Modern Linux security

G. Lettieri

14 December 2023

1 Introduction

Linux has come a long way from its Unix roots and today it implements many features that are either not found in other operating systems, or are implemented in incompatible ways.

Many of these features are used together to implement the *container* abstraction. Containers are a way to isolate a set of processes and make them think that they are the only ones running on the machine. The machine they see may feature only a subset of the resources actually available on the entire machine (e.g., less memory, less disk space, less CPUs, less network bandwidth). Many different containers may coexist on the same machine. To achieve this, containers implementations (like Docker) leverage Linux's *namespaces* and *control groups*.

Containers are *not* virtual machines, even if they may look like ones in some cases. Processes running inside a container are normal processes running on the host kernel. There is no guest kernel running inside the container, and this is the most important limitation of containers with respect to virtual machines: you cannot run an arbitrary operating system in a container, since the kernel is shared with the host (Linux, in our case). The most important advantage of containers with respect to virtual machines is performance: there is no performance penalty in running an application inside a container compared to running it on the host.

Containers are complex environments that use complex and relatively new Linux features. Many people don't trust their security and additional Linux features are added into the mix, like *security modules*. These are kernel modules that hook into several security-sensitive operations in the kernel and perform additional checks. SELinux and Apparmor are examples of such modules.

2 Capabilities

One of the problems of the Unix security model is the existence of the root user who can bypass all file system access controls. The problem has only become worse when more and more features have been added to Unix derivatives, since the privilege of using these new features has invariably been granted only to the root user. Thus, binding a socket to a TCP or UDP port below 1024 needs root privilege, and this means that all DNS, mail, web, ftp, telnet, servers (to name just a few) where usually run as root. This gave these servers also the ability to, e.g., completely wipe out the entire file system, if exploitable bugs where found in them (and a long list of such bugs where indeed found in all of them).

After the first successful exploits, these servers were changed so that they would hold root privilege only while executing the operations that really needed it, and dropping the privilege for the rest of the execution. Root privilege can be dropped permanently by calling `setuid(uid)` with an `uid` greater then zero (like the `login` program does), but other system calls can be used to drop it only temporarily, so that it can be regained later.

Capabilities try to further improve the handling of root privilege by splitting it into many smaller privileges—the capabilities—that can be granted, revoked (temporarily or permanently) and inherited individually. So, for example, the `CAP_DAC_OVERRIDE` capability gives a process full access to any file and directory, while the `CAP_NET_BIND_SERVICE` capability gives a process the ability to bind a low numbered Internet port. Internet servers can be given the latter capability, without the need to also grant them the former one.

Capabilities where being defined in the POSIX 1e standard, which however was never ratified. Nonetheless, many systems, including Linux, have implemented the extant part of the standard extending it in many ways. The Linux kernel only checks privilege by looking at the capabilities of the processes, and emulates the legacy root behavior through a set of compatibility rules.

In the POSIX model, also implemented by Linux, each process has a set of *permitted* and a set of *effective* capabilities. When the kernel needs to check if a process has the right to perform a given action (such as binding a socket to a low numbered port) it only looks for the corresponding capability in the effective set. Processes can manipulate their capability sets using the new `setcap()` system call: they can copy a capability from their permitted set to their effective one, or they may drop any capability from either the permitted or the effective set. Dropping a capability from a permitted set is like calling `setuid(uid)` to permanently drop root privilege, while dropping it from the effective set is like dropping root privilege only temporarily, while retaining the ability of re-acquiring it later. These sets are inherited across a `fork()`.

