Kernel Security Primitives of Linux Containers

Master's Thesis

ps1337

28.02.2019

XYZ University
Course: Computer Science

Abstract

Over the past years, Linux containers have been increasingly adopted by the industry, researchers and private parties. One reason for this is the possible boost in productivity in combination with the ability to save cost at the same time when compared to virtual machine environments. This thesis discusses and dissects the internals of today's containers.

Containers are a construct of existing kernel primitives rather than a feature directly implemented in the kernel. The main building blocks are namespaces and control groups in combination with the *capabilities* facility and the secure computing (*seccomp*) kernel feature. Namespaces define and isolate the environment that's visible for a specific process while control groups can be used to monitor and restrict resource usage. Container managers like *Docker* combine and configure these primitives in a user-friendly manner and provide sane defaults to allow spawning containers with minimal effort.

As of today, using and configuring specific mechanisms used for containerization requires elevated privileges. While it's possible to create containers as an unprivileged user, not all available isolation directives may be used in such a scenario. There were and are ongoing efforts to steadily minimize the permissions required to perform containerization. It's theoretically possible to remediate these limitation completely in the future.

By creating a survey regarding the usage of isolation directives in unit files of *systemd* services, it has been determined that the general adoption of this feature is low. Moreover, most directives are rarely used while a large number is not being used at all. In many cases their usage does not follow best practices or does not provide additional security because of the way they are being utilized.

Contents

List of Figures

List of Tables

Listings

Acronyms

ACL Access Control List. 80

API Application Programming Interface. 6, 14, 17, 82

BPF Berkeley Packet Filter. 70–72

CFQ Completely Fair Queuing. 9, 11, 64

CI/CD Continuous Integration/Continuous Delivery. 1, 77

CLI Command Line Interface. 25, 26, 36, 38, 46, 57, 69

CPU Central Processing Unit. 7–9, 65, 69

GID Group Identifier. 50–54, 71, 80, 81

HTB Hierarchical Token Bucket. 63

I/O Input/Output. 5, 9, 11, 63, 64

ICMP Internet Control Message Protocol. 13

IPC Inter Process Communication. 9, 32, 35, 47, 48

JSON JavaScript Object Notation. 26

KVM Kernel-Based Virtual Machine. 29, 31

LSM Linux Security Module. 73

LXC Linux Containers. 11–13, 27, 28, 57, 69, 76, 87

MAC Mandatory Access Control. 73

NoSQL Not Only SQL. 82

NTP Network Time Protocol. 16, 17

OCI Open Containers Initiative. 25–28, 30, 31, 42, 43, 50, 74

OOM Out Of Memory. 81

PaaS Platform as a Service. 88

PID Process Identifier. 32, 33, 35, 38, 39, 44, 45, 47, 50, 55, 59, 69, 80, 81

QDisc Queueing-Discipline. 63

RAM Random Access Memory. 11

REST Representational State Transfer. 25

RPC Remote Procedure Call. 26

SCONE Secure Linux Containers on Intel SGX. 88

SGX Software Guard Extensions. 88

SUID Set User ID. 13, 15, 76, 80, 87

UID User Identifier. 13, 16, 49–54, 58, 71, 80, 81

UML Unified Modeling Language. 69

UTS Unix Time Sharing. 21, 32, 35–38

VM Virtual Machine. 1, 5, 27–29, 87, 88

1 Introduction

For years software, its dependencies and configurations have often been deployed in monolith environments requiring manual interaction. These environments usually consist of server mainframes with high computing power that serve multiple applications at the same time. The act of creating scalable server landscapes in this scenario is as simple as adding more computing hardware to the existing mainframe, respectively adding more self-contained mainframes. Updates processes also demand manual actions or custom scripts in order to deploy pre-compiled software packages on a mainframe. These steps are being performed directly on the mainframe systems. Over time, with the steady increase of server load and the resulting requirements of deployment automation and optimization of resource usage, virtual machine environments came to use [Iva17].

By using Virtual Machine (VM) environments the goal to *isolate* applications, ensuring their availability additionally to constraining and optimizing resource usage can be fulfilled. In combination with automatic deployment processes, highly available services may be offered more easily. By adding more virtualized appliances in an automated way, scalability is being provided with much less effort compared to going through this process manually.

However, the VM approach also has its drawbacks because a large overhead has to be taken into account. This usually results in more disk space being consumed, less performance, longer boot times and additional complexity being required to run VMs compared to another approach of solving this kind of problem: *Containerization*. Virtual machines and containers are considered to be the foundation of today's cloud computing [cpf].

With the implementation of novel mechanisms in the Linux kernel, the efficient usage of container environments becomes possible, bringing them a vastly growing adoption [Iva17]. The comparatively lightweight containerization approach can be more efficient than the virtualization process in many ways, including one of the most important factors: Cost [tns]. On top of that, there are additional benefits (see 2.1.2), for example packaging applications along with all required components such as dependencies and configuration files. Also, the suitability of containerization for Continuous Integration/Continuous Delivery (CI/CD) processes enables developer teams to deliver products and features faster in order to gain a competitive advantage [tns]. The development process can generally be improved and optimized too: Developers are able to work more agile, for example rolling back configurations or applications to older version in case of failures is accomplished in a matter of seconds.

As of today, the media services provider *Netflix* uses container environments on a large scale [nmd]. Originally circa one hundred engineers developed a monolithic application in order to process application requests [nms]. However, this traditional development process did not seem to satisfy the ever growing number of customers and requests. *Netflix* solved this problem with a microservice approach: A large number of loosely coupled [wam] software containers are serving the application together. In 2017, the company launched over one million containers in a week. Using a custom container management platform [tts], container deployments can take place within one to two minutes. With virtual machines this would take tens of minutes [nmd] which illustrates the extent of performance advantage when comparing virtual machines and containers.

1.1 Problem Statement and Scope

With the steadily growing spread of containerization now and in the future, it becomes increasingly necessary to properly understand the internals and potential security threats resulting from aspects like kernel vulnerabilities, container misconfigurations and wrong use. This also includes optimizing the process of deploying and distributing containers and their environments to increase the productivity and efficiency, which directly impacts the cost factor.

To accomplish this, the Linux kernel primitives responsible for containerization are being taken into account. By using information on these mechanisms in combination with details to the corresponding kernel code, the functionality in regard to these mechanisms is being documented and illustrated. This explains how the operation of containers is possible and describes the internal processes of creating them. Additionally, potential pitfalls can be identified and prevented in the future. As the kernel source code is constantly subject to changes, all provided information is to be understood in regard to Linux version `4.19-rc3` (commit `11da3a7f84f1`).

Various types of containers may use different subsets of the available kernel primitives. By comparing diverse container types, respectively container engines, these differences can be uncovered and documented. The resulting security implications may determine the use-cases for a specific container type.

To understand how container security can be improved even more, unprivileged containers are also being described. This is a rather new development for certain container engines. This thesis analyzes the internals of this aspect in detail and also discusses its security implications.

Furthermore, it's important to determine how the available primitives are being used in practice. This will be taken into account for the security options of *systemd-nspawn* containers. In a Linux distribution-wide survey, all available *systemd-nspawn* configurations are being analyzed to build statistics.

Because of the large number and various types of containers and container engines available, not every type and engine can be taken into account when comparing the usage, security aspects and internals. Also, only Linux-based containerization is considered in this thesis. This also results in compatibility layers like *Docker* for Windows not being taken into account.

1.2 Desired Results

After reading this thesis, the reader should be enabled to properly understand the internals and underlying primitives of containerization. This, among other things, includes substantiated and detailed knowledge of the key mechanisms called control groups and namespaces. With this knowledge, efficient, optimized and secure container environments can be developed and distributed. By analyzing and discussing kernel source code the reader is able to gain insight on aspects of operating system kernel development and how to analyze such code structures for a specific goal.

By differentiating various container types and engines it's possible to choose a fitting containerization solution for a given use-case to maximize the advantages. This also includes distinguishing hypervisors and container primitives and gaining knowledge of the possible advantages and disadvantages.

Additionally, this thesis tries to illustrate ranges of topics that can be improved in the future or will begin to be increasingly relevant in the future of containers. This, among other things, includes the adoption of isolation primitives when developing *systemd* service units and the topics around unprivileged containers.

1.3 Thesis Structure

The second chapter provides background information regarding the building blocks of containers. It distinguishes between containerization and related topics and introduces a number of popular container engines. This makes it possible to differentiate the approaches behind these engines with the goal to associate selected container engines to the information found in the subsequent chapters.

Among others, namespaces and control groups are being discussed in chapter 3. Every covered kernel primitive is described along with implementation details and the ideas behind it. Another topic is the combination of kernel mechanisms in order to build a safeguarded environment that may still protect from certain attacks in case one security level fails to do so.

After that, unprivileged containers are being considered. The process of creating an unprivileged container is being dissected and illustrated. This chapter also contains information on open questions and topics that are still being worked on.

