BLOG

# Linux Capabilities in OpenShift

November 23, 2020 | by Alexandre Menezes

## Introduction and Goals

The purpose of this article is to explain in depth how capabilities are implemented in Linux and why they can't be used to it's full extent in Kubernetes or OpenShift without developing some external tools to handle switching between superusers and non root users between process calls, or in other words, between runc calling a container and the container process when runc is controlled by a container engine such as CRIO under OpenShift or Kubernetes.

It's also an attempt to demonstrate a few alternatives to narrow down permissions for containers using capabilities. While doing that, we try to add our security considerations for each case.

In the end we discuss future developments that will certainly change this scenario in Kubernetes and OpenShift.

## Disclaimer - what this article is not

It's important to know that, even though the content presented here has pretty solid references, it's still a matter of opinion in many circles and we do not intend to be normative in any way. Besides that a full security assessment is necessary on any deployment in order to determine the weakest links in the chain and define the proper use of the technologies presented here. We do not provide, by any means, a security solution of this nature here.

That said, any use of the techniques presented here, specially in production environments, is at your own risk.

## What are Linux Capabilities?

We know that Linux has two types of users: Privileged and Unprivileged. Processes running under privileged users will bypass certain kernel permission checks and will be capable of doing almost everything in the system. Processes running under non-root, unprivileged, users are subjected to kernel permission checks which means the process credentials will be verified. So permissions granted for the user ID, group ID and/or supplementary group ID will tell what that specific process can do.

Some tasks are only allowed to be performed by the root user, user ID zero or superuser. In order to allow an unprivileged process to perform those tasks certain flags were created. Those flags are used by the kernel to check the process permissions for a specific task. These are what we call capabilities. In other words capabilities grant granular permissions on specific "privileged" tasks to unprivileged processes.

Taking from man pages we get this definition: "Starting with kernel 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled." Check man(7) capabilities.

A very repeated example in books and blogs is the one where we need our application to bind a socket to a privileged port. Technically speaking, they are called the "well known port numbers", ranging from 0 to 1024, and were defined in RFC 1340. One of the most important pillars of internet's networking. And they can be used for both TCP and UDP protocols.

The story here is simple. We have an application that we want to serve on port 80 or 443. But we don't want that process to be privileged. Capability seems to be an amazing solution for that. We then use CAP_NET_BIND_SERVICE and nothing else. That should be enough to bind our service to a port running an application process under an unprivileged user.

Example:

If I try to use nc to work on port 443 without becoming root that's what we get:

```
$: nc -lvu 443
nc: Permission denied
```

If we are sudoers and we can change to root that's the result:

```
$: sudo nc -lvu 443
[sudo] password for alex:
Listening on [0.0.0.0] (family 0, port 443)
```

The problem with that is that now root is running a process with it's full set of capabilities to do anything in the system.

```
ps aux | grep 'nc -lvu 443'
root     14169  0.0  0.0  72708  4284 pts/0    S+   10:22   0:00 sudo nc -lvu 443
root     14170  0.0  0.0  13592  1092 pts/0    S+   10:22   0:00 nc -lvu 443
```

It opens a second shell on sudo and runs the command with full privileges. Let's check the capabilities invested here. We can find it under /proc/**process number**/status and grep for the Cap expression:

```
$: grep Cap /proc/14170/status
CapInh: 0000000000000000
CapPrm: 0000003fffffffff
CapEff: 0000003fffffffff
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000
```

We explain in detail in later sections of this article each field we see here. Let's just follow through and interpret those with the command capsh in the effective capability set that is the third one. Here we have all capabilities that the user root raises when it initiates a process:

```
capsh --decode=0000003fffffffff
0x0000003fffffffff=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_
immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_ch
root,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_leas
e,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read
```

What if I want to narrow those capabilities to a subset of those permissions? Having only the capability to bind that process on a privileged port number? That we show a bit further on this article.

So, at the moment we learn what capabilities are, the idea of having a white list of specific tasks as a security mechanism narrowing down the permissions we would like to give to our applications immediately pops up in our minds. But this subject is not that simple as it seems in the containerized world.

When we talk about container platforms, container engines and container runtimes we have a very complex and not very clear path on how to configure capabilities for containers. And worse than that, at the time of this writing, some of the most recent features in the Linux kernel are not yet addressed by those platforms.

In the case of the example above one of the most intriguing questions is why can't I bind a server or service with non-root user to a socket on a privileged low number port inside a container running on Kubernetes or OpenShift if the container has its own network namespace? Even with CAP_NET_BIND_SERVICE!

First of all the network stack is a clone of the root namespace network stack and is compliant to all standards and RFCs such as the one cited above. And that is simply because the protocols are the same and use the same code. It leads us to think smart and ask the same question but with reverse logic. Why can't I bind to a non privileged port then? Is there really any good reason not to do so?

But let's say we have a use case for this and maybe other use cases that will pass through the same problem that is analyzing the required capabilities to have a "semi-privileged" environment (or processes + binaries + platform). Then we need to go deeply into the weeds of how the Linux kernel treats those privileges.

## Process Capabilities and File Capabilities

One important note before diving into this subject is that both processes and files can have capability sets. The process capabilities are tied to its user or inherited from its parent process. And here I may refer to process kind of interchangeably with threads or tasks. If needed I'll specify whether it's a parent process or a child one. And when we refer to file capabilities we talk about another level of permission that is encoded in the binary file extended attributes being run by that process. Which means that if the process has proper capabilities but the binary that the process is trying to run doesn't it may be denied due to the absence of a capability of some sort. That we'll explore a bit further. But it's important to know that both the process and the binary it runs must be checked when troubleshooting capabilities.

If you want to take a look at the history behind file capabilities check this article:

Fixing CAP_SETPCAP by Jake Edge, 2007 https://lwn.net/Articles/256519/

### Process Capability Sets

Things begin to get more complicated when we look closer to the Linux code base. How are Linux Capabilities implemented? At a high level overview we used the word flags. Ok. But what flags? So capabilities are stored in a data structure that can be associated with processes and files. We've said that already.

And, when we talk about processes, the information to manage and limit the action space for those specific privileged tasks is organized into 5 different sets. Below is a summary of the documentation that you can check in full here: capablities(7).

`Permitted Capabilities`

Is the superset that limits what can be in Effective and Inherited capabilities set.

`Inherited Capabilities`

Is the set of capabilities to be preserved when a program calls execve(2) to run another one that would inherit those capabilities.

> What is important here is that if the caller program is not running under a privileged user those capabilities won't be preserved and then ambient capabilities must be used in that case.