To fully reproduce the functionality of the old root privilege, however, we must define rules for what happens when a process `execve()`s a new program. In the legacy Unix model, a set-uid-root program can grant new privilege. Note that we mention set-uid-*root* programs explicitly: a set-uid program with a non-root user will only change the uid of the process, but a set-uid-root program will also give the process all the root powers. In a capability environment, this should translate into programs that can grant some new capabilities to the processes that execute them. Moreover, if the process already runs as root, it inherits this privilege across the `execve()`. A capability environment may want to more precisely control this privilege inheritance. For the purpose of granting new capabilities and inheriting old ones across an `execve()`, each process also as a set of *inheritable* set of capabilities, and each *executable file* also carries a *permitted* set and an *inheritable* set of capabilities. The capability sets of files can be inspected and manipulated using the new `getcap` and `setcap` utilities. Assume first that all inheritable sets are empty. Then, the file's permitted set becomes the new permitted set of any processes that executes the file. Note that this *replaces* the old process' permitted set, dropping all capabilities that the process may had. To keep a capability across an `execve()`, a process must copy it into its own inheritable set, but this is not sufficient: the same capability must also be present in the inheritable set of the executable file. If the capability is present in both sets, `execve()` will add it to the new permitted set of the process, alongside the capabilities found in the file's permitted set.

The effective set of the process after the `execve()` should normally start empty: the idea of the model is that programs should always explicitly enable their capabilities (by copying them from the permitted set) only when they actually need them. Linux, however, adds a new feature for compatibility with legacy set-uid-root programs that where written without any knowledge of capabilities (i.e., most of them): an "effective flag" can be turned on the executable files. If this set is on, the new permitted capabilities (the ones in the executable permitted set, plus those found in both the process' and in the file's inheritable sets) are automatically copied into the process' effective set during the `execve()`. For an example, you can look at the `ping` program: this program needs to open a raw socket to inject an ICMP packet into the network, and for this it needs the `CAP_NET_RAW` capability. The `ping` executable file has this capability in its permitted set, so the capability will go into the permitted set of the processes that execute it. However, the legacy `ping` program doesn't know that it has to copy this capability into the effective set. Instead of modifying the program, Linux distributions just set the effective flag on the executable, so that the kernel can copy the capability by itself[1].

The inheritable sets are probably the most complex feature in the model. The standard's designers probably intended that most programs should always come with a full inheritable set, so that capabilities could be inherited, e.g., across scripts. This is how Linux originally implemented the idea. The discovery of a nasty privilege-escalation bug, though, made Linux switch to the opposite default: inheritable sets (both in processes and in files) are normally empty[2] This makes capabilities much more harder to use, so that Linux developers later added yet another set of capabilities ("ambient" capabilities) which are easier to inherit.

We can use the `capsh` utility to experiment with capabilities. In particular, "`capsh --print`" will print all the capability-related information of the current shell. The `setcap` and `getcap` utilities, instead, can be used to, respectively, write and read file capabilities. For example, "`getcap /bin/ping`" should show that the `ping` program has the `cap_net_raw` capability in its permitted set, and that the effective flag is set.

[1]Modern versions of `ping` compiled for Linux now understand capabilities and are able to use the effective and permitted sets appriately.

[2]You can read about the bug here: `https://sites.google.com/site/fullycapable/thesendmailcapabilitiesissue`.

3 Namespaces

Namespaces provide a means to segregate system resources so that they can be hidden from selected processes. Let's start with an old example of the idea.

3.1 The `chroot` feature

Unix V7 included the `chroot()` system call, which makes it possible for each process to have its own idea of where the root directory is. This can be used to make a set of processes think that a subset of the filesystem actually is the entire file system. The feature was implemented as follows:

- For each process, the kernel remembers the inode of the the process *root directory*;
- the new `chroot(path)` system call can be used to change the calling process root directory; only root can call this primitive;
- the process current root directory is used as a starting point whenever the process passes the kernel (e.g., in `open()`, but also in `chroot()` itself); a filesystem path that starts with "`/`";
- whenever the kernel walks through the components of any filesystem path used by the process and reaches the process root directory, a subsequent "`..`" path element is ignored;
- the process root directory is inherited across `fork()` and `execve()`.