Statistics and diagrams have been created for chapter 5: The survey on *systemd* unit files for services.

This thesis concludes with remarks regarding the challenges, a conclusion and work that's planned or about to be released in the future.

2 Background

Before discussing the differences between *virtualization* and *containerization*, it's first important to distinguish and define [szd] two terms commonly used in conjunction with containers in this chapter:

- **Security mechanism**: To mitigate the possible impact of security violations it may be important to restrict the context a potential attacker is able to operate in. That is, for example, denying the execution of operations like arbitrary memory access, device access or loading kernel modules. This effectively provides processes or potential attackers with a limited subset of all operations available to reduce the exposed attack surface.
- **Isolation mechanism**: Preventing processes from interfering with other processes can be accomplished by defining boundaries a specific application can operate in. Possible ways of interfering includes tracing and intercepting calls of applications. An important aspect of isolation mechanisms is handling the access to global resources, such as network interfaces or system identifiers. It's possible to provide each application with its own view of a system resource.

2.1 Containerization Versus Virtualization

There exist multiple ways to package, isolate and constrain applications. While this thesis focuses on containerization, it's also important to address the aspects of virtualization as this approach is widely used in the industry and a powerful approach to solve similar problems. Furthermore this helps to prevent confusing these aspects – hence in this section the differences between these two technologies will be discussed.

2.1.1 Hypervisor-based Virtualization

The goal of this approach is to isolate an entire operating system [scv]. To accomplish this, a *hypervisor* is being used to simulate the entire hardware and environment of the virtualized operating system. Because of this, *virtual machines* enable users to run multiple operating systems of various architectures on a single machine. Virtualized operating systems are called *guests*, whereas the machine executing the hypervisor is called the *host*. The following schematic illustrates the described scenario:

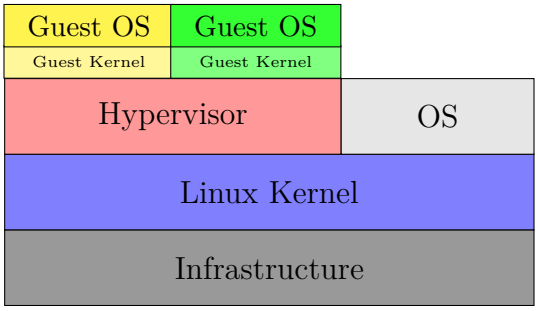


Figure 1: Hypervisor: Schematic Diagram (Hosted)

The hypervisor serves as an abstraction layer between the host kernel and the virtualized operating system. This, for instance, makes it possible to virtualize and share devices, files and network interfaces with the guest operating system.

There exist different types of hypervisors, namely type 1 and type 2 [hhv]:

- **Native / type 1** hypervisors run on bare-metal, without the requirement of an additional host operation system. On the schematic above this would remove the requirement of the Linux kernel. Examples of this kind of hypervisors are *Xen* [xen], which is available since the late 1990 years [xhs], and *Hyper-V* [hyp].
- **Hosted / type 2** hypervisors require an operating system in order to host virtual machines. *VirtualBox* [vbx] is an open source hypervisor of this type.

Among being helpful to build and operate application server mainframes, hypervisor-based virtual machines are often being used to perform the following tasks:

- Malware analysis and examination of foreign applications by making use of the isolation from the host operation system. This also includes the usage of network isolation. Please note that there's *no* guarantee that breakouts are being successfully prevented [vbc].
- Simplifying the development and deployment process by providing a common runtime environment for all involved machines.
- Provide reliability: Prevent system crashes by isolating multiple instances of the same server application. Crashes in a VM are unlikely to cause the whole virtualization cluster to crash.

Hypervisor-based virtualization relies on a certain level of privileged permissions, allowing guest operating systems to execute privileged operations through the hypervisor. This can include accessing protected memory segments and Input/Output (I/O) ports. In most of today's processors, there's a protection model implemented with privilege levels that are called *kernel rings* (adapted from [krp]):

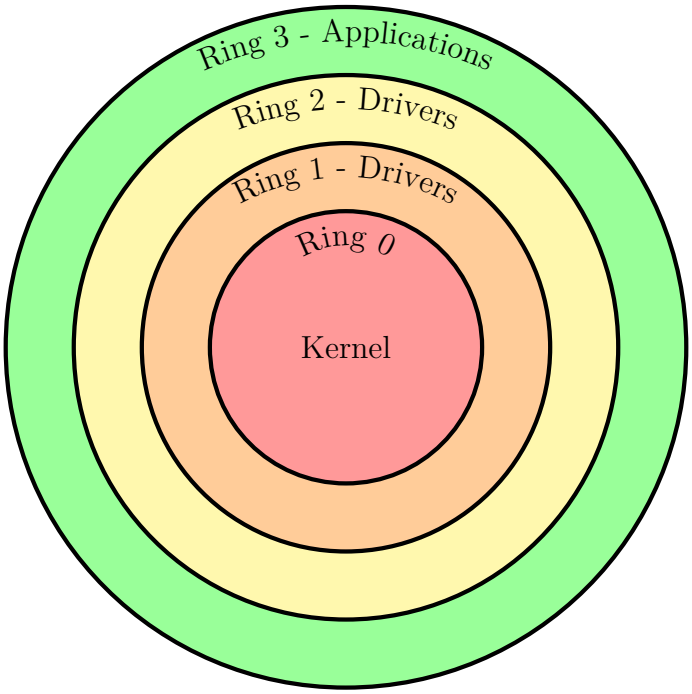


Figure 2: Hypervisor: Processor Ring Model

The general idea is that code runs with a specific *privilege level* and resources have a *protection level*. Access to a resource is granted in case the privilege level of the context an application is running in is equal or greater than the protection level of the resource.

The kernel of an operating system runs in kernel mode which is also called ring 0 and has the highest privileges. Depending on the required privileges, device drivers are able to run on ring 1 or ring 2. All user applications run in the least privileged ring. The rings 1 to 3 are also called *userland*. The Linux kernel allocates ring 3 for userland and leaves the rings 1 and 2 unused.

In order to perform privileged actions from userland, applications are required to interact with the operating system Application Programming Interface (API) using system calls (*syscalls*). This delegates application calls from userland to kernel mode in a pre-defined manner that also includes permission checks and error handling. It's important to note that this is in fact a transition from userland to kernel space.

Hypervisors run in the context of kernel ring 0 – either by being natively executed on the host hardware or by loading a custom module into the host kernel. Therefore, ring 0 is not available for guest operating systems as it's already occupied by the hypervisor. This requires the hypervisor to provide a solution for this problem – for example by using one of the following approaches [hvv]:

- **Paravirtualization**: By modifying the guest operating system, privileged operations of the guest that would run on ring 0 are being executed by the hypervisor instead. This may cause problems with proprietary operating systems and unknowingly modified guests.

- **Full virtualization**: This approach does not require modified guests. The hypervisor virtualizes a Central Processing Unit (CPU) for the guest operating system with the goal to provide a better compatibility level. In turn, reduced performance results from this method.
- **Hardware virtualization**: In short, this method adds another ring above ring 0 for the hypervisor to use. Certain CPU features, namely *AMD-V* and *Intel VT-x*, allow hypervisors to access this ring. Most of the time these features are disabled by default to prevent software from leveraging malicious actions with permissions levels above the level of the host operating system.

While providing a certain level of security, hypervisors can not *always* prevent breakouts [xvl]. Previously published vulnerabilities suggest that breakouts can compromise virtual machine hosts and other virtualized guests from the context of a guest operating system.

By design, using virtual machines has a number of drawbacks depending on the area of operations:

- **Required storage space**: By storing a whole operating system for each virtual machine, a high storage capacity has to be available depending on the number of virtual machine images.
- **Additional overhead and complexity**: The kernel of the host operating system is not shared with the guests. Therefore the execution of an operating system kernel per running virtual machine is required. Hypervisor internals can be complex and obscure in detail, adding additional complexity to environments requiring virtualization. While it may be a suitable approach for a large number of use-cases, hypervisor based virtualization can suffer from lower performance compared to other virtualization techniques.

2.1.2 Containerization

This section discusses the basics of containerization, which, in contrast to the hypervisor-based approach, isolates a set of processes [scv] rather than a full-fledged operating system including its kernel:

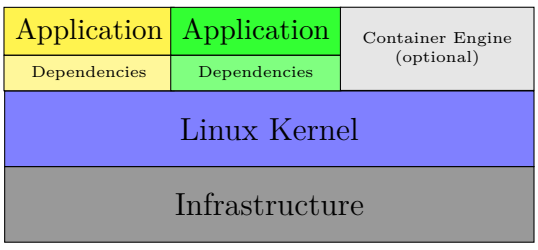


Figure 3: Containerization: Schematic Diagram

The *container engine* (see 2.7) as seen above is responsible for managing the lifecycle of all containers. An example of such an engine is *Docker* [dkr]. In today's IT landscapes these

engines are used frequently – however containers can also be created and managed manually by directly interacting with the host operating system and using the corresponding kernel primitives.