`Effective Capabilities`

"This is the set of capabilities used by the kernel to perform permission checks for the thread."

`Bounding Capabilities`

This is another layer of capability limitation applied to the thread when it tries to raise a new capability.

`Ambient Capabilities`

"This is a set of capabilities that are preserved across an execve(2) of a program that is not privileged."

The problem it's trying to solve is described in details in the article below:

capabilities: Ambient capabilities By Andy Lutomirski, 2015 (https://lwn.net/Articles/636533/).

> ⚠️ A few comments about containers and orchestration platforms.
>
> As of this point in time CRIO doesn't have the ability to handle ambient capabilities as we see in the portion of code below:

```
func setupCapabilities(specgen *generate.Generator, capabilities *pb.Capability) error {
        // Remove all ambient capabilities. Kubernetes is not yet ambient capabilities aware
        // and pods expect that switching to a non-root user results in the capabilities being
        // dropped. This should be revisited in the future.
        specgen.Config.Process.Capabilities.Ambient = []string{}
```

> That source code can be found here if it's not changed by the time you read it.
>
> And the discussion around ambient capabilities can be found here. It's an issue opened in November 25 of 2017 and it remains under discussion by October 2020. Further in this article we discuss some of the alternatives that we can take to accomplish what we need in this case.

## File Capability Sets

And when we talk about file capabilities they are stored in the file extended attributes. And we have:

"The file capability sets, in conjunction with the capability sets of the thread, determine the capabilities of a thread after an execve(2)."

`Permitted Capabilities`

If set in the file they are automatically permitted to the thread running it.

`Inherited Capabilities`

It sums up with the thread's inherited capability set to get the final one.

`Effective Capabilities`

And finally, on files effective capabilities are just a bit that enables all permitted capabilities to be added to the final effective capability set of the thread.

> In other words file capabilities extend the thread's capabilities.

## Capability-aware programs:

There are two types of programs in this case. The ones in binary files that were marked in their attributes as having capabilities to be addedd to the thread with the effective bit set to true and the ones that actually recognize and understand capabilities because they are running system calls in their code defined in libcap in order to manipulate capability information and configuration.

On the capabilities(7) man page under the section "Safety checking for capability-dumb binaries" it becomes clear what it means.

**Example with All Capabilities Included - The setuid bit Set**

Testing in a Ubuntu Xenial distro with the ping command we can see that it doesn't have any capabilities associated with it.

`getcap /bin/ping` doesn't output anything.

But when looking at the `ls -l /bin/ping` we can see:

```
-rwsr-xr-x 1 root root 64424 Jun 28  2019 /bin/ping
```

Look at the `s` in the permission field. It means that `setuid` is set to true and that the execution of the file will happen with, in this case, the owner's permissions which happens to be root. Therefore all capabilities would be included here to send a simple ICMP message over the network.

**Example with an Specific Capability Set**

Let's try the same in a CentOS 7 distro.

```
$ ls -l /usr/bin/ping
-rwxr-xr-x. 1 root root 66176 Aug  4  2017 /usr/bin/ping
```

First of all we see no `s` in the permission field. Now let's get caps.

```
$ getcap /usr/bin/ping
/usr/bin/ping = cap_net_admin,cap_net_raw+p
```

Here the permitted capability set has the caps above and the effective set has the bit flipped to true which adds those 2 permitted capabilities to the permitted capability set and effective capability set belonging to the process running it that may be under a regular user. That's how it get permission to execute the tasks. But it's important to say that as long as you run whatever binary it is it will have those capability raised during all the lifetime of that thread because they are configured in the extended attributes on the file system. Even if the task it needed to be performed runs only once at during the process lifetime.

#### Example Cap Dumb File

The nc `netcat` application is an example of a capability dumb file. As far as I know in most linux distributions it won't be installed with the `s` bit set or the any file capability at all. So you need to be root if you want for example to bind a lower port number socket under normal kernel circumstances. Because of that, it's a very fine example to craft the cap_net_bind_service in many ways and see what happens. This is why it is one of our main examples here.

## Verifying and Tracing Capabilities:

As we saw in the previous sections, binaries can be capability aware. That means that they can understand and manipulate capabilities on the fly. So new capabilities can be raised, lowered or dropped.

That's one of the reasons why the tool `capsh` may not be the best way to analyse capabilities since it "greps" the actual state of the thread's capability sets. Of course it can be used to test binaries ad hoc and let us understand a bit how our application is depending on capabilities to accomplish some tasks.

#### capsh with normal user

We can't see any capabilities in the Current capability set, that must mach the effective one I guess here.

```
capsh --print -- -c "ping 127.0.0.1"
Current: =
Bounding set
=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_
bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrac
e,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,
cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,35,36
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
uid=1000(vagrant)
gid=1000(vagrant)
groups=993(docker),1000(vagrant)
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.108 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.037 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.089 ms
```

#### capsh with `sudo`

Here we can see the current capability set but just this and the bounding set. That's not actually helpful. Where are the other ones?

```
sudo capsh --print -- -c "ping 127.0.0.1"
Current: =
cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,35,36+ep
Bounding set
=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,35,36
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root)
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.042 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.086 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.083 ms
```

### How to get the other sets

Let's put a ping to loopback in the background:

```
ping 127.0.0.1 > /dev/null &
[1] 718
```

Now let's grep `Cap` from its status and remark that we can see all 5 capability sets:

```
grep Cap /proc/718/status
CapInh: 0000000000000000
CapPrm: 0000000000003000
CapEff: 0000000000000000
CapBnd: 0000001fffffffff
CapAmb: 0000000000000000
```

Now let's use `capsh --decode=` to decode it:

```
for line in $(grep Cap /proc/718/status | awk '{print $2}'); do capsh --decode=$line; done;
0x0000000000000000=
0x0000000000003000=cap_net_admin,cap_net_raw
0x0000000000000000=
0x0000001fffffffff=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,35,36
0x0000000000000000=
```

### Example: capsh adding capabilties

If we want to set capabilities with capsh we can. But we need to understand the privileges that capsh itself needs to raise those capabilities. And one very important consideration: older systems won't have the capsh version that comes with the `--addamb` parameter that allow us to craft the ambient capability set.