The original purpose of this feature was to test new distributions of Unix: the developers would created a filesystem subtree containing the new binaries with all their supporting files and then **`cd`** and `chroot` to the root of the subtree: the new shell would then start using only the new binaries. Later, the `chroot` feature was used to segregate untrusted processes that provide network services and, because of possible bugs in their implementations, may be forced by remote attackers to execute arbitrary code. The idea is to prepare a subtree in the filesystem that contains only the things that are needed for the execution of the server, and nothing else—a so-called chroot environment. Then, the server process is started after a `chroot()` to the root directory of the chroot environment. Even if the server is subverted, it cannot access any file outside of the chroot environment.

Exercises

3.1. If the implementation of the mechanism is exactly as described above, then the root user can actually escape from a chroot environment. How?

3.2 Linux namespaces

The chroot feature creates a new *namespace* for file system paths: all paths, like "`/etc/passwd`" or "`/..`", acquire a meaning that depends on the root directory of the process using the path. This, however, is not sufficient to completely segregate a process. Contrary to popular belief, in fact, not everything is a file in Unix. For example network interfaces, network ports, users and processes are not files. While we can have as many instances as we want of `/etc/passwd`, each different and living in its own chroot environment, we can only have one port 80 throughout the system (thus, only one web server), only one process with pid 1 (thus, only one init process), and user and process ids will have the same meaning in all chroot environments. Thus, for example, a process running in a chroot environment will still be able to see all the processes running in the system, and it will be able to send signals and do a `ptrace` attach to all the processes belonging to any user with the same user id as its own.

Namespaces have been introduced to create something similar to a chroot environment for all these other identifiers. In Linux, each process has its own network namespace, pid namespace, mount namespace, user namespace and a few others. Network interfaces and ports are only defined inside a namespace, and the same port number may be reused in two different namespaces without any ambiguity. The same holds true for processes and users. Normally, all processes share the same namespaces, but a process can start a new namespace that will be then inherited by all its children, grandchildren, and so on. This is done when the process is created using the `clone()` system call. This system call (taken from the Plan 9 OS) is the new, preferred way to create new processes in Linux, since it generalizes the behavior of `fork()` and can be also used to implement `pthread_create()`. The idea is that both processes and threads share something with their creating process, and have a private copy of something else. For example, processes share open files with their parent, but have a private copy of all the process memory. Threads, instead, also share the process memory. The `clone()` system call receives a set of flags that allow the programmer to choose what to share and what to copy. This same system call has been extended to implement namespaces, essentially by adding flags for the sharing or copying of the network, pid, user namespaces and so on. A couple of other system calls have also been added: `unshare()`, which accepts the same flags as `clone()`, can be used to create new namespaces for the current process, and `setns()`, which can be used to put the current process in an existing namespace.

The `unshare` utility can be used to experiment with namespaces. It accepts a few switches to select which new namespace(s) must be created, and the path of a program (`$SHELL` by default). It then calls `unshare()` with the appropriate flags, followed by an `execve()` of the program. The program will then be executed in the new namespace. For example, if we type the following as root[3]

[3] Actually, with the `CAP_SYS_ADMIN` capability in the effective set.

```
unshare -n /bin/sh
```

we obtain a `/bin/sh` that runs in a new network namespace. An `ifconfig -a` (or `ip link show`) command will show only the loopback interface: all the other interfaces are in the original network namespace and are no longer accessible. The loopback interface itself is not the original one: it's a new instance that can have an address of its own.

The network namespace is a bit special because the new one starts empty. Most other namespaces, instead, start with a copy of the original one. For example, we can enter a new mount namespace with

```
unshare -m /bin/sh
```

If we issue a `mount` command with no arguments in the new shell, we will see the same output as outside the namespace. Differences will start to show up when we mount or unmount file systems in either namespace: the changes will not (by default) propagate to the other one.

We can find the namespaces of each process by looking at its directory in the `proc` pseudo-filesystem. In particular, the `ns` subdirectory contains one "magic" symlink for each kind of namespace. Each magic symlink contains the name of the corresponding namespace kind, followed by a double colon, followed by a numeric identifier in square brackets. Processes that share the same numeric identifier for the same kind of namespace live in the same namespace of that kind.