To a *certain* extent containerization is also a way of virtualization, namely operating system based virtualization [osc]. However, in this thesis there's a distinction between virtualization and containerization in order to illustrate the differences.

A container is a set of processes that's isolated from the host environment, processes, file hierarchy and network stack. Often containers are being created using *images*. These are minimal root filesystems that include the required binaries, dependencies and configuration files for the container to run. There's no additional abstraction layer between the kernel and the application and there are no devices being virtualized. Instead, the kernel of the host system is being *shared* with the isolated processes. Isolation is being implemented using primitives available in the Linux kernel.

Because of these aspects containers are very efficient:

- **Storage**: To build container images, often minimal base images of Linux distributions are being used. For example, an Ubuntu base image is 188 megabytes in size [isd]. That's only a small fraction of the size of a Ubuntu virtual machine. Moreover, there are even smaller base images like *Alpine* Linux [alp] with five megabytes and *Busybox* with only two megabytes in size. On top of that, base images can be reused for multiple images. If for instance multiple images are making use of the same base image, the base image only has to be stored once if the container engines supports layered images.
- **Performance**: Because of the shared kernel, there's minimal additional cost when running containers compared to the execution without any containerization. Containers are able to be added and removed in seconds, making them a handy tool for agile application deployment.
- **Complexity**: Every primitive that's being used to isolate processes is already included in the operating system kernel. Without the requirement of an additional hypervisor layer, the operating system is aware of all parts of the containerization process and can handle the execution natively. For example, memory management for containers can be as simple as managing the memory of a single native process.

It's important to address that using containerization it's *not* possible to host guests with another CPU architecture or operating system in case the provided host kernel is not supported by the guest.

Similar to the situation with hypervisor-based virtualization (2.1.1), there's always a certain possibility for container breakouts. In case of kernel vulnerabilities or misconfigurations, it may be possible for an attacker to execute commands or access devices and files in the context of the host operating system [dvl]. While containers do not prevent that services get compromised, they can help to limit the risk imposed by vulnerable services.

Using the containerization approach, applications can be shipped and deployed along with the required environment information, dependencies, network isolation and configuration

files in an efficient manner. This makes developing, sharing and deploying large scale applications in an automated way easier and saves cost.

Containers, as described throughout this thesis, are created by *combining* multiple kernel primitives into an isolation construct. This means that the kernel does not implement containers directly. They are rather being implemented in layers that use existing operating system features, while relying on the kernel to properly handle all security and isolation aspects. The term *container* does not have a strict definition. Therefore it's also possible to implement containerization by using only a subset of the kernel features described in this thesis or using an entirely different approach.

2.2 Performance Comparison

To illustrate possible performance losses when using virtualization or containerization, this section details results from various experiments [cpf]. The test objects are:

- **Linux-VServer**: This technology provides containerization using its own kernel patches [cpf] in order to perform the isolation process. Namespaces and *cgroups* are not being used.
- **OpenVZ**: Using the PID and network namespaces [cpf], *OpenVZ* isolates processes from each other. Every container has its own shared memory segments. This technology brings its own resource management compounds and CPU scheduler. The Completely Fair Queuing (CFQ) scheduler is being used to manage resources related to I/O.
- **LXC**: This utilizes Linux kernel primitives, namely namespaces and *cgroups*. Similar to *Docker* it uses the PID, networking and Inter Process Communication (IPC) namespaces and manages and confines available resources using *cgroups*. Similar to *OpenVZ*, the CFQ scheduler finds its use.
- **Xen Hypervisor** as previously explained in 2.1.1

The tests were performed on four identical Ubuntu machines running the Linux kernel with version `2.6.32-28`. At the time the experiments took place, running an older kernel version was necessary because of the requirement of a *single* kernel version supporting all four virtualization/containerization technologies at the same time. These are the test results:

- **Computing performance**: This benchmark shows the computing performance of a single CPU core. By solving linear equations with the least squares method [lpk], the *Linpack* benchmark determines the maximum peak performance of a system with CPU-intensive operations. The following diagram shows the results specific to this benchmark:

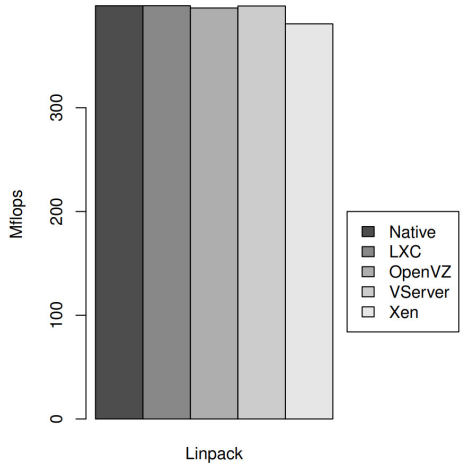


Figure 4: Benchmark: Linpack (CPU) [cpf]

There's almost no statistical significant difference between the performances when running Linpack natively or in a container. The average overhead when using the *Xen* hypervisor was 4.3% [cpf].

- **Memory performance**:

  The metric calculated in this benchmark is the maximum memory throughput. To create this benchmark, the tool *STREAM* [str] is used. It executes four different vector operations, namely Copy, Add, Scale and Triad and data structures larger than the usual memory cache size in order to produce as many cache misses as possible. This results in memory accesses that will be benchmarked:

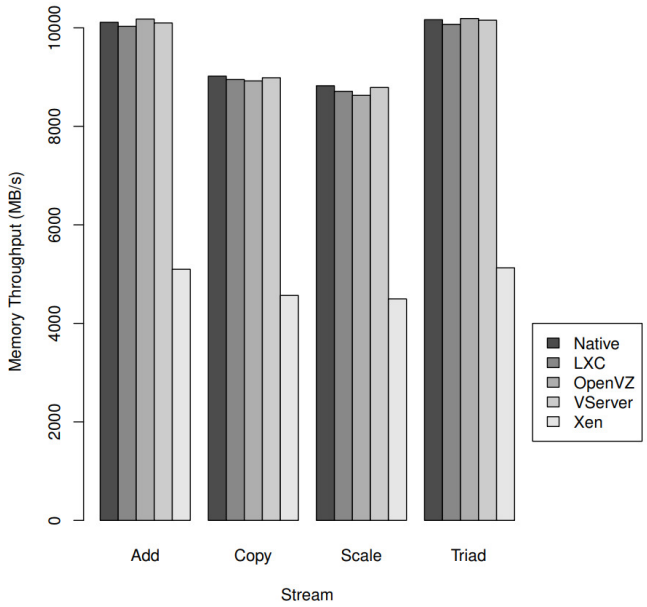


Figure 5: Benchmark: STREAM (Memory) [cpf]

This metric shows a significantly lower memory throughput with the *Xen* hypervisor in place: An overhead of approximately 31% has to be taken into account. This most

likely results from the hypervisor having to perform a memory translation from the guest memory space to the appropriate host memory space when accessing values in Random Access Memory (RAM). It's important that, similar to the situation when measuring the computing performance, almost no statistical difference is noticeable when comparing native execution and containerization.

- **Disk performance**:

  The IOZone benchmark [ioz] measures the I/O performance by making use of multiple access operations and patterns in combination with small and large files. For the tests performed in the experiment, 10GB and 4KB files were utilized [cpf]. The statistics are as follows:

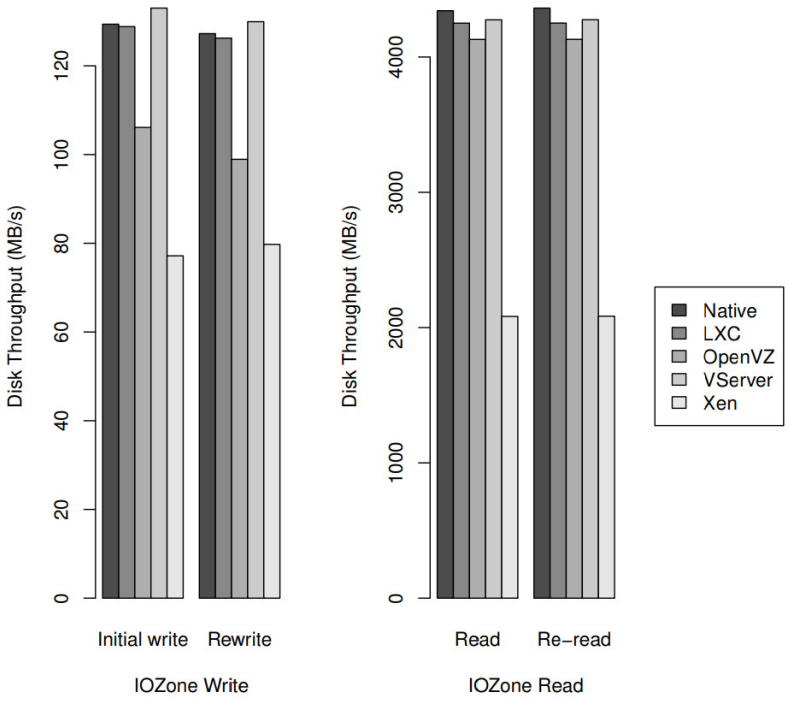


Figure 6: Benchmark: IOZone (I/O) [cpf]

Linux Containers (LXC) can gain almost native performance when reading and writing. Similar to the previous experiments, *Xen* also has the lowest performance in this benchmark. *OpenVZ* also has to suffer from an increased performance drop when writing files. In contrast, Linux-VServer can even show a higher performance than the native execution. This may result from LXC and Linux-VServer using the *deadline* scheduler where *OpenVZ* instead uses the CFQ I/O scheduler [cpf].