Using capsh with a normal user:

```
$ capsh --caps='cap_net_bind_service+eip' --user=alex --addamb='cap_net_bind_service' -- -c 'nc -lvu 443'
Unable to set capabilities [--caps=cap_net_bind_service+eip]
```

The non-root user can't set capabilities. Quickly analyzing the capsh command we have:

```
---caps='<comma separated list of caps here>+eip'
                                          |||__ p for permitted
                                          ||___ i for inherited
                                          |____ e for effective
--user=<your non root user>
--addamb='<the ambient capabilties set comma separated>' <-- Here we enter the final set we want our process to run.
```

Let's try with sudo:

```
$ sudo capsh --caps='cap_net_bind_service+eip' --user=alex --addamb='cap_net_bind_service' -- -c 'nc -lvu 443'
[sudo] password for alex:
Unable to set group list for user: Operation not permitted
```

So it's calling one of the setgid system calls. Let's add that to the caps set.

```
$ sudo capsh --caps='cap_net_bind_service,cap_setgid+eip' --user=alex --addamb='cap_net_bind_service' -- -c 'nc -lvu 443'
Failed to set uid=1000(user=alex): Operation not permitted
```

The `--user` parameter triggers some of the setuid system calls. For that we need to add that too:

```
$sudo capsh --caps='cap_net_bind_service,cap_setgid,cap_setuid+eip' --user=alex --addamb='cap_net_bin_service' -- -c 'nc -lvu 443'
unable to raise CAP_SETPCAP for AMBIENT changes: Operation not permitted
```

Finally, let's add that last capability:

```
sudo capsh --caps='cap_net_bind_service,cap_setgid,cap_setuid,cap_setpcap+eip' --user=alex --addamb='cap_net_bind_service' -- -c 'nc -lvu
443' q
Listening on [0.0.0.0] (family 0, port 443)
```

Now let's check this process with another terminal session:

```
ps aux | grep 'nc -lvu 443'
root     15436  0.1  0.0  65848  4432 pts/0    S+   17:01   0:00 sudo capsh --caps=cap_net_bind_service,cap_setgid,cap_setuid,cap_setpcap+eip
--user=alex --addamb=cap_net_bind_service -- -c nc -lvu 443
alex     15438  0.0  0.0  13592  1064 pts/0    S+   17:02   0:00 nc -lvu 443
```

Below we conclude that the user root starts the process running capsh with 4 capabilities to the classical fields inherited, permitted and effective: cap_net_bind_service, cap_setuid, cap_setgid and cap_setpcap. capsh on it's turn will call a child process with nc running under a non-root user. So there is a user switch that requires the ambient capabilties.

```
$ pstree -apu 15436
sudo,15436 capsh --caps=cap_net_bind_service,cap_setgid,cap_setuid,cap_setpcap+eip --user=alex --addamb=cap_net_bind_service -- -c nc -lvu
443
  └─nc,15438,alex -lvu 443
```

What if we take out the ambient capabilities but let the others though? We will get a permission denied message. That's why ambient capabilities were created. To address the limitations the older cap sets have when switching from root user to non-root user.

```
$ sudo capsh --caps='cap_net_bind_service,cap_setgid,cap_setuid,cap_setpcap+eip' --user=alex -- -c 'nc -lvu 443'
nc: Permission denied
```

So finally let's see what the process got in terms of capabilities:

```
grep Cap /proc/15438/status
CapInh: 00000000000005c0
CapPrm: 0000000000000400
CapEff: 0000000000000400
CapBnd: 0000003fffffffff
CapAmb: 0000000000000400
```

The inherited cap set has:

```
capsh --decode=00000000000005c0
0x00000000000005c0=cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service
```

The only cap that gets preserved is in the cap ambient set and goes to permitted and effective sets is:

```
capsh --decode=0000000000000400
0x0000000000000400=cap_net_bind_service
```

There we go. We have a non-root user running a process with just one effective capability. Anyway, other questions must be considered here. Remark that the inherited set has 3 powerful capabilites and the bounding set, that's out of scope here, has everything from the parent process. Is it possible to hack it somehow? That's another discussion. But the process has only one capability at the moment we see it like that.

**Example Using `getcap` to analyze file capabilities:**

```
$ which ping
/usr/bin/ping
$ getcap /usr/bin/ping
/usr/bin/ping = cap_net_admin,cap_net_raw+p
```

⚠️ Please remark that this ping binary comes from a CentOS machine. In other Linux distros you may find that there is no capability bit set at all and the ping command is actually running with root privileges with the setuid bit turned to true. But that's another story.

#### Using iovisor to trace capabilities

For people interested in monitoring or tracing live capability requests to the kernel I recommend taking a look at the iovisor project at https://www.iovisor.org and at https://github.com/iovisor. I'm particularly interested for the sake of this article in the BCC tools. Specially the one developed by Brendan Gregg that you can find in the bcc project at ./tools/capable.py.

We already talked about dynamic capability manipulation. This is the tool that can grab some information from the kernel in poll mode like regarding process capabilities on the fly. Some explanation about that can be found here and also here.

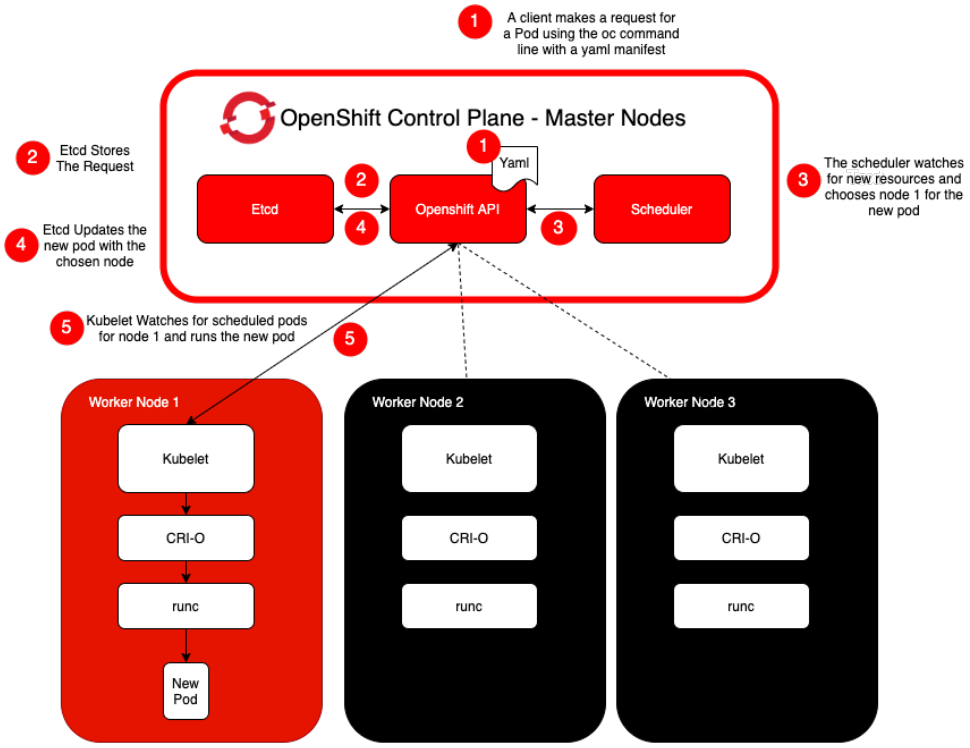
#### libcap-ng

Another possible interesting project for that matter is the libcap-ng that you can find here https://people.redhat.com/sgrubb/libcap-ng/. Specially for those interested in working with some custom developed application and don't want to use the traditional libcap provided by Linux.

