can be observed that the process is not assigned to the actual root on the host machine. Whereas if the previous commands are run in rootful docker, it can be seen that the root inside the container is also the root on the host machine.

Rootless docker contains some limitations [13]. For example, AppArmor, checkpointing and overlay network cannot be used. Also, only certain types of storage drivers are supported. Cgroup are only supported with cgroup version 2 and systemd . However, one can use `ulimit` and `cpulimit` to limit resources in a more traditional way.

3.4 Networking

Much of the rootless containers complexity comes from networking requirements [7]. As mentioned previously, binding to a low-numbered port cannot be done with non-root users as it requires capabilities that apply at the host machine level. Incoming internet connections cannot directly react to network namespaces. vEth pairs cannot be created without privileges. Also, lower network performance occurs as the networking system needs additional components to function as in privileged environments. Aforementioned problems are part of the reason networking tools as RootlessKit and slirp4netns exist. Those help to setup rootless containers while providing the same functionality as in privileged environments. RootlessKit is an open-source project created to run Docker as pseudo-root, leveraging user namespaces [9]. slirp4netns provides networking for unprivileged network namespaces by creating a tap device that acts as a default route [9]. Slirp4nets is a component used in rootless containers to provide communication from inside a container to the outside [7]. It achieves this by using a tap device.

Network interfaces cannot be implemented with user namespaces alone because vEth pairs need to be created between a container and a host [7]. RootlessKit helps with this as it relays communication from the outside to an intermediate network namespace. Intermediate network namespace has a bridge that containers can connect to and create its vEth pair. RootlessKit is also used for example, to make /etc writable.

4 Sandboxing

Running containers as root is not a problem itself, unless the root user is able to escape the container and access the host machine. One approach to tackle such problem is to use sandboxing. Sandboxing means to isolate an application so that it has a limited access to resources [10]. The definition overlaps with containers quite a bit as they also limit the access and visibility of resources. However, limiting resources with sandboxing means to limit what the actual code can do on the host machine.

There exists many sandboxing mechanisms [17]. But in this paper, sandboxing is briefly covered through only one application, called gVisor. gVisor is an application kernel developed by Google [10]. It can be viewed as a layer between a running application and the host operating system.

Its task is to intercept system calls of the application and then to replace or modify them to be executed on the host machine.

gVisor does not allow the application itself to control the system calls [14]. This provides security as the system call API can occasionally be exploited due to bugs and race conditions. gVisor does not just directly relay the system call in the host machine but rather it might make modifications or limit the call. It is important to note that using system calls through gVisor will still result in a system call on the host machine.

gVisor does still come with limitations. Mainly reduced performance and incomplete features [10, 20]. It does not implement all of the Linux system calls which obviously is a drawback as some of the system calls are simply not available. If the sandboxed application makes frequent system calls, it might significantly impact the performance, gVisor has been measured to have approximately 50% overhead in system calls compared to bare-metal [17]. Finally, the gVisor is large and complex which increases the likelihood of having vulnerabilites in itself [10].

5 Container outbreak

As mentioned previously, the main motivation for rootless containers is to protect the host machine in the event of container outbreaks. Container outbreak is an event where the container user is able to access or see resources on the host machine. This might happen in multiple ways, often due to poor configuration [10].

This section introduces a real life example of breaking out of a container and taking malicious action. Before going into the example, `confidential.sh`-script has been added as root under the `/usr/bin`-folder. Running the script is only allowed by root and running it echoes "This is confidential. This file cannot be modified.". Groups or others do not have any permissions enabled for the file. For the rest of the section, a non-root user is used.

An example of poor configuration is to mount a host directory to be available inside a container [10]. This can achieved with `-v` flag. One can run `docker run -it -v /bin:/rootbin ubuntu bash` in host machine's root folder to map `/bin` on the host machine to rootbin-folder in the container. Running `ls` inside the container shows indeed a `rootbin`-folder being present. Inside the rootbin a `confidential.sh`-file is found and it can be read with `cat confidential.sh`. The container was started with non-root user and it is still able to read the file. This happens because the container user is actually mapped to the root user on the host machine. This would also mean that the user is able to edit the file, which is indeed possible and it can be edited with, for example, `cat » confidential.sh echo "This file has been modified."`. Now executing the file outputs `This file has been modified` on the last line. To be clear, the aforementioned actions have been made inside the container.

On the host machine's terminal `confidential.sh` can be executed again and it can be observed that the file output has changed, on the last line it reads `This file has been modified`. This is only one example of what could be done with poor configuration. The container user could also modify other files in the bin file, for example, hide some scripts inside some common executables like `ls`-command. The possibilities are limitless and very severe damage can be done.

Now observe what happens when rootless context is enabled. Container is run with the same command as before and `rootbin`-folder is created and inside it is `confidential.sh`-file. However, executing the file outputs: `permission denied`, the same applies when trying to read or write to the file. Running `whoami` outputs `root` but this is not actually root on the host machine and therefore the container user has no permissions on the file. Still, mapping the bin-folder to the container reveals the existence of `confidential.sh`-file, it would be best not to map the folder altogether if not needed.

Note that gVisor does not help preventing the above exploit as it does not concern about namespaces or cgroups. gVisor is only allowed to block malicious system calls. It is advised to run rootless containers paired with gVisor for maximum protection.