- **Networking performance**:

  By making use of the NetPIPE [npp] benchmark, network performance is determined in various conditions and with increasing message size. Also, network disturbances are being simulated. The resulting diagrams are shown in the next figure:

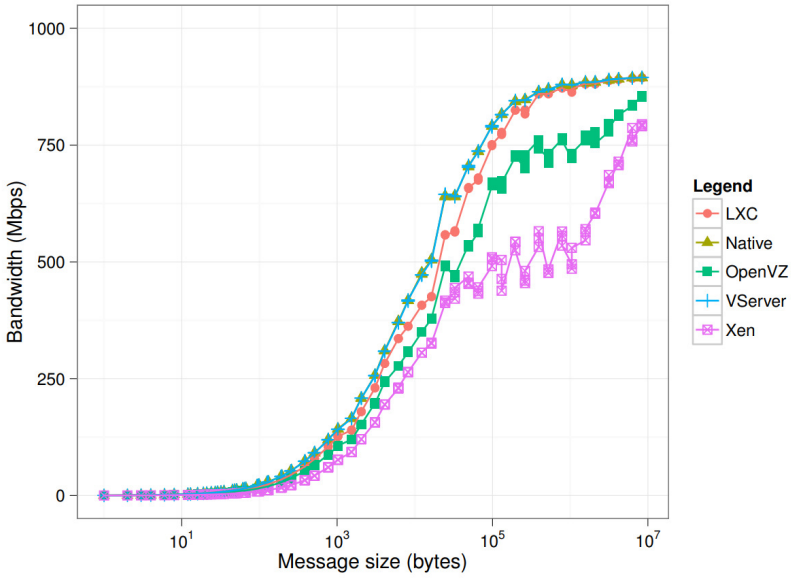


Figure 7: Benchmark: Network Bandwidth Usage (NetPIPE) [cpf]

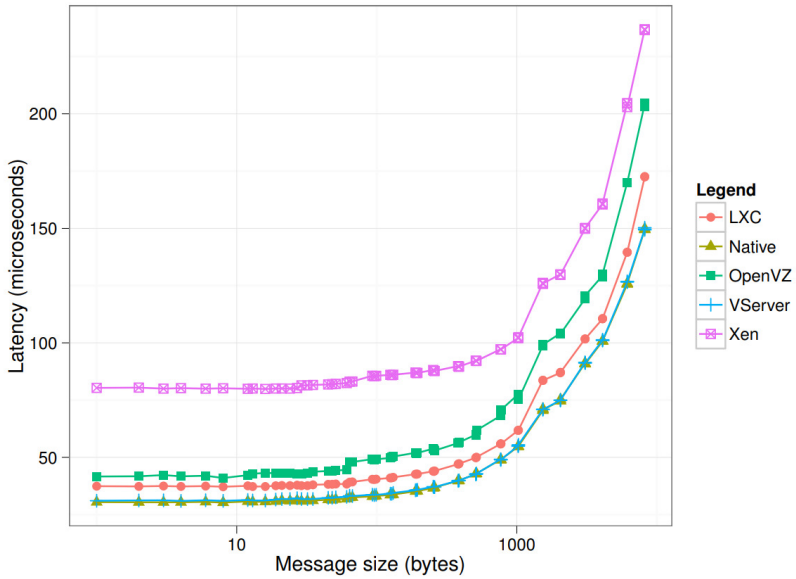


Figure 8: Benchmark: Network Latency (NetPIPE) [cpf]

OpenVZ, Linux-VServer and LXC all achieved similar results to the native execution. *Xen* reached a 41% lower bandwidth [cpf] than the native execution while providing a much higher latency time because of the additional complexity of the hypervisor layer.

All in all, the performance of guests virtualized by the *Xen* hypervisor showed less performance compared to the native execution because of the additionally required mappings and computations by the hypervisor. In contrast, containerization showed to cause little

to no additional overhead, effectively providing various isolation mechanisms with almost no performance impact. This applies to LXC in particular using all of the kernel primitives that will be examined in-depth in the following chapters.

2.3 Capabilities

The traditional way of handling permissions in Linux involves exactly two process types: *Privileged* and *unprivileged* processes. When disregarding capabilities, a process is privileged in case the effective User Identifier (UID) is equal to `0` – providing permissions commonly referred to as *root privileges*. The Linux kernel makes a sharp distinction between privileged and unprivileged processes. For example, privileged processes are allowed to bypass various kernel permission checks [cmp]. This results in a severe security violation in case untrusted applications are allowed to run with `root` privileges.

In practice, many system services and applications have to be executed with `root` privileges. Otherwise the permissions for privileged actions may be missing for a given process. For instance, consider sending Internet Control Message Protocol (ICMP) network packets with the `ping` command: Before capabilities were introduced in the Linux kernel, it was required to execute applications that use raw sockets with elevated privileges – for example using the Set User ID (SUID) bit. This theoretically enables the `ping` process that's being executed as `root` to perform privileged system changes or actions that may or may not be desired – additionally to performing the task the process is designated for. Tracing other processes, mounting devices and loading arbitrary kernel modules are some of the actions that will be allowed, additionally to using raw sockets. Therefore, the resulting attack scenario involves executing untrusted binaries or running potentially vulnerable system services with `root` privileges that may be used by a malicious actor to harm a system.

Because of the security concerns resulting from this, *capabilities* have been introduced starting from kernel version `2.2`. The basic idea is to split the `root` permission into small pieces that are able to be distributed individually on a thread basis [tts] without having to grant all permissions to a specific process at once. A complete list of all available capabilities is present in the capability manual page [cmp]. In particular, a powerful capability is `CAP_SYS_ADMIN` which allows to perform comprehensive actions on a system. Please note that this capability is overloaded and its use is discouraged [cmp] in case better alternatives are available, for example using only a minimal set of capabilities. However, sometimes using this capability is required – for example `mount` operations require this specific capability. Considering the example with `ping` would involve adding the `CAP_NET_RAW` capability to the process making use of raw sockets without having to grant a full set of `root` privileges.

There are two ways a process can obtain a set of capabilities:

- **Inherited capabilities**: A process can inherit a subset of the parent's capability set [cmp]. To inspect the available capabilities for a process, the `/proc/<PID>/status` file can be examined.
- **File capabilities**: It's possible to assign capabilities to a binary, e.g. using `setcap`. The process created when executing a binary of this type is then allowed to use the specified capabilities on runtime. This requires that the process is *capability-aware*, meaning that it explicitly requests the kernel to allow it to use these capabilities. In case a binary was developed before capabilities were introduced in the kernel, this is not the case as the required system calls for this task were not available at the time the binary was compiled. Hence, the process can fail to perform its designated task because of missing permissions. The Linux kernel manual calls these binaries *capability-dumb binaries*. A solution for this kind of problem will be discussed later on. Querying a binary for file capabilities is accomplished with the `getcap` utility. Performing this on *Arch* Linux and the included `ping` binary shows that the `CAP_NET_RAW` file capability is present, allowing pings without granting `root` permissions to the process.

2.3.1 The `execve` System Call

In the Linux kernel, the two main operations regarding processes are `fork` (see 3.1.2) and `exec`. The `fork` system call creates a new process as a copy of the parent process. On the other hand, `exec` replaces the current process with the contents resulting from executing another program. Both mechanisms are often used in combination, for example when invoking a command in a terminal that's not a shell-builtin:

1. `fork` is called to create a child process.
2. The child invokes `exec` to replace its contents with the desired program.
3. The parent waits for the child to exit and retrieves the exit code.

Please note that `fork` is required in the scenario above because the original process is not ready to exit yet – it rather invokes an external command and continues its execution afterwards. Without calling `fork` the *current* process would be replaced with the desired program. This process exits immediately after its execution which is not desired in the context of a shell.

Discussing `exec` above does not refer to a single system call. It rather describes the usage of a system call of the `exec`-family. While there are multiple `exec`-calls, most of them are built by making use of `execve` [elm]. The differences in the function APIs manifest in the provided parameters for each call.

The `execve` call is particularly important regarding the various capability sets and thereto related rules which will be discussed further on.