## Capabilities and Containers

Let's take a few steps back and try to picture what we have here. Capabilities are assigned to processes and files. Containers can be defined as isolated processes with their own file systems into namespaces and cgroups. Runtimes create or, better, automate the creation of those isolated processes. Container engines are the piece that talks to the actual runtime passing the configuration parameters for container creation. Orchestration platforms have the abstraction layer that will use those engines to spin up all their pods and deployments.

Let's recap here. The flow of configuration could be simplified as:




First and probably most important information here is: where goes the capability information in each configuration phase in order to get to the container itself?

Basically, the configuration flow for containers would be using:

1. security context from the container and pod definition with the SCCs application, listing what are the dropped default capabilities and the added capabilities.
2. Default capabilities included in CRI-O configuration.
3. The capability sets available in runc configuration.

```
1) SecurityContext (Container Spec) + SCCs
   |
   -->  2) Default Capabilities List (crio.conf)
                 |
                 -->  5 different process cap sets (runc conf.json)
```

```
1)
type SecurityContext struct {
        // The capabilities to add/drop when running containers.
        // Defaults to the default set of capabilities granted by the container runtime.
        // +optional
        Capabilities *Capabilities `json:"capabilities,omitempty" protobuf:"bytes,1,opt,name=capabilities"`

  ...
```

which can be found here and is part of your regular pod or deployment yaml. And the type Capability:

```
type Capability string

// Adds and removes POSIX capabilities from running containers.
type Capabilities struct {
        // Added capabilities
        // +optional
        Add []Capability `json:"add,omitempty" protobuf:"bytes,1,rep,name=add,casttype=Capability"`
        // Removed capabilities
        // +optional
        Drop []Capability `json:"drop,omitempty" protobuf:"bytes,2,rep,name=drop,casttype=Capability"`
}
```

Finally CRI-O has also a default capability field:

```
2)
...
default_capabilities=[] List of default capabilities for containers. If it is empty or commented out, only the capabilities defined in the
container json file by the user/kube will be added.

The default list is:

  default_capabilities = [
          "CHOWN",
          "DAC_OVERRIDE",
          "FSETID",
          "FOWNER",
          "SETGID",
          "SETUID",
          "SETPCAP",
          "NET_BIND_SERVICE",
          "KILL",
  ]
  ...
```

Which you can find here

3)Finally runc will also have a structure to hold capability information:

---

For Linux-based systems, the `process` object supports the following process-specific properties.

...

- **`capabilities`** (object, OPTIONAL) is an object containing arrays that specifies the sets of capabilities for the process. Valid values are defined in the man(7) capabilities, such as `CAP_CHOWN`. Any value which cannot be mapped to a relevant kernel interface MUST cause an error. `capabilities` contains the following properties:
    - **`effective`** (array of strings, OPTIONAL) the `effective` field is an array of effective capabilities that are kept for the

process.
- **`bounding`** (array of strings, OPTIONAL) the `bounding` field is an array of bounding capabilities that are kept for the process.
- **`inheritable`** (array of strings, OPTIONAL) the `inheritable` field is an array of inheritable capabilities that are kept for the process.
- **`permitted`** (array of strings, OPTIONAL) the `permitted` field is an array of permitted capabilities that are kept for the process.
- **`ambient`** (array of strings, OPTIONAL) the `ambient` field is an array of ambient capabilities that are kept for the process. ...

---

So something is missing in those types, right? There is no field for the 5 different capability sets in the Pod/Container Spec type or in the CRIO configuration file. Here is where things begin to get muddy. If for some reason there is a need for ambient capabilities, which is exactly the case when we want to use non-root users, those capabilities will be dropped in the end of the line for reasons we explored in previous sections. That said if the binary file is not capability aware or has it's own permitted set configured with the effective bit set to true with proper permissions it will get a permission denied error.

## Working with Capabilities on OpenShift

### 1. Dropping Capabilities from the root user

#### Setting up a custom SCC to restrain the root access

Thinking only about users and permissions, i. e., not going into Linux Security Modules, to restrict the root access on some process we may want to cut out it's capabilities by dropping them.

That would require us to list all possible capabilities and put them in the required drop capabilities of the Security Context Constraint in OpenShift. And apart from that we will also want to restrict this container user from being able to access host permissions on the worker node it's running. It seems reasonable to build this SCC from the anyuid built-in SCC. Below there is an example for the NET_BIND_SERVICE capability:

```
allowHostDirVolumePlugin: false
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: true
allowPrivilegedContainer: false
allowedCapabilities:
  - NET_BIND_SERVICE
apiVersion: security.openshift.io/v1
defaultAddCapabilities: null
fsGroup:
  type: RunAsAny
groups:
- system:cluster-admins
kind: SecurityContextConstraints
metadata:
  annotations:
    kubernetes.io/description: anyuid provides all features of the restricted SCC
      but allows users to run with any UID and any GID.
  generation: 1
  name: anyuid-netbind
priority: 10
readOnlyRootFilesystem: false
requiredDropCapabilities:
- MKNOD
- SETGID
- SETFCAP
- SETPCAP
- SETUID
- AUDIT_CONTROL
- AUDIT_WRITE
- BLOCK_SUSPEND
- CHOWN
- DAC_OVERRIDE
- DAC_READ_SEARCH
- FOWNER
- FSETID
- IPC_LOCK
- IPC_OWNER
- KILL
- LEASE
- LINUX_IMMUTABLE
- MAC_ADMIN
- MAC_OVERRIDE
- NET_ADMIN
- NET_BROADCAST
- NET_RAW
- SYS_ADMIN
- SYS_BOOT
- SYS_CHROOT
- SYS_MODULE
- SYS_NICE
- SYS_PACCT
- SYS_PTRACE
- SYS_RAWIO
- SYS_RESOURCE
- SYS_TIME
- SYS_TTY_CONFIG
- SYSLOG
- WAKE_ALARM

runAsUser:
  type: RunAsAny
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
users: []
volumes:
- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- projected
- secret
```