3.2.1 Pid namespaces

Pid namespaces create a new namespace for process identifiers. This kind of namespace behaves specially for several reasons:

1. a process created in a new namespace actually lives in *both* the new namespace and the one(s) of its parent, with a different pid in each one of them;
2. the first process created in a new pid namespace (pid 1) becomes the new "child reaper" of the namespace (see below);
3. the `unshare()` and `setns()` system calls don't change the pid namespace of the calling process, but only the one of the child processes created from that point on.

The "child reaper" is the process that becomes the new parent of the orphaned processes, i.e., processes whose parent has terminated. No process can live in a namespace if there is no child reaper for it: if the child reaper terminates, all processes in the namespace are killed and no new one can be created[4].

[4]The reaper of the first process is the reaper in the pid namespace of its *parent* process. This is also true for any other process whose parent lives outside of the pid namespace.

```
#include <stdio.h>
#include <unistd.h>

int main()
{
  printf("%d\n", getpid());
  getchar();
}
```

Figure 1: A simple program that prints its own pid and waits for a newline.

We can observe the peculiarities of the pid namespaces by experimenting with `unshare`. First, let's create a program that prints its own pid (see Figure 1). Let's call it `mypid`. Now, let's start a shell with a new pid namespace: (remember to run the command as root):

```
unshare -p /bin/sh
```

If we type "**`echo`** `$$`" to ask the shell to print its own pid, we see that it is not 1. Actually, we see the same pid that we see outside of the namespace. This is because the `unshare` command calls `unshare()`, which creates a new pid namespace for the *children* of the process, but then calls `execve()` to transform *itself* into `/bin/sh`: the shell, therefore, is still outside of the new namespace, as per point 3 above. If we look at the shell's symlinks in its `/proc/<pid>/ns` directory, we see a new entry called `pid_for_children`. This symlink will point to the new namespace when the first child will be created (note that the **`echo`** above uses the shell's builtin command without creating a new process). If we type `./mypid` the program will print "1". At the same time, the `pid_for_children` symlink will now actually point to the newly created pid namespace.

If we let `mypid` exit (by typing enter) we render the pid namespace unusable as per point 2 above: any other (non builtin) command that we try to execute will fail with an error on `fork()`. To remedy this, exit from the shell and issue the `unshare` command again. This time, let's create a first process that does not exit immediately, e.g.:

```
tail -f /dev/null &
```

Now we can run as many programs as we want in the new namespace. Let's run `mypid` again. This time it will print "2". While `mypid` is still running, let's search for it in the parent namespace (just use another terminal). We can see that the process is visible in the parent namespace too, but with a different pid. This confirms point 1 above. If you exit from the shell, remember to "**`kill`** `-9`" the `tail` process (it will ignore any other signal except `SIGSTOP` and you cannot kill it from within the namespace, not even with `-9`).

The above trick of the tail is not necessary, since `unshare` can be asked to execute the command in a new child with `-f`:

```
unshare -f -p /bin/sh
```

In this way, the shell itself will have pid 1, keeping the namespace active for the entire session.

Now let's try to run "**ps** `-ef`" in the new shell. We perhaps expect to see only the processes in the new namespace, but actually we can see *all* the processes, with their *original* pids. This is because **ps** takes its informations from `/proc`, and we are still using the original `proc` filesystem. We need to mount a `proc` from within the new namespace and, if we want **ps** to get information from it, we have to mount it on the `/proc` directory. This, however, would confuse the rest of the system, so it is better to do it in a new mount namespace. Exit from the shell again and run

```
unshare -f -p -m /bin/sh
mount -t proc proc /proc
```

Now **ps** inside the new shell will show only the processes in the new pid namespace. Moreover, thanks to the new mount namespace, the `mount` command has not affected the rest of the system, so **ps** run outside continues to work as usual[5].