2.3.2 Capability Sets

There exist five different capability sets for each process, each represented by a 64 bit value:

- **CapEff**: The *effective* capability set represents all capabilities the process is using at the moment. For file capabilities the effective set is in fact a single bit indicating whether the capabilities of the permitted set will be moved to the effective set upon running a binary. This makes it possible for binaries that are not capability-aware to make use of file capabilities without issuing special system calls.
- **CapPrm**: The *permitted* set includes all capabilities a process may use. These capabilities are allowed to be copied to the effective set and used after that.
- **CapInh**: Using the *inherited* set all capabilities that are allowed to be inherited from a parent process can be specified. This prevents a process from receiving any capabilities it does not need. This set is preserved across an `execve` and is usually set by a process *receiving* capabilities rather than by a process that's handing out capabilities to its children.
- **CapBnd**: With the *bounding* set it's possible to restrict the capabilities a process may ever receive. Only capabilities that are present in the bounding set will be allowed in the inheritable and permitted sets.
- **CapAmb**: The *ambient* capability set applies to all non-SUID binaries without file capabilities. It preserves capabilities when calling `execve`. However, not all capabilities in the ambient set may be preserved because they are being dropped in case they are not present in either the inheritable or permitted capability set. This set is preserved across `execve` calls.

The sets discussed above allow a fine grained control over how capabilities are being distributed and inherited across processes and binaries. For backwards compatibility all capabilities are being granted to processes running as `root` user, including SUID binaries.

2.3.3 Capability Rules

Child processes created using `fork` inherit the full set of the parent's capabilities. Moreover, there exist rules [cmp] to determine whether and how capabilities of a process are being inherited or modified upon calling `execve` when creating a child process:

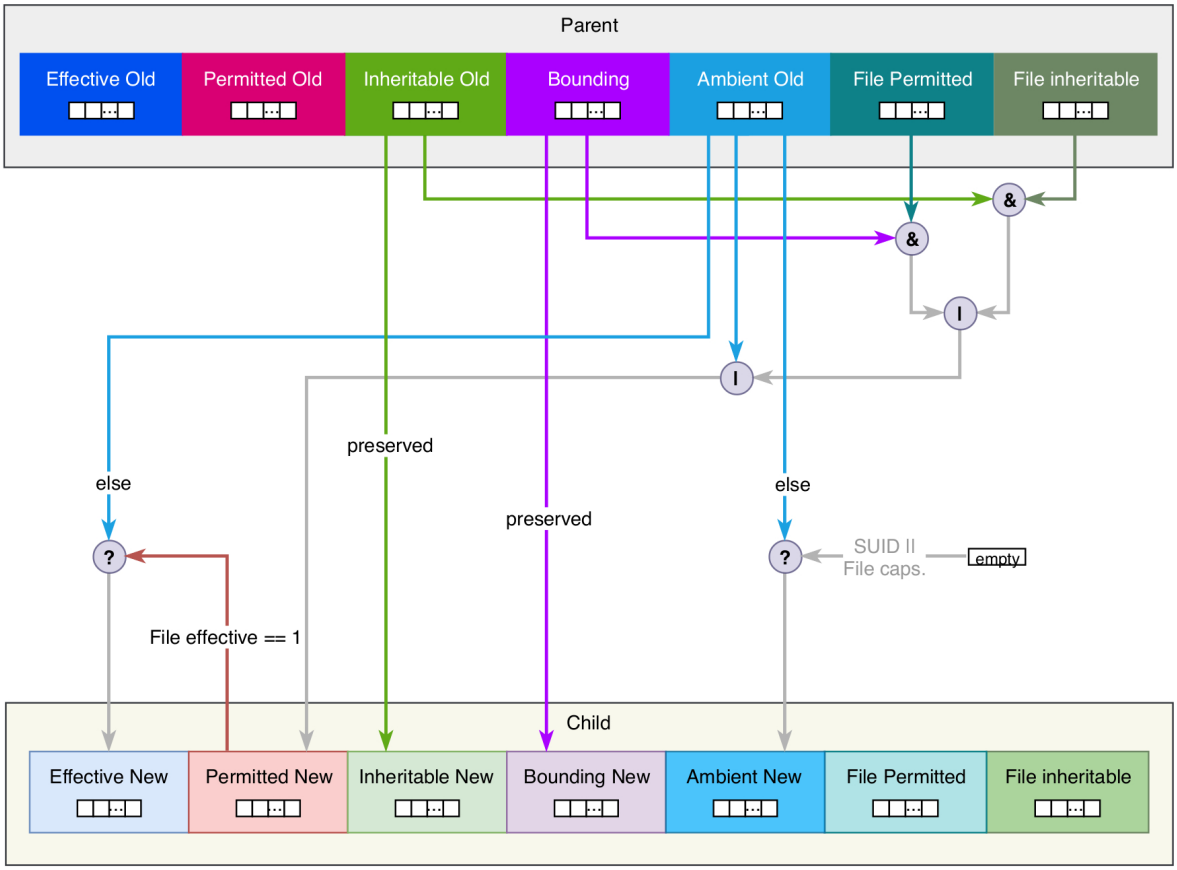


Figure 9: Capabilities: Preservation and Inheritance Rules for `execve`

In consequence of the rules above, UID changes have an effect on the available capabilities. If there's a transition from a zero to a non-zero effective UID value, the effective and permitted capability sets are being cleared [cmp].

There may be scenarios where a process requires a capability to perform initial configuration of the system and is then able to drop specific capabilities because their usage is not required anymore. In general it's a good software design to drop all unnecessary capabilities – container engines like *Docker* do this by default. To perform this task, the `prctl` utility can be of use. By utilizing this tool, the *secure bits* of a process are being manipulated. These flags control how capabilities are being handled for the `root` user. One flag, namely `SECBIT_KEEP_CAPS`, controls whether capabilities are being cleared in case a UID transition, for example with `setuid`, takes place. By setting this bit the current capability set is being retained when switching to another user from a process. For example, the implementation of `ntp` [ntd], the Network Time Protocol (NTP) daemon drops its capabilities as follows:

1. Set the `PR_SET_KEEPCAPS` secure bit using `prctl`. At this point `ntpd` is still being executed with `root` privileges.

2. Use a `setuid` call to switch to an unprivileged user. Because of the previous step all capabilities are retained across the UID change.

3. At this point the NTP daemon is running as an unprivileged user. However, all capabilities are still available to the `ntpd` process. With `cap_set_proc` of the `libcap` library only a selected subset of the currently available capabilities are being explicitly used while all others are being dropped automatically.

Preventing a certain capability from being present in a capability set of a given process can be achieved with the `PR_CAPBSET_DROP` flag. It effectively drops a capability from the bounding set and drops privileges prior to being added to one of the other capability sets.

While container runtimes can effectively make use of capabilities (see 3.1.11), the general adoption of this security mechanism that was once the hope for an innovation in privilege management [cng] was hindered [cng]. This results from capabilities being considered too complex and unhandy to use in the past [cng]. After all, specific capability sets and thereto related mechanisms, such as the ambient set that was added in Linux `4.3` (2015), were added long after the initial capability implementation. With the current capability implementation and API many of the original points of criticism can be disqualified.

2.4 Chroot, Jails and Zones

Apart from the previously mentioned mechanisms, there exist a few specific isolation methods that are being commonly used as alternatives to containerization on other platforms. Also, they inspired building containers in the past. It's worth noting that these methods do not use exactly the same primitives as virtual machines and containers. Three of these isolation methods will be discussed briefly in this section.

Chroot

The `chroot` system call [crs] was introduced in 1979 [crh] which allows a limited isolation of processes on the filesystem level. This system call changes the root directory of a process and its children in order to prevent it from accessing other folders above in the filesystem tree. Please note that all other resources are still being shared with a process after calling `chroot`.

In order to create a `chroot` environment, it's required to prepare a designated folder that will serve as new root directory. Depending on the desired use-case additional binaries or shared objects may have to be copied to the new root directory. Without copying shared objects to the new root directory it's not possible to run dynamically linked binaries because the system's shared objects are not accessible from a `chroot` environment. The following listing shows the commands required to setup a *very* basic `chroot` environment running `/bin/sh`:

```
root@box:~# mkdir chroot && cd chroot
root@box:~# mkdir bin etc lib var home
root@box:~# ln -s lib lib64
root@box:~# ldd /bin/sh # Check required shared objects
  linux-vdso.so.1 (0x00007fffa23f0000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f241825b000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f241886c000)
root@box:~# cp -L /bin/sh bin
root@box:~# cp -L /lib/x86_64-linux-gnu/libc.so.6 lib
root@box:~# cp -L /lib64/ld-linux-x86-64.so.2 lib
root@box:~# chroot . /bin/sh # Enter the chroot
```

Listing 1: Chroot: Creation of the Environment

While `chroot` provides a very basic process isolation, the resulting security level is limited as well. For example, in case code-execution is given, it's possible to escape the `chroot` environment by using `chroot` again and spawning a shell in the system's root directory [cre]. Currently the *Docker* project uses `chroot` to securely import images in a manner that prevents attacks using relative symbolic links embedded into manipulated image archives.