Mounting critical files to a container's environment poses a very serious threat but an even bigger threat can be introduced with Docker's `privileged` flag. In Docker's default environment root user is used to run containers but it does not contain all capabilities [10]. The `privileged`-flag effectively gives a much more wider range of capabilities for the container user, the capabilities include CAP_SYS_ADMIN which can be used to modify namespaces and mount filesystems. This means that a malicious user is able to mount the `/bin`-folder by itself.

6 Conclusion

Containers often run as root by default, for example, this is the case considering Docker's default environment. Running containers as root is not an issue in itself but it does leave a possibility of malicious user being able to escape the container environment with root access on the host machine.

Rootless containers is a technology that mitigates the threat of container outbreaks by running containers as pseudo-root. In the event of an outbreak the container user is not able to leverage root privileges on the host machine. Rootless allows for unprivileged users to create, run and manage containers [11]. They do have added complexity compared to rootful containers but this is rarely visible to the end-users as transferring rootful containers to rootless usually only requires making sure that the runtime is run by non-root on the host machine.

Rootless containers are relatively new technology and their large-scale adoption is still underway. It is encouraged for everyone to use rootless containers instead of rootful as there really is no reason for one to predispose their infrastructure to container outbreaks.

References

[1] Donnie Berkholz. *Docker Index Shows Continued Massive Developer Adoption and Activity to Build and Share Apps with Docker - Docker*. URL: https://www.docker.com/blog/docker-index-shows-continued-massive-developer-adoption-and-activity-to-build-and-share-apps-with-docker/ (visited on 10/19/2022).

[2] *capabilities(7) - Linux manual page*. https://man7.org/linux/man-pages/man7/capabilities.7.html#DESCRIPTION. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html#DESCRIPTION (visited on 10/20/2022).

[3] *Containers*. URL: https://bitnami.com/stacks/containers (visited on 10/20/2022).

[4] *Docker Hub Container Image Library | App Containerization*. URL: https://hub.docker.com/ (visited on 10/23/2022).

[5] *Docker overview*. Docker Documentation. Oct. 19, 2022. URL: https://docs.docker.com/get-started/overview/ (visited on 10/19/2022).

[6] Jorge Gomes et al. "Enabling rootless Linux Containers in multi-user environments: The udocker tool". In: *Computer Physics Communications* 232 (2018), pp. 84–97. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2018.05.021. URL: https://www.sciencedirect.com/science/article/pii/S0010465518302042.

[7] Naoki Matsumoto and Akihiro Suda. "Accelerating TCP/IP Communications in Rootless Containers by Socket Switching". In: (July 2022).

[8] nishanil. *What is Docker?* URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-define (visited on 10/19/2022).

[9] Alex Rapatti. "Rootless Docker Containers in Continuous Integration". In: (Nov. 2021). URL: http://www.theseus.fi/handle/10024/498794.

[10] Liz Rice. *Container security : fundamental technology concepts that protect containerized applications*. 1st. Sebastopol, CA: O'Reilly Media, 2020. ISBN: 1-4920-5669-3.

[11] *Rootless Containers | Rootless Containers*. URL: https://rootlesscontaine.rs/ (visited on 10/02/2022).

[12] Bowen Ruan et al. "A Performance Study of Containers in Cloud Environment". In: *Advances in Services Computing*. Ed. by Guojun Wang, Yanbo Han, and Gregorio Martínez Pérez. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 343–356. ISBN: 978-3-319-49178-3. DOI: 10.1007/978-3-319-49178-3_27.

[13] *Run the Docker daemon as a non-root user (Rootless mode)*. Docker Documentation. Oct. 21, 2022. URL: https://docs.docker.com/engine/security/rootless/ (visited on 10/23/2022).

[14] *Security Model*. URL: https://gvisor.dev/docs/architecture_guide/security/ (visited on 10/22/2022).

[15] *user_namespaces(7) - Linux manual page*. URL: https://man7.org/linux/man-pages/man7/user_namespaces.7.html (visited on 10/23/2022).

[16] Guillaume Everarts de Velp, Etienne Rivière, and Ramin Sadre. "Understanding the performance of container execution environments". In: *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*. WOC'20. New York, NY, USA: Association for Computing Machinery, Jan. 11, 2021, pp. 37–42. ISBN: 978-1-4503-8209-0. DOI: 10.1145/3429885.3429967. URL: http://doi.org/10.1145/3429885.3429967.

[17] Xingyu Wang, Junzhao Du, and Hui Liu. "Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes". In: *Cluster Computing* 25.2 (Apr. 1, 2022), pp. 1497–1513. ISSN: 1573-7543. DOI: 10.1007/s10586-021-03517-8. URL: https://doi.org/10.1007/s10586-021-03517-8 (visited on 11/11/2022).

[18] *What are containers?* Google Cloud. URL: https://cloud.google.com/learn/what-are-containers (visited on 10/02/2022).

[19] *What is a Container? - Docker*. Nov. 2021. URL: https://www.docker.com/resources/what-container/ (visited on 10/02/2022).

[20] Ethan G. Young et al. "The True Cost of Containing: A gVisor Case Study". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotcloud19/presentation/young.