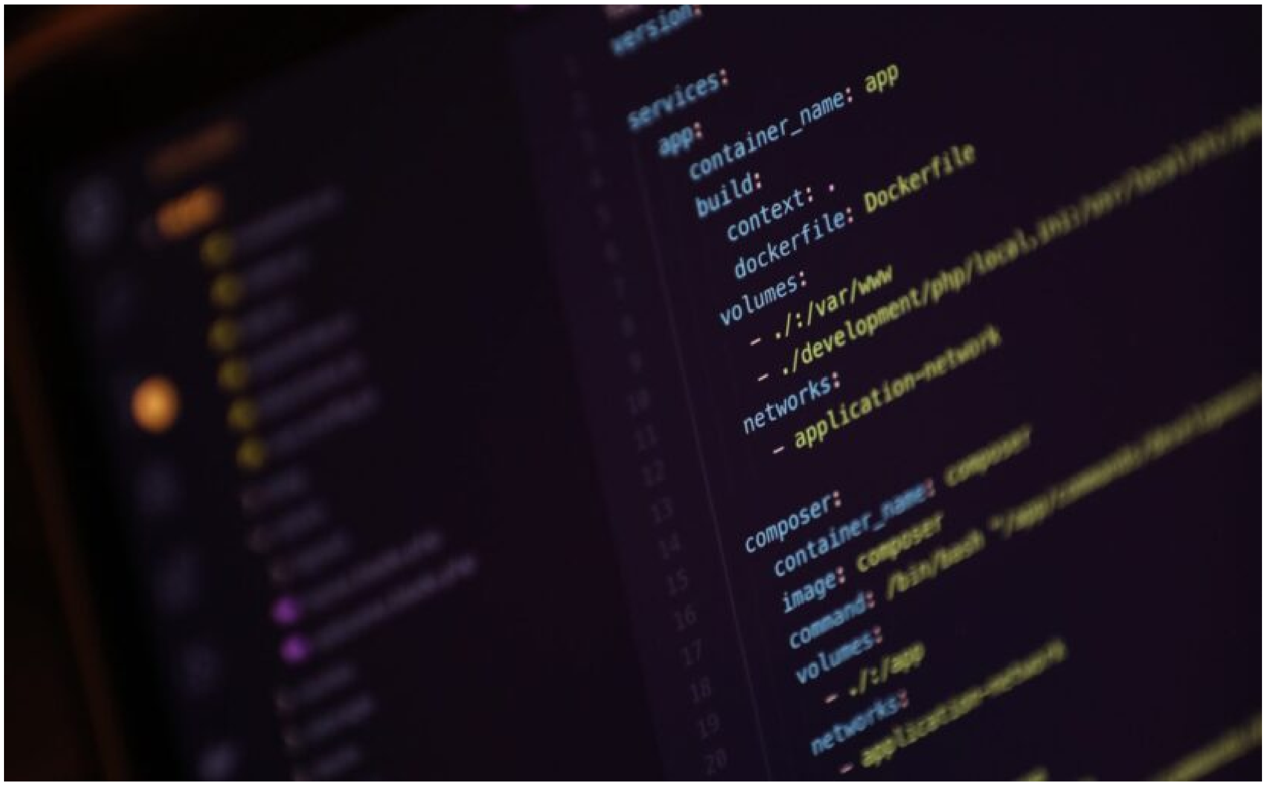
**Bentham’s Gaze**

Information Security Research & Education, University College London (UCL)



# Vulnerability in Linux containers – investigation and mitigation

Operating system access controls, that constrain which programs can open which files, have existed for almost as long as computers themselves. Access controls are still widely used and are more flexible and efficient when compared to cryptographically protecting files. Despite the long history, there continues to be innovation in access control, particularly now in containers, like Docker and Kubernetes and similar technologies offered by cloud providers. Here, rather than running lots of software on a single computer, the service is split up into microservices running in containers. Each container is isolated from others on the same computer as if it has its own computer and operating system and is prevented from reading files in other containers.

However, in reality, there's only one operating system, and the container runtime's role is to create the illusion that there are more than one. As part of its job, the runtime should also set up containers such that access control works inside each container because not every program running inside a container should be able to access every file. Multiple containers can also be given access to the same directory, and access controls used to restrict what each container can do with the directory contents. If access controls don't work properly, an attacker could read or modify files they should not be able to.