3.2.2 User namespaces

Probably the most interesting kind of namespace is the "user" namespace. These namespaces allow processes to have a personalized view of user and group identifiers. Let us focus only on uids, and let us call "internal" ids the uids used inside a user namespace, and "external" ids the uids used outside of it. Each user namespace has a mapping between internal uids and external uids. When a process inside a namespace uses a uid (e.g., in a `setuid()` system call, or when creating a new file or directory) the internal uids are converted to the external ones. Conversely, when a process inside the namespace reads uids from, e.g., a file system inode, the kernel converts them from external to internal. User ids that have no mapping cannot be used (unmapped external ones are converted to "nobody" inside the namespace). In particular, the internal uid 0 can be mapped to an external, non-privileged uid. When coupled with the other namespaces, this creates the possibility to have containers that look like normal, independent systems, and the administrators of the subsystem have no special privilege on the "external" systems, nor on any other independent subsystem. User namespaces are special in many ways. In particular, they act as "owners" of the other namespaces that are created from within them, and the root user inside the the user namespace has full capabilities on the resources managed by the owned namespaces. Moreover, unlike other namespaces, which can only be

[5]The `unshare` command can mount `proc` by itself when passed the `--mount-proc` option.

created by root (actually, by processes with the `CAP_SYS_ADMIN` capability in their effective set), user namespaces can be created by any user.

We can verify the latter assertion by running the following command *as a normal user*:

```
unshare -r
```

We apparently obtain a root prompt, from a non-setuid program, without any effort. The `id` command will show that our uid is now 0. If we do an "ls -l" on our home directory, we will see that all of our files now apparently belong to root. On the contrary, all the files that previously belonged to root (or to any other user) now show up as belonging to nobody.

What has actually happened is that the `unshare -r` command has created a new user namespace, where uid 0 inside the namespace is mapped to our effective uid outside of the namespace, and other uids are all unmapped. We can verify that this is the case if we lookup our new shell from another terminal: the process is still running with our uid, and no file has actually changed ownership.

If we run "`capsh --print`" inside the namespace, we can see that our new shell has a full set of capabilities. However, this capabilities only apply to resources owned by our user namespace. Since we have not created any other namespace, our user namespace doesn't own any resources. Indeed, if we try, e.g., to change the address of a network interface, we still get a permission denied error, since the network interface lives in the original network namespace, where we have no capabilities. We will also get a permission denied if we try to mount a filesystem, and so on.

Now exit from the shell and run

```
unshare -r -n
```

The `unshare` command will now create a user namespace and then also a network namespace. The user namespace will become the owner of the new network namespace, and if we try to assign an address to the loopback interface (the only one that lives in the new network namespace) we will succeed. Instead, trying to mount a filesystem will still fail, because we have not created a new mount namespace. Adding `-m` to the above `unshare` command will enable this capability too.

4 Control Groups

While namespaces can be used to hide and create private copies of all the system entities, they are not sufficient in isolating sets of processes so that they cannot interfere with each other. Processes may interfere also by abusing the system resources, e.g., allocating to much memory, using to much CPU time, or disk and network bandwidth. To properly implement containers, therefore, we also need to limit the usage of resources by the processes that live in the container. This is another thing that was not done very well before the introduction of *control*

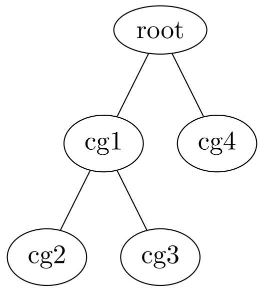

Figure 2: An example cgroup hierarchy

groups (often abbreviated in *cgroups*). The problem is that, before enforcing a limit on a set of processes, we need to know which processes belong to the set, and the processes must not be able to escape from the set. We would also like some flexibility in the definition of the set. Traditional Unix has a concept of process groups, but unfortunately any process is free to enter or leave a group. Processes are also grouped by user id (the user that is running them), but this is not very flexible; moreover, processes may switch their user id when they execute setuid programs.