Jails

Starting from FreeBSD 4.X [fjl], *Jails* are available which are based on `chroot`. Jails expand the functionality provided by `chroot` with the goal to prevent escapes and provide a more mature isolation. For example, Jails usually possess own hostnames and network stacks.

The system calls `Jail` and `Jail_attach` are responsible for creating Jails and moving processes to specific Jails. By utilizing the corresponding application, these tasks can also be performed from the command line [ppd]. According to the Jail restriction documentation [ppe], there exist checks throughout the kernel code checking whether a process is jailed to eventually deny accessing certain resources or functionality. This means that there's a built-in support for Jails in the operating system. As previously described this is not the case for containers since these are not directly considered in the kernel code.

Using various settings, called `sysctl`s, the actions and resources Jailed processes are able to execute can be configured.

Zones

The *Solaris* operating system implements its own isolation mechanism called *Zones* [szd]. It internally uses `chroot` while applying mitigations on the kernel level to prevent breakouts by stacking `chroot` environments (see 2.4). When creating a Zone, the root directory and network stack, among other things, are isolated. The basic goal is to prevent Zones from interfering with each other and isolate resource accesses. Inter-Zone communication can take place using a shared network stack.

By default all processes reside in the global Zone where no isolation is being applied. All files, network devices and processes are visible and usable for processes running in

the global Zone. This is also useful for administration purposes. Otherwise administrative actions are only possible for the Zone an administrator is currently residing in. By distinguishing the global Zones from all others, there's the possibility of having global administrators and less privileged administrators for other parts of the system. On top of that, various system calls corresponding to administrative tasks are only allowed in the global Zone. This includes loading custom kernel modules, rebooting, accessing kernel memory and physical devices and configuring network interfaces.

The superuser privileges are split into separate privileges, similar to Linux capabilities (see 2.3), for a fine granular control over privileges assigned. For example, processes running inside a Zone may be allowed to mount devices but it may not be allowed to reboot the system. The system provides a sane default set of privileges each Zone can possess without the risk of compromising the security of other parts of the system.

Each Zone is identified using a name and a unique identifier. Persistent configurations can be created with the `Zonecfg` utility with the possibility of exporting configurations in XML format.

Similar to Jails, Zones are an operating system construct and are therefore directly implemented in the kernel.

2.5 Container Insight

After discussing what containers are, this section provides additional information on container images, performance aspects and the building blocks of containerization.

2.5.1 Images and Containers

When talking about containerization, there has to be a strict distinction between *containers* and *images*:

- **Images** consist of an immutable root filesystem that provides all necessary libraries, applications and dependencies in order to run a container. Images can consist of layers that are stacked upon another using a union filesystem during the image building process, but this does not have to be the case necessarily. Images can be created manually or by using an *image builder* that takes an image description file, for example a *Dockerfile*, as input and yields a ready-to-use container image.
- **Containers** are instances of images. For example, in the layer-based situation of *Docker* containers this would result in an additional layer above all images layers which is writable in order to allow changes on runtime:

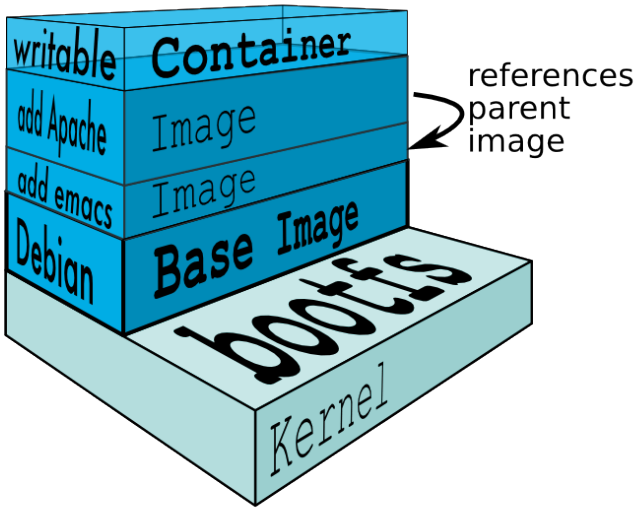


Figure 10: Docker: Image layers (Modified [dil])

All changes to files in layers below will only be present in the writable layer [dsd]. Depending on the used container engine it may be possible to convert a container into an image, creating a snapshot of a running container after applying certain changes.

2.5.2 Container Building Blocks

Before going into detail about the kernel primitives containers are based on in chapter 3, this section discusses the basic building blocks of containers. This is required in order to properly understand the following sections of this chapter and already briefly introduces aspects that will be explained in-depth in the following chapters.

Namespaces

This kernel primitive defines the scope of a process in regard to what it can *see*. This results in processes having different views of the same system. For example, this includes the root filesystem mappings: This can be set to another root file hierarchy that's in scope for a specific process, making other files and folders above the specified hierarchy out of scope for the process and therefore isolating them.

There are multiple namespaces available to isolate certain aspects of a system. A simple example is the Unix Time Sharing (UTS) namespace. It, among other things, isolates the hostname that's available to a container in order to allow it to set its own hostname. The Linux command line utility *unshare* that will be discussed in 3.1.3 allows to perform this kind of isolation:

```
1 root@box:~# hostname # Read the current hostname
2 box
3 root@box:~# unshare -u /bin/bash # Enter new UTS namespace
4 root@box:~# hostname anotherbox # Change the hostname
5 root@box:~# hostname # Confirm it has been changed
6 anotherbox
7 root@box:~# exit # Return to the parent shell
8 root@box:~# hostname # Check: hostname did not change system-wide
9 box
```

Listing 2: Container Building Blocks: Using Namespaces

This depends on elevated privileges because creating namespaces other than user namespaces (see section 3.1.11) requires the capability `CAP_SYS_ADMIN`.

As shown above, the hostname of the spawned `bash` process has been isolated from the system's hostname, effectively allowing the child process to change the hostname without changing it globally for the whole system. This rather simple example illustrates the basic functionality and idea of namespaces. An in-depth analysis of namespaces can be found in section 3.1.5.

Control Groups

To limit the resource usage of processes *Control Groups*, or *cgroups*, are being implemented in the Linux kernel. While *cgroups* will be discussed in section 3.2 in detail, this section provides a short example of the functionality and the usage of common user space applications to manage *cgroups*.

Control groups are accessible through a virtual filesystem. Using this filesystem they can also be modified using shell commands. However, using the command line applications can simplify this process.

To mount the virtual filesystem, the command `cgroups-mount` is being invoked. An overview of the mounted virtual control group files can be seen in the listing below:

```
$ cat /proc/mounts | grep cgroup
tmpfs /sys/fs/cgroup tmpfs [...]
cgroup /sys/fs/cgroup/net_cls,net_prio cgroup [...]
cgroup /sys/fs/cgroup/cpuset cgroup [...]
cgroup /sys/fs/cgroup/hugetlb cgroup [...]
cgroup /sys/fs/cgroup/pids cgroup [...]
cgroup /sys/fs/cgroup/perf_event cgroup [...]
cgroup /sys/fs/cgroup/cpu,cpuacct cgroup [...]
cgroup /sys/fs/cgroup/freezer cgroup [...]
cgroup /sys/fs/cgroup/devices cgroup [...]
[...]
```

Listing 3: Container Building Blocks: Mounted Control Groups

For the the following example, the *cgroup* `devices` will be used to restrict a process from accessing `/dev/null`.

To limit the usage of devices, their major and minor numbers have to be used. These numbers are the respective identifiers of a device in the filesystem tree. The major number describes the driver that's required and is used by the kernel in order to access a specific device. The minor number is used by the device driver to distinguish logical and physical devices resulting from the existence of a certain device. In the above example for `/dev/null` these numbers can be identified using `stat -c "major: %t minor: %T" /dev/null` which yields the values `1` for major and `3` for minor.

First, a device control group has to be created with `cgcreate -g devices:nodevnull` with the identifier of the control group being `nodevnull`. To add the current shell process to this group, the command `cgclassify -g devices:nodevnull $$` will be invoked. The process identifier of the current shell process is `$$`. To finally deny accessing the `/dev/null` device, this command will be executed: `cgset -r devices.deny="c 1:3 rwm" nodevnull`. The format [cgd] of the parameter for `devices.deny` is as follows:

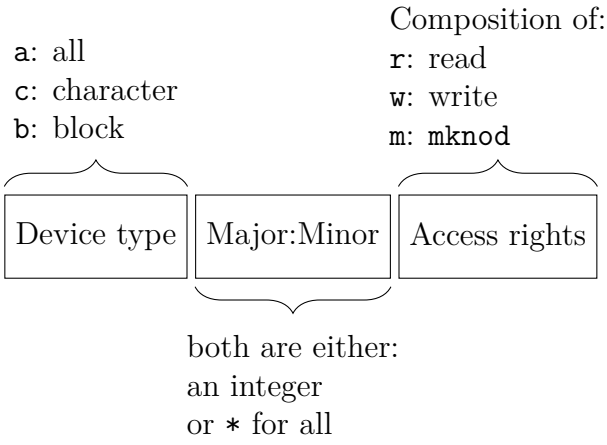


Figure 11: Container Building Blocks: `devices.deny` Parameters

The device type is determined by using the first character of the output of `ls -la /dev/null`, which shows that it's a character device.