Unfortunately, there is such a vulnerability. The bad news is that it originates from an omission in the specification that underlies all the major container runtimes and so is present regardless of which container runtime you use (e.g. runc, crun, Kata Containers) and regardless of whether you use containers directly (e.g. through Docker or podman) or indirectly (e.g. through Kubernetes). The good news is that the vulnerability affects a feature of Linux access control permissions that is not widely used – negative group permissions. However, if your system does depend on this feature then the vulnerability could be serious. Read on for more details about the vulnerability, why it exists and what can be done to mitigate the problem.

## Introduction to Linux permissions

In Linux there are user accounts and each user is also a member of a group. Each object (files, directories, devices, etc.) has an associated owner and associated group. The object also has a set of permissions associated with the three classes: owner, group, and other. These permissions tell the operating system whether a user should be able to read from the object (r), write to the object (w) and execute the object (x). If a user is the owner of an object the owner-class permissions are used, if the user is a member of the file's group, the group-class permissions are used, and otherwise the other-class permissions are used.

For example, a file containing a company's finance database could be owned by the Chief Financial Officer (CFO) and have owner class permissions "r+w". It could have the group set to "auditors" with group-class permissions only "r" and other-class permissions set to nothing. Then the CFO could freely read and write to the database, all members of the group auditors could read it, and everyone else cannot access the database at all.

This sort of configuration works but isn't flexible. What happens if an auditor is also a member of the company sports team and that team use group-class permissions for controlling access to the database of upcoming fixtures? Would the auditor have to leave the auditors group and join the sports team group and so lose access to the accounts database? In fact, no, because not only does a user have a primary group, but they can have supplementary groups. When an access control decision is made, the operating system uses the group-class permissions if the group of an object matches any of the user's groups. So a user could be a member of the auditor group and the sports team group. Not only can a user take on the rights of any of its supplementary groups, but it can also change the group of any file they own to any of their supplementary groups.

The final concept I need to introduce is set-group programs. Let's suppose that we don't want every user to be able to see the finance database, but we do want every user to be able to get a balance report for their department. We can create a program that checks which user started the program and get the balance report for that user. But that won't work because programs inherit the rights of the user that runs them, and the user can't read the database. So we make the program set-group "auditor", and now when it runs, it will take on the rights of the auditor group and be able to read the finance database. (There's similar functionality for set-user programs, but I won't go into this here).

## Negative group permissions

In the above example, the user-class permissions (r+w) were greater than the group-class (r), which were, in turn, greater than the other-class (nothing). This is the way things usually are, with the more specific class having at least as many rights as less specific classes. However, it doesn't need to be that way. If an object is set as, for example, user: r, group: nothing and other: r, then everyone can read this file except members of the object's group. This is an intentional feature of Linux permissions and does get used, albeit not very frequently. Where it might be useful is for building up a deny-list for a sensitive object so that certain untrustworthy programs cannot access it.

To support such scenarios, users should not be able to drop a group because they could then bypass access control enforced through negative group permissions. But could set-group executables be used to do so? A user could create a program, change its group to one of the user's supplementary groups, and then run this program. When the program runs, its group will be the supplementary group and not the user's primary group. Would that allow the user to drop the primary group and get access to an object its group would not allow?

Fortunately, it does not, because when the user logs in, their primary group is duplicated and added to the list of supplementary groups. When the set-group program runs, its group is indeed the supplementary group chosen, but the list of supplementary groups still includes the original primary group. So the operating system will still reject the access request because the group that is denied access is in the supplementary group list. However, if the code is running in a container, the situation is different.

## Primary group is not duplicated in containers

The behaviour of duplicating a user's primary group at login was introduced in the 4.4 BSD operating system released in 1994, and Linux takes a similar approach. However, Docker containers did not follow and while the primary group is set, this group is not duplicated in the list of supplementary groups. The Open Container Initiative created an interoperable standard for containers which does not explicitly say what containers should do here. Still, all the implementations I am aware of followed the example of Docker. Consequently, a program running in a container could drop its primary group. If negative group permissions are in use, the program would be able to violate the system's access control policy.

## Exploiting the vulnerability

If an attacker can run code in the container, they can create a set-group program with the group set to one of the supplementary groups. When this program runs, its group will be the supplementary group, and the list of supplementary groups will be that user's supplementary groups. However, the user's primary group will have been dropped. If the program accesses a file that has negative group permissions for the user's primary group, it will be able to access the file in a way that the user was not supposed to, and so violate the intended permissions.

A demonstration of this vulnerability can be found in the video below, which uses the proof-of-concept code I've made available on Github. The container images I created are on Docker Hub, for both x86_64 and aarch64.