Control groups, instead, are groups explicitly created by the administrator, who can later assign processes to them. The administrator may setup the system so that processes cannot escape their control group.

There are two implementations of the control groups framework in Linux: cgroups version 1 and cgroups version 2. Version 1 has been deprecated, so we only talk about version 2.

Control groups can be organized in a tree-shaped *hierarchy*, like the one in Fig. 2. Each process in the system must belong to exactly one control group in the hierarchy, and therefore the hierarchy is a partition of the system processes. When a process creates a child process, the child inherits the control group of its parent.

Control groups are managed by interacting with a pseudo-filesystem which is usually mounted on `/sys/fs/cgroup`. New cgroups can be defined by creating subdirectories in this file directory. For example, the hierarchy of Figure 2 can be created in the following way (as root):

```
cd /sys/fs/cgroup
mkdir cg1 cg4
mkdir cg1/cg2 cg1/cg3
```

Each new directory is automatically populated with a set of pseudo-files that let us interact with the corresponding cgroup. In particular, the `cgroup.procs` pseudo-file can be used to list the processes that belong to the cgroup (when read) or to move processes into the cgroup (by writing their pid into it). A simple `rmdir` can be used to remove a cgroup, provided that its `cgroup.procs` file is empty and that it doesn't have any sub-cgroup.

Once we have grouped processes into cgroups, we can start controlling their resources. In particular, a set of controllers may be active in any cgroup:

memory limits the amount of main memory used by each cgroup;

cpu limits the maximum fraction of CPU that each cgroup may use and may schedule the CPU based on cgroups *weights*;

cpuset on multi-cpu systems, limits the set of CPUs that may be used by each cgroup;

pids limits the number of processes that can be created in a cgroup.

io limits access to I/O devices.

Each of these controllers can be configured by writing into its own set of pseudo-files, where the pseudo-files of the memory controller have names starting with "`memory.`", and so on. The idea is that each cgroup controls the resources of its immediate children, by using the resources received from its parent. So, for example, we may limit the maximum amount of CPU time used by cg1 by writing into `/sys/fs/cgroup/cg1/cpu.max`. This will also limit the maximum total CPU time that can be used by cg2 and cg3.

5 Docker

Docker is the most popular implementation of containers for Linux and, recently, also for Windows. In Linux it uses the namespaces, control groups and capabilities features, together with other features like seccomp, to isolate containers from the host and from each other.

For a long time, however, Docker has not used the *user* namespaces, and it still doesn't use them by default. We can check this by creating and entering a container:

```
docker pull ubuntu
docker run -it ubuntu /bin/sh
```

We obtain a root prompt and, if we lookup the the new shell from *outside* the container, we can see that it runs as root even there[6] The only thing that limits the powers of this shell is that Docker has removed some capabilities (for example, we cannot load/unload modules in the kernel, since we lack the `CAP_SYS_MODULE` capability) and that the other (non user) namespaces hide the system resources from us. However, we still have root access to the large attack-surface of the host kernel, and we have to trust the Docker developers that they really have thought of all the possible things that root may abuse to

[6] Note that the `docker` program is not suid-root. The shell is running as root because the processes running inside the container are created by the `containerd` daemon, which runs as root. The `docker` command is just a client program that talks to `containerd` using a socket.

escape from the container. At least one thing should be kept in mind: whoever has the right to run the `docker` command can become root in no time, by just typing this:

```
docker run -it -v /:/shared ubuntu /bin/sh
```

This mounts the host root directory on the `shared` directory inside the container. Now the user has root access to the full host filesystem.

For this reason, Docker containers are considered less-secure than virtual machines. User namespaces should improve the container security, since their purpose is to remove the need to run programs with real root privileges inside containers. However, user namespaces are very complex and had to be retrofitted into an already large and complex kernel. Inevitably, a long list of security-critical bugs have been found in the implementation, thus creating a climate of distrust among container developers. That said, Docker now implements an option for user namespaces[7], so things may improve in the future.

[7]See `https://docs.docker.com/engine/security/userns-remap/`.