Now accessing the specific device is being blocked, even for processes running as `root` user:

```
1 root@box:~# echo "a" > /dev/zero # Allowed
2 root@box:~# echo "a" > /dev/null # Denied
3 bash: /dev/null: Operation not permitted
```

Listing 4: Container Building Blocks: Deny Access to Devices Using Control Groups

The usage of the introduced command line tools may be convenient for many use-cases because there's a possibility to create snapshots of control group configurations. This allows restoring the created groups and rules since these rules are *not* persistent by default.

Namespaces and control groups are the basic building blocks of containers. In contrast to virtual machines, there's still the possibility to apply very fine granular configurations of all security and isolation mechanisms in place, providing the ability to develop extremely flexible setups. For example, it's possible that only a specific namespace set is in use for certain containers and sharing namespaces with multiple containers is possible to effectively link containers. Linking containers allows, among other things, putting various container in a shared network namespace in order to allow a communication between their otherwise isolated processes.

2.6 Copy-On-Write Filesystems

Upon starting a container from a specific image, a thin writable layer, sometimes called the *container layer*[dsd], is being created above all image layers. This makes container storage more efficient since the container layer is as small as possible in size.

When reading files in a running container, the requested file is being searched in all subjacent image layers starting from the newest layer. When performing write operations to a file the same search is being run in order to copy the file to the writable layer and apply changes afterwards. This results in all changes in reference to the image layers below being stored in the container layer. This operation is called *copy up* and is a part of the *copy-on-write* strategy. Any further write operations to a file which has already been copied up happen directly in the container layer. This leads to a maximum of one search operation per file in the tree of image layers. After a container performs a write operation, any other container originating from the same image is still able to use the original and unchanged file since there's no sharing of the writable layer taking place.

This approach effectively minimizes the required storage space per container and also increases the startup time of containers by avoiding unnecessary copy operations. Without the copy-on-write strategy container, the storage strategy would be similar to virtual machine storage with one virtual disk per container, making them far more inefficient.

2.7 Container Engines

To manage containers and images easily, container engines are commonly being used. Some included features include:

- Specifying image layers, for example with a *Dockerfile*, and building images
- Exporting and importing images
- Pulling images from public or private repositories
- Creating, managing and deleting containers
- Mounting files or folders in order to share data
- Linking containers to allow inter-container communication
- Applying custom namespace and control group configurations

It may be possible to perform the tasks mentioned above manually, for example as previously shown in 2.5.2 in the case of namespaces and control groups. However for maximum usability, large scale deployments and recurrent tasks a convenience layer is required. This is being provided by container engines that are often also called container runtimes. Additionally, container engines often provide secure defaults for various container security settings. This section discusses various container engines and illustrates major differences between them and the resulting containers.

2.7.1 Docker

The *Docker* project is an open source [dce] container platform which was first released in March 2013 [jfk]. The basic features are:

- Building images using *Dockerfiles*
- Importing and exporting images
- Managing the lifecycle of containers
- Providing sane security defaults, like the default *seccomp* profile (see 3.3)

Internally it uses the kernel primitives mentioned in 2.5.2, namely namespaces and control groups, to provide containerization. Additionally it's possible to use other containerization back-ends such as Jails or Zones by using another execution driver, uniting multiple containerization techniques in one workflow [ded].

It consists of a client-server application structured as follows:

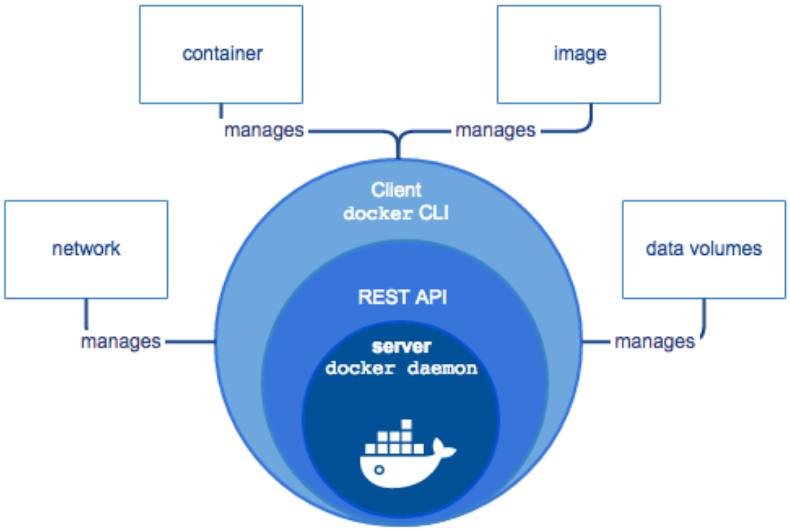


Figure 12: Docker: Engine Overview[deo]

The Command Line Interface (CLI) client is available with the `docker` command after installing the corresponding software package. It's written in the *Go* programming language and communicates with the privileged *Docker daemon* in order to manage containers. This results in the CLI application not managing containers on its own – it rather delegates this task to the daemon component. The communication takes place via a Representational State Transfer (REST) API.

Internally the daemon uses `runC` [rcg] and `containerd` [cdg] [drc]. In releases older than version `1.11` *Docker* used a library called `libcontainer` instead of `runC`. This change resulted from *Docker Inc.*, the company behind the *Docker* project, contributing `libcontainer` to the Open Containers Initiative (OCI) [oci]. The goal of this initiative is to create an open industry standard for container formats and container runtimes. This means that initially `runC` basically was a repackaging of `libcontainer` implementing the OCI runtime specification, serving as a reference implementation. A schematic overview of the interactions between `runC` and `containerd` is shown in the following figure:

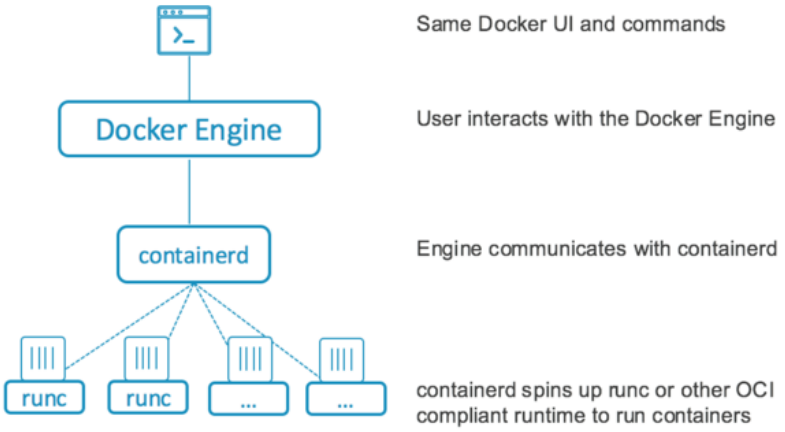


Figure 13: Docker: Usage of runC and containerd [dcr]

The purpose of each involved component is:

- `runC`: Spawns and runs containers according to the OCI specification. This CLI application can use JavaScript Object Notation (JSON) based configuration files and spawns containers from this configuration. By moving the code of `runC/libcontainer` out of the *Docker* code base it's possible to upgrade *Docker* without having to restart all running containers as this may cause issues for large-scale environments.
- `containerd`: Manages the container lifecycle, as in image storage, container execution, supervision and networking. It's a `runC` multiplexer and decouples the container execution from the runtime environment. The `containerd` utility can not create containers on its own and therefore delegates this task to `runC`.
- `container-shim`: This component effectively replaces `runC` in the figure above on runtime. After `runC` starts a container, it's useful to let the `runC` process exit. This allows to run containers without having any long-running runtime processes on a system responsible for container execution. The second reason for using a shim process is to keep `stdout` and all file descriptors of a container open in case `dockerd` or `containerd` crash, allowing the container to keep running. The third task of the shim is to report the exit code of a container. To perform the tasks mentioned above, the `PR_SET_CHILD_SUBREAPER` [srp] flag of the `prctl` system call is being used by the shim process to mark itself as the reaper of the container process after `runC` has exited [mcd]. Otherwise the parent of the container process would be `init` (or an equivalent to it) with process ID 1.
- `gRPC`: The communication between `containerd` and the *Docker* engine is being handled using the `gRPC` [grp] Remote Procedure Call (RPC) library.