## Mitigating the problem

I reported this vulnerability to Docker on 27 May 2022, but investigation of this vulnerability was passed onto the Moby project, since it affects multiple implementations of Linux containers and not just Docker. Developers of affected software have been working on patches that would fix the vulnerability and address the omission in the specification. Since the vulnerability only affects an access control feature that is not commonly used, the severity of the vulnerability has been assessed to be low and so the vulnerability can be discussed publicly while the mitigation work is underway.

Common Vulnerabilities and Exposures (CVE) IDs have been allocated to track this vulnerability as it relates to affected software packages. Why are there multiple CVE IDs? The Open Container Initiative Runtime Specification does not require the vulnerable behaviour but just is sufficiently ambiguous as to permit it. Even though, currently, all implementations of this specification are vulnerable, the CVE IDs are still associated with the vulnerable software rather than the specification. I hope the specification will also be revised to explicitly require that implementations enforce the correct behaviour. The CVE IDs allocated so far are:

- CVE-2022-2989: podman
- CVE-2022-2990: buildah
- CVE-2022-2995: cri-o

- CVE-2022-36109: Moby (Docker Engine)

In the meantime, there are however some steps that can be taken to work around the vulnerability.

## Initialising containers with "su"

The vulnerability exists because the container runtime does not set up groups properly when these are specified in the container configuration (i.e. the `USER` Dockerfile instruction and the `runAsUser`/`runAsGroup` Kubernetes setting). An alternative is to start the container as root, and then use the "`su`" command with the "`-l`" flag to set the user. This approach will set up groups properly and so avoid the problem.

## Duplicating groups manually

The container runtime uses the contents of `/etc/passwd` and `/etc/group` to set up the primary group and supplementary groups, respectively. If you modify `/etc/group` to add the user to their primary group a second time, the container runtime will set up the groups properly. This approach is a bit of a hack but it seems to work.

For example, here the user's primary group is specified both in `/etc/passwd` (with the numeric group ID 1000) and in `/etc/group` where the user is also listed as a member.

**`/etc/passwd`:**
```
user:x:1000:1000:Linux User,,,:/home/user:/bin/ash
```

**`/etc/group`:**
```
user:x:1000:user
```

## Blocking setuid/setgid programs

Exploiting the vulnerability depends on having a set-group program to change the available privileges. If you don't need setgid or setuid programs you can disable this functionality through the Docker "`--security-opt no-new-privileges`" flag or the Kubernetes "`allowPrivilegeEscalation=false`" setting.

### Identifying the use of negative group permissions

It is difficult to know whether software relies on negative group permissions because this is an implementation detail which is not often explicitly documented. To serve as a starting point for investigation, I've written a program that searches for files and directories that have negative group permissions. It is not perfect – it won't identify cases where negative group permissions are set for temporary files. It also won't find cases where negative group permissions are implemented in configuration files (e.g. for `sudo`) rather than on the file system. There can also be false positives. On my systems, I did find many files that have negative group permissions, apparently by accident.

## Lessons for system design

This vulnerability should, of course, be fixed, but there are also some broader lessons that can be drawn. Including a user's primary group within the supplementary groups isn't written down as a requirement – it is undefined. What "undefined" means is that applications shouldn't rely on a particular behaviour, but that doesn't mean all choices are equally correct. Interoperability standards have this problem: they specify the minimum to allow maximum flexibility, but that isn't what you want for security. The EMV card payment interoperability standard suffers from the same weakness. There is a separate role for a standard that requires certain behaviour for security or other reasons.

There are also lessons for software rewrites. There's a movement to rewrite C software in memory-safe languages, with Rust now a popular choice. Doing so can indeed bring substantial security benefits, but any rewrite also risks missing essential features, particularly when they are documented nowhere but in legacy code. Docker effectively rewrote the group initialisation code in Go and avoided many of the potential flaws in the original C code but created a new vulnerability by not implementing the login sequence in the same way as Linux. That is not to say that rewriting is always a bad idea, but just that doing so introduces new risks that should be mitigated.

**Update 2022-08-21:** Add Common Vulnerabilities and Exposures (CVE) references to vulnerabilities in affected products: CVE-2022-2989 (podman), CVE-2022-2990 (buildah), and CVE-2022-2995 (cri-o).

**Update 2022-09-09:** Add CVE reference to vulnerability in Moby (Docker Engine): CVE-2022-36109.

*Photo by Mohammad Rahmani on Unsplash.*

2022-08-22 Steven J. Murdoch Operating systems