Then we can apply it to the cluster by running:

```
oc apply -f anyuid_netbind_scc.yaml
```

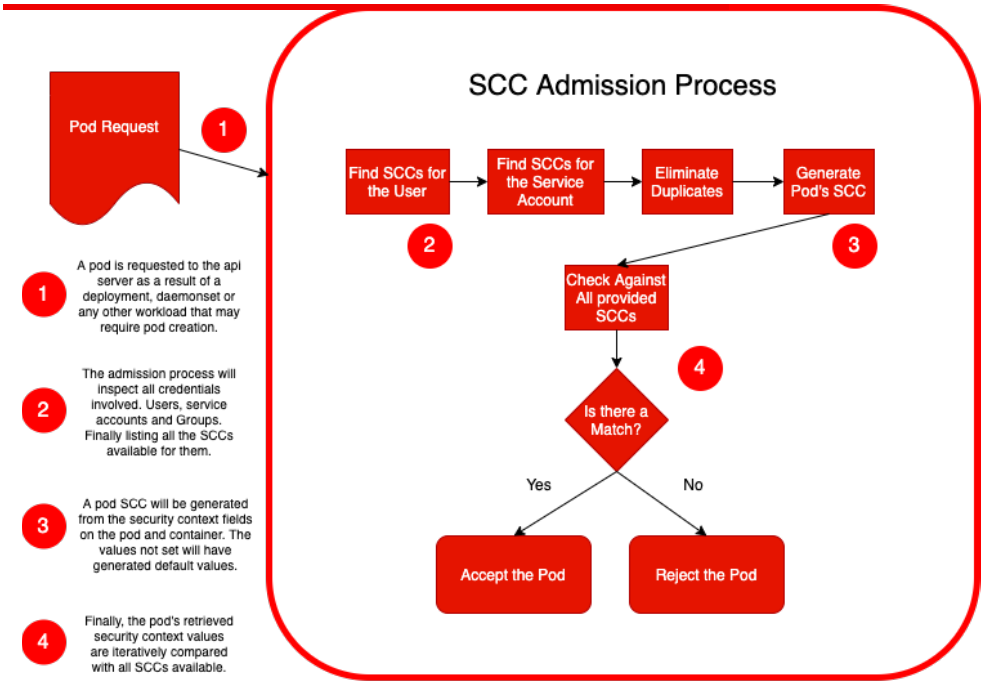
### Setting up a Security Context in the Pod to run the specific capability

Once we have our SCC in place we can create our deployment with proper RBAC permissions. Here we have a simple pod using the Linux `nc` command to bind a socket using a low number port.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: anyuid-netbind-acc
  namespace: cap-test
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: anyuid-netbind-role
  namespace: cap-test
rules:
- apiGroups:
  - security.openshift.io
  resourceNames:
  - anyuid-netbind
  resources:
  - securitycontextconstraints
  verbs:
  - use
---
  apiVersion: rbac.authorization.k8s.io/v1
  kind: RoleBinding
  metadata:
    name: anyuid-netbind-rolebinding
    namespace: cap-test
  subjects:
  - kind: ServiceAccount
    name: anyuid-netbind-acc
    namespace: cap-test
  roleRef:
    kind: Role
    name: anyuid-netbind-role
    namespace: cap-test
    apiGroup: rbac.authorization.k8s.io
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: anyuid-netbind
  namespace: cap-test
spec:
  replicas: 1
  selector:
    matchLabels:
      name: anyuid-netbind
  template:
    metadata:
      labels:
        name: anyuid-netbind
    spec:
      serviceAccountName: anyuid-netbind-acc
      containers:
        - name: anyuid-netbind
          image: nicolaka/netshoot
          command: ["sleep"]
          args: ["infinity"]
          imagePullPolicy: Always
          securityContext:
            capabilities:
              add: ["NET_BIND_SERVICE"]
```

Remark in the end of the file that we don't need to put the DROP capabilities in the security context. That is because the OpenShift SCC admission process will fill in what's not configured according to the SCC applied to this deployment. If you want to take a detailed look in this process checkout this link: https://www.openshift.com/blog/managing-sccs-in-openshift

And the diagram below, extracted from that article, explains how it works:




So now let's check what we have:

```
oc get pods -n cap-test
NAME                              READY   STATUS    RESTARTS   AGE
anyuid-netbind-857977ccb6-hjgkk   1/1     Running   0          6s
```

Run `oc get pods anyuid-netbind-857977ccb6-hjgkk -o yaml`.

You will see that all dropped capabilities were added to the container spec in the security context field.

If we log into the container we can see it's running the first process as `nc -lvu 443`.

```
oc exec -it anyuid-netbind-857977ccb6-hjgkk -n cap-test -- /bin/bash
bash-5.0# ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 nc -lvu 443
    7 root      0:00 /bin/bash
   13 root      0:00 ps aux
```

### Security Considerations with this approach

Inside the container:

```
bash-5.0# ls -l
total 8
drwxr-xr-x    1 root     root           220 May  6 04:45 bin
drwxr-xr-x    5 root     root           360 Jun 29 14:29 dev
drwxr-xr-x    1 root     root            25 Jun 29 14:29 etc
drwxr-xr-x    2 root     root             6 Apr 23 13:10 home
drwxr-xr-x    1 root     root          4096 May  6 04:45 lib
drwxr-xr-x    2 root     root            34 May  6 04:45 lib64
drwxr-xr-x    5 root     root            44 Apr 23 13:10 media
drwxr-xr-x    2 root     root             6 Apr 23 13:10 mnt
drwxr-xr-x    2 root     root             6 Apr 23 13:10 opt
dr-xr-xr-x  255 root     root             0 Jun 29 14:29 proc
drwx------    1 root     root            26 Jun 29 14:36 root
drwxr-xr-x    1 root     root            21 Jun 29 14:29 run
drwxr-xr-x    1 root     root          4096 May  6 04:45 sbin
drwxr-xr-x    2 root     root             6 Apr 23 13:10 srv
dr-xr-xr-x   13 root     root             0 Jun 10 19:38 sys
drwxr-xr-x    2 root     root             6 May  6 04:46 termshark_2.1.1_linux_x64
drwxrwxrwt    1 root     root            46 May  6 04:46 tmp
drwxr-xr-x    1 root     root            19 May  6 04:45 usr
drwxr-xr-x    1 root     root            66 May  6 04:45 var
```

All directories are owned by root. Now let's try something forbidden here:

```
bash-5.0# cd var
bash-5.0# ls -l
total 0
drwxr-xr-x    1 root     root            29 Apr 23 13:10 cache
dr-xr-xr-x    2 root     root             6 Apr 23 13:10 empty
drwxr-xr-x    1 root     root            34 May  6 04:45 lib
drwxr-xr-x    2 root     root             6 Apr 23 13:10 local
drwxr-xr-x    3 root     root            20 Apr 23 13:10 lock
drwxr-xr-x    2 root     root             6 Apr 23 13:10 log
drwxr-xr-x    2 root     root             6 May  6 04:45 mail
drwxr-xr-x    2 root     root             6 Apr 23 13:10 opt
lrwxrwxrwx    1 root     root             4 May  6 04:45 run -> /run
drwxr-xr-x    1 root     root            30 May  6 04:45 spool
drwxrwxrwt    2 root     root             6 Apr 23 13:10 tmp
```

Deleting the mail folder for example:

```
bash-5.0# rm -rf mail
bash-5.0# ls -l
total 0
drwxr-xr-x    1 root     root            29 Apr 23 13:10 cache
dr-xr-xr-x    2 root     root             6 Apr 23 13:10 empty
drwxr-xr-x    1 root     root            34 May  6 04:45 lib
drwxr-xr-x    2 root     root             6 Apr 23 13:10 local
drwxr-xr-x    3 root     root            20 Apr 23 13:10 lock
drwxr-xr-x    2 root     root             6 Apr 23 13:10 log
drwxr-xr-x    2 root     root             6 Apr 23 13:10 opt
lrwxrwxrwx    1 root     root             4 May  6 04:45 run -> /run
drwxr-xr-x    1 root     root            30 May  6 04:45 spool
drwxrwxrwt    2 root     root             6 Apr 23 13:10 tmp
```

But didn't we restrict the root user? That's the first misconception that I wanted to address. Let's check the process running `nc`.

This is the process number 1.

```
ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 nc -lvu 443
    7 root      0:00 /bin/bash
   44 root      0:00 ps aux
```

Now let's get it's capability sets:

```
grep Cap /proc/1/status
CapInh: 0000000000000400
CapPrm: 0000000000000400
CapEff: 0000000000000400
CapBnd: 0000000000000400
CapAmb: 0000000000000000
```

Now let's decode it:

```
bash-5.0# for line in $(grep Cap /proc/1/status | awk '{print $2}'); do capsh --decode=$line; done;
0x0000000000000400=cap_net_bind_service
0x0000000000000400=cap_net_bind_service
0x0000000000000400=cap_net_bind_service
0x0000000000000400=cap_net_bind_service
0x0000000000000000=
```

Only NET_BIND_SERVICE is allowed. And remark that there are no ambient capabilities at all.

If we look at our bash session it's quite the same:

```
bash-5.0# grep Cap /proc/7/status
CapInh: 0000000000000400
CapPrm: 0000000000000400
CapEff: 0000000000000400
CapBnd: 0000000000000400
CapAmb: 0000000000000000
```

```
bash-5.0# for line in $(grep Cap /proc/7/status | awk '{print $2}'); do capsh --decode=$line; done;
0x0000000000000400=cap_net_bind_service
0x0000000000000400=cap_net_bind_service
0x0000000000000400=cap_net_bind_service
0x0000000000000400=cap_net_bind_service
0x0000000000000000=
```

Conclusion: it's still the root user and owns a lot of the files and folders that are crucial to the system. So it's possible to delete, change and create files. It's probably possible to install things as well. It's the same root user as the host root, but limited by a few tricks on host access because of the anyuid SCC. Besides that any other process inside that container gets that capability as we saw with the bash session. If one can manage to access the host file system... I think I don't need to say more.

## 2. Using the setuid bit in the binary with a non root user

### Setting up a container image with binaries setuid/setgid capable

Remeber the ping binary and why we as normal users can ping in Ubuntu for example? Let's try that approach.

So from the previous example let's check the permissions on the `nc` binary:

```
bash-5.0# ls -l /usr/bin/nc
-rwxr-xr-x    1 root     root         35184 Dec  4  2018 /usr/bin/nc
```

Others can execute the binary as long they don't try low port numbers. In that case the user must be root or have the NET_BIND_SERVICE capability.

What if we can allow regular users to run this binary with root permissions by default?

Here it goes. We need to change the binary permission to setuid bit in the owner's permission. For that we can create a docker file like the one below. And in that case we put our entry point as our `nc` command binding the privileged port 443:

```
FROM nicolaka/netshoot
RUN chown u+s /usr/bin/nc
CMD ["/usr/bin/nc", "-lvu", "443"]
```

After building a new image with this Dockerfile above let's test using a complete random non-root user deployment without any security context request or SCC.

`oc apply -f nonroot-deployment-setuid.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nonroot-setuid
  namespace: cap-test
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nonroot-setuid
  template:
    metadata:
      labels:
        name: nonroot-setuid
    spec:
      containers:
        - name: nonroot-setuid
          image: quay.io/acmenezes/nc-setuid:test
          imagePullPolicy: Always
```

Now we have a second pod running:

```
oc get pods -n cap-test
NAME                              READY   STATUS    RESTARTS   AGE
anyuid-netbind-857977ccb6-hjgkk   1/1     Running   0          77m
nonroot-setuid-c79498fb8-87ptr    1/1     Running   0          6s
```

Let's check it:

```
oc exec -it nonroot-setuid-c79498fb8-87ptr -n cap-test -- /bin/bash
bash-5.0$ ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 /usr/bin/nc -lvu 443
    6 10006600  0:00 /bin/bash
   11 10006600  0:00 ps aux
```

First of all look the user running `nc`. It's root. And remark that now we have an `s` instead of an `x` in the owner permission.

```
bash-5.0$ ls -l /usr/bin/nc
-rwsr-xr-x    1 root     root         35184 Dec  4  2018 /usr/bin/nc
```

But check the bash user. It's not root. And try to delete something that doesn't belong to it:

```
bash-5.0$ rm -rf /var/mail
rm: can't remove '/var/mail': Permission denied
```

#### Security Considerations with this approach

Better now right? But remember the binary will behave like that for any non root user that tries it. It will run as root. So the extent of the features embedded in the executable will dictate how dangerous it is. Being netcat as our example we can scan through ports, manipulate TCP and UDP sockets and maybe try some sort of DDoS attack. So the security gap must be verified at the container image scanning and certification phase. But still we can do a little bit better. Check the next section.

### 3. Using File Capabilities

#### Using the setcap command in the container image

Instead of giving all root privileges on the binary we can limit it with file capabilities and here is how we do it. Create another docker image but this time using the setcap command:

```
FROM nicolaka/netshoot
RUN setcap 'cap_net_bind_service+ep' /usr/bin/nc
CMD ["/usr/bin/nc", "-lvu", "443"]
```

after doing `oc apply -f nonroot-deployment-capset.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nonroot-capset
  namespace: cap-test
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nonroot-capset
  template:
    metadata:
      labels:
        name: nonroot-capset
    spec:
      containers:
        - name: nonroot-capset
          image: quay.io/acmenezes/nc-cap:test
          imagePullPolicy: Always
```

Now we have a third pod running:

```
oc get pods -n cap-test
NAME                                READY   STATUS    RESTARTS   AGE
anyuid-netbind-857977ccb6-hjgkk     1/1     Running   0          99m
nonroot-capset-54d6c854c4-8bc29     1/1     Running   0          8s
nonroot-setuid-c79498fb8-87ptr      1/1     Running   0          22m
```

Let's check it:

```
oc exec -it nonroot-capset-54d6c854c4-8bc29 -n cap-test -- /bin/bash
bash-5.0$ ps aux
PID   USER     TIME  COMMAND
    1 10006600  0:00 /usr/bin/nc -lvu 443
    6 10006600  0:00 /bin/bash
   11 10006600  0:00 ps aux
```

Remark that it's running `nc` using a low port number but with a non-root user.

Now let's check it's permissions:

```
 ls -l /usr/bin/nc
-rwxr-xr-x    1 root     root         35184 Dec  4  2018 /usr/bin/nc
```

No setuid bit. Let's check it's capabilities:

```
grep Cap /proc/1/status
CapInh: 000000000004251b
CapPrm: 0000000000000400
CapEff: 0000000000000400
CapBnd: 000000000004251b
CapAmb: 0000000000000000
```

```
for line in $(grep Cap /proc/1/status | awk '{print $2}'); do capsh --decode=$line; done;
0x000000000004251b=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot
0x0000000000000400=cap_net_bind_service
0x0000000000000400=cap_net_bind_service
0x000000000004251b=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot
0x0000000000000000=
```

Remark that with permitted and effective sets only cap_net_bind_service is there. This are the process capability sets. Now let's check the file extended attributes from where it got the cap_net_bind_service:

```
getcap /usr/bin/nc
/usr/bin/nc = cap_net_bind_service+ep
```

**Security Considerations with this approach**

There we go. Refined from full root access on the file to just one cap. If the program had other powerful functions non related to what it needs to do in the application specific case we can block them by taking off some of the capabilities when it's possible. And it's important to highlight that since it's done in the container image the security duty to audit ant test it is on the container image scanning process because no SCC will block this. Another thing to take into consideration is that those capabilities are definitely held during all the lifetime of the pod/container. There is a way to do it temporarily raising capabilities only to perform certain tasks and then lower or drop them completely after that time.

## 4. Using a Helper Program to Set Ambient Capabilities

**Using the libcap-ng library to setup the environment**

The libcap-ng library was written by Steve Grubbs and aims to have a nicer easier api to work with capabilities and processes. Here you can have a better understanding of what it's all about.

So Kubernetes can't work with ambient capabilities, from it's resources and abstractions. And that's is due to not having this type implemented as an input to Pod or Container Specs. And this becomes a problem when a container spins up because the privileged process that creates it handles over the namespace and everything else to an unprivileged process. As a side effect the ambient capability set is cleaned up.

To recreate that Linux behavior we can have a helper program with proper privileges running before the actual container entry point binary to set the environment. Check this awesome blog post from Adrian Mouat that inspired this section. I'm taking this as the example and building the image as below:

```
FROM nicolaka/netshoot
COPY set_ambient /usr/bin/set_ambient
RUN setcap 'cap_net_bind_service+ep' /usr/bin/set_ambient
CMD ["/usr/bin/set_ambient", "/usr/bin/nc", "-lvu", "443"]
```

Now let's deploy it to OpenShift. Create the nonroot-deployment-setambient.yaml file.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nonroot-setambient
  namespace: cap-test
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nonroot-setambient
  template:
    metadata:
      labels:
        name: nonroot-setambient
    spec:
      containers:
        - name: nonroot-setambient
          image: quay.io/acmenezes/nc-setambient:test
          imagePullPolicy: Always
```

Then we run `oc apply -f nonroot-deployment-setambient.yaml` and we should have a 4th pod running.

```
oc get pods -n cap-test
NAME                                  READY   STATUS    RESTARTS   AGE
anyuid-netbind-857977ccb6-hjgkk       1/1     Running   0          27h
nonroot-capset-54d6c854c4-8bc29       1/1     Running   0          25h
nonroot-setambient-69d65865c4-dxp26   1/1     Running   0          31s
nonroot-setuid-c79498fb8-87ptr        1/1     Running   0          26h
```

Let's take a look a little closer:

```
oc exec -it nonroot-setambient-69d65865c4-dxp26 -n cap-test -- /bin/bash
bash-5.0$ ps aux
PID   USER     TIME  COMMAND
    1 10006600  0:00 /usr/bin/nc -lvu 443
    6 10006600  0:00 /bin/bash
   11 10006600  0:00 ps aux

bash-5.0$ grep Cap /proc/1/status
CapInh: 000000000004251b
CapPrm: 0000000000000400
CapEff: 0000000000000400
CapBnd: 000000000004251b
CapAmb: 0000000000000400

bash-5.0$ for line in $(grep Cap /proc/1/status | awk '{print $2}'); do capsh --decode=$line; done;
0x000000000004251b=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot
0x0000000000000400=cap_net_bind_service
0x0000000000000400=cap_net_bind_service
0x000000000004251b=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot
0x0000000000000400=cap_net_bind_service
```

So the biggest change here is that now we have that last set `CapAmb` filled with some value. And that is specific to that running instance of that process. If I login and try running `nc` by hand it won't work. I doesn't have permissions on the file.

```
oc exec -it nonroot-setambient-69d65865c4-dxp26 -n cap-test -- /bin/bash
bash-5.0$ nc -lvu 443
nc: Permission denied
```

On the other hand we now transfer the capability to the helper file that can run anything with CAP_NET_BIND_SERVICE:

```
bash-5.0$ /usr/bin/set_ambient /usr/bin/nc -lvu 80
Starting process with CAP_NET_BIND_SERVICE in ambient
Listening on [0.0.0.0] (family 0, port 80)
```

If somebody hacks the container this helper program is a liability to the system. Some other security measures must be put in place to avoid access to that file with this approach.

#### Security Considerations with this approach

This is better than having file capabilities if we can prevent access to the helper file for two main reasons. The first one is that the capabilities applied by the helper program are applied only on a single instance of that process. And the second reason is that if the application receiving that capability is capability aware, meaning that it's using libcap or libcap-ng to manipulate capabilities, it can raise the specific capability for a short period of time to accomplish some tasks and then lower it down or even drop it if it's

not going to use it anymore.

The greatest caveat here is that OpenShift is not aware of any of this. And this is just because the underneath Kubernetes is not there yet. The types we have to work with container specs are not ready for this level of detail. That is a work in progress.

### 5. Adding the needed capability as a default value to CRIO

CRI-O is the container engine for OpenShift as we saw that can also be used as the container engine for any kubernetes distribution. It has a configuration file that normally goes under /etc/crio/crio.conf.

And there we can find that: "default_capabilities=[] List of default capabilities for containers. If it is empty or commented out, only the capabilities defined in the container json file by the user/kube will be added." Taken from here --> https://github.com/cri-o/cri-o/blob/master/docs/crio.conf.5.md

So, if we put new capabilities on that field it will include those on the effective set of our containers by default. What happens with that is that any container in the platform will have those capabilities. Even containers that are part of kubernetes control plane.

Is it a good solution?

#### Security Considerations with this approach

Only if the entire cluster can be trusted and has a very limited set of users, in a non production enviroment for testing, it would make sense. I would strongly recommend avoiding this approach, specially if the capability given has some superuser setting such as setuid, setgid, setcap etc. Of course, if for some odd reason you need to lock in portions of memory to use huge pages in a predictable way or you want to encrypt portions of memory with no disk writing for security purposes then IPC_LOCK capability may be used. But kubernetes developers are not taking that capability into consideration when putting new features in the control plane pods such as kubeapi-server. In other words that would definitely impact the whole cluster with no guarantee or support of any kind.

## Some Possible use Cases

Multiple network vendors and telecom companies are developing quite cool applications that lay down one level below our regular apps. They are manging networking connections in a way that kubernetes can't provide us.

The business case for those are the edge networking, the IP video and audio multicasting and streaming, multiple called Container Networking Functions that can enrich data, secure communications, make services more flexible and performant between clusters, deploying the next generation 5G networks etc.

The needs imposed by those sometimes are low number port binding (as our examples), special on the fly network configurations, pinging with no priviledges etc. On node scope we see the need for special kernel modules, drivers, real time kernel, CPU pinning, Smart NIC, FPGA with dedicated access to devices etc.

We can see also some special uses on mounting priviledged paths, kernel module building and loading from containers (check the kmods-via-containers project), other host machine configurations (check the machine config operator project) and very specific memory setups that require precise memory allocations and/or memory encryption with swap disabled. This last one can be the case for the IPC_LOCK capabiltiy.

### Some General Security Guide Lines

When talking about root users running containers I would agree with LXCs position in this sentence:

```
"LXC upstream's position is that those containers aren't and cannot be root-safe...".
```

So yes, we can and should protect our host from the root user where possible using SELinux, apparmour, seccomp filtering and dropping capabilities that are not needed. But if the root user must be put in place then we need to trust the workloads it's running as well as their tasks.

For untrusted, or unknown customer workloads for example, then unpriviledged containers are the way to go. In kubernetes that would be even better if the user namespaces could be implemented, which is out of scope here. That can make the root container user map a random non-existent host user making it safer.

Meanwhile if capabilities are needed be aware there is no easy way to configure it from the security contexts that we find today in kubernetes pod's and container abstractions without facing some of those side effects.

## Conclusion

Finally, we talked about all types of capability sets present on Linux systems nowadays, how to check the capabilities in use for a particular process or binary in development phase, pointed out some possible projects and libraries to monitor capability requests on the fly and how all of that traslates to the container world and orchestration platforms with what we have at this point in time.

If you really need a non-root user + a couple of capabilities one way to get out with that is using file capabilities. That would require a strong image scanning process analysing all upgrades on your binaries to make sure there is nothing more that what is needed being delivered in the pipeline.

I hope this blog post is informative for you and also may serve as a reference point on subject for further exploration and testing. Thaks for reading! See you soon!

REFERENCES

**Books:**


```
Chapter 7. Process Credentials in
"Hands-On System Programming with Linux"
by Kaiwan N Billimoria, Packt Pub. 2018

Chapter 8. Process Capabilities in
"Hands On System Programming with Linux"
by Kaiwan N Billimoria, Packt Pub. 2018

Chapter 8. Linux Kernel Security, Capabilities, and Seccomp in
"Observability with BPF"
by David Calavera; Lorenzo Fontana
Published by O'Reilly Media, Inc., 2019

Chapter 11. Security in
"BPF Performance Tools: Linux System and Application Observability"
by Brendan Gregg
```




**Articles and Documentation:**

```
capabilities(7) from Man Pages
https://man7.org/linux/man-pages/man7/capabilities.7.html

getcap, setcap and file capabilities, 2017
https://www.insecure.ws/linux/getcap_setcap.html

Linux Capabilities: making them work in
"Proceedings of the Linux Symposium - Ontario, Canada 2008"
http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/33528.pdf

Ambient Capabilities
by Andy Lutomirski, 2015
https://lwn.net/Articles/636533/

Linux capabilities support for user namespaces
by Jake Edge, 2010
https://lwn.net/Articles/420624/

Fixing CAP_SETPCAP
by Jake Edge, 2007
https://lwn.net/Articles/256519/

Linux bcc Tracing Security Capabilities
by Brendan Gregg, 2016
http://www.brendangregg.com/blog/2016-10-01/linux-bcc-security-capabilities.html

Linux Capabilities: Why They Exist and How They Work
By Adrian Mouat
https://blog.container-solutions.com/linux-capabilities-why-they-exist-and-how-they-work

Linux Capabilities In Practice
By Adrian Mouat
https://blog.container-solutions.com/linux-capabilities-in-practice
```

### Videos:

```
Capable - Beginning at 00:10:00 - https://www.youtube.com/watch?v=44nV6Mj11uw
BSidesSF 2017 - Linux Monitoring at Scale with eBPF (Brendan Gregg & Alex Maestretti)
```

### Github Issues:

```
Kubernetes should configure the ambient capability set
https://github.com/kubernetes/kubernetes/issues/56374

Can't bind to privileged ports as non-root
https://github.com/moby/moby/issues/8460

Add support for ambient capabilities
https://github.com/moby/moby/pull/26979
```

**Special Thanks to:**
Matt Dorn
William C. Babilonia
Dave Baker

---

CATEGORIES

How-tos, Operators, Linux

< Back to the blog