Docker Inc. provides the *Docker Hub* [dhb] on which container images can be uploaded and shared among other *Docker* users. As of March 2018 there have been 37 billion image downloads and 3.5 million *dockerized* applications with a fast rising trend [dhs].

While many contributions to *Docker* originate from the project owners themselves, *Docker* profits from contributions of companies like *Red Hat*, *IBM*, *Google* and the open source community.

2.7.2 LXC

With its first release in 2008 [lxg] LXC has emerged as a containerization solution that has been available for several years. It uses the same kernel primitives like *Docker*, making their performance similar [ldr]. Additionally, *Docker* originally used LXC as back-end before `libcontainer` was available [ded]. However, their use-cases and benefits differ.

The goal of LXC is to provide operating system containers. In contrast, *Docker* provides application containers with the goal to containerize a single application and its dependencies. This results in a different focus and workflow when comparing LXC and *Docker*.

The data stored in LXC containers is state-based. By default the container data is available in the directory `/var/lib/lxc`. This means that this data will still be available after stopping or restarting a container. Additionally, LXC does not bring the same benefits to how application code and its dependencies are packaged and deployed [ldr]. For instance, data storage is not handled via mounts – it's handled by directly adding files to the container filesystem. In contrast *Dockerfiles* make building images transparent and mounts show exactly what data was manually supplied to a container. The Copy-on-Write mechanism is not available to LXC containers, making image store more inefficient when compared to the approach *Docker* uses. Nevertheless, in case whole operating systems have to be containerized, using LXC may be a more suitable solution.

LXC containers are created by preparing or downloading a root filesystem template and configuring the container using a settings file. With several API bindings for C, Python and various other programming languages containers of this type can be configured and managed programmatically.

2.7.3 Kata Containers

This project tries to combine the security of VMs in combination with the speed of containers, effectively creating lightweight virtual machines. The goal is to be compliant to the OCI specification in order to allow *Kata Containers* to be orchestrated by *Docker* and *Kubernetes*, hence it's covered in this section although *Kata Containers* may not be containers per definition.

The basic idea is to create a lightweight hypervisor to allow every container to have its separate kernel:

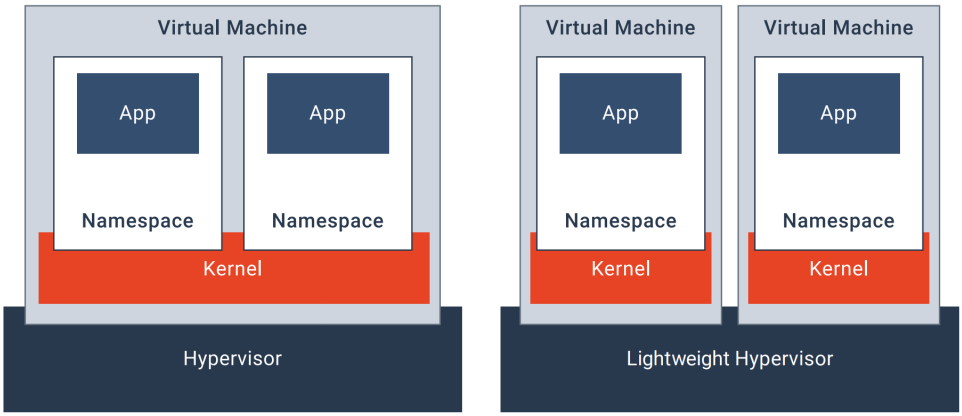


Figure 14: Kata Containers: Hypervisor-based Virtualization and the Kata Architecture [kcs]

While this approach may consume more computing resources than LXC and *Docker*, it can offer security benefits. As the host kernel is *not* being shared with the guest systems, the impact of kernel based container breakouts can be limited. *Kata Containers* consist of several components listed in the schematic below:

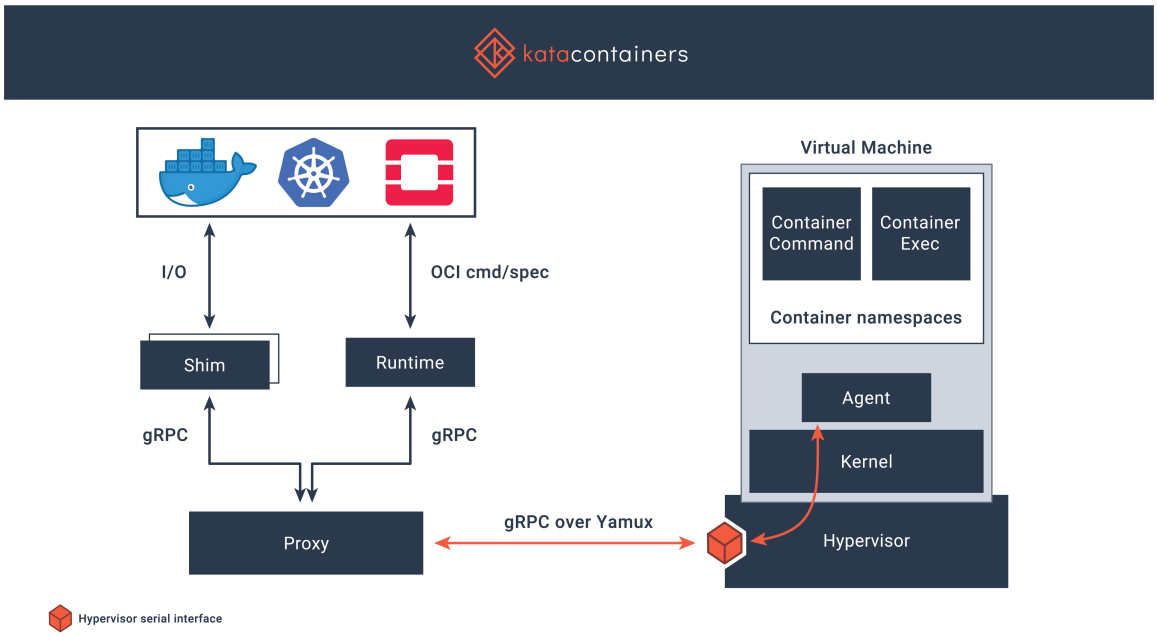


Figure 15: Kata Containers: Components [kcs]

- **Agent**: To follow the OCI specification, the agent is based on `libcontainer` in order to process requests and images based on the OCI specification. Agents run inside a minimal VM.
- **Proxy**: The (optional) proxy runs on the host system. Communications between agents and the host systems are being managed using virtualized serial ports. The

multiplexing of this communication is being carried out by the proxy. This can also be handled by using virtualized sockets, making the proxy optional.

- **Shim**: Similar to the shims used by *Docker* (see 2.7.1), *Kata Containers* use an additional shim process to translate signals sent from container management tools to commands inside of a container.

The project aims to support multiple architectures and hypervisors focussing on Kernel-Based Virtual Machine (KVM). For environments and use-cases having a virtualization-only policy this technique may allow the usage of principles, tools and workflows of the container world in combination with lightweight VMs. For example, managing lightweight VMs with the container orchestration approach *Kubernetes* is possible.

2.7.4 systemd-nspawn

The system and service manager *systemd* [sdd] implements its own containerization solution, namely *systemd-nspawn*. The term `nspawn` is short for *namespaces spawn*. This already suggests that this container type does not interact with *cgroups* on its own – it rather delegates this task to the `systemctl` application belonging to *systemd*. It's based on `chroot` [aws] and exists for the purpose of debugging, testing and building software. The lifecycle of containers of this type can be managed with the `machinectl` utility.

In modern Linux distributions, the default service manager is *systemd*. It manages all standard services and, as long as no additional configuration is being provided, executes them on the host system. However, these system services can also be executed in *systemd-nspawn* containers. The following listing [sdu] shows a *systemd* unit configuration:

```
1  [Unit]
2  Description=Some HTTP server
3  [...]
4  Requires=sqldb.service memcached.service
5  AssertPathExists=/srv/www
6
7  [Service]
8  Type=notify
9  ExecStart=/usr/sbin/some-fancy-httpd-server
10 [...]
11 PrivateTmp=yes
12 [...]
```

Listing 5: Systemd-nspawn: Unit Configuration Example

The example service defined in the configuration can hereupon be executed in a containerized environment. Directives like `PrivateTmp` allow a flexible configuration of several isolation aspects, like private temporary folders for a container. The usage of these directives will be discussed in chapter 5.

Images for this container type can be created manually or with tools like `debootstrap` which for instance prepares a filesystem for a *Debian* environment. Also, various image types of other container runtimes may be utilized.

2.7.5 gVisor

To mitigate the risk of container breakouts resulting from exploiting kernel vulnerabilities, the *gVisor* project [gvi] implements a user-space kernel for containers in the *Go* programming language. *Go* was chosen because of the built-in mitigations against common security issues like stack overflows and use-after-free vulnerabilities. By limiting the exposed attack surface for potential malicious actors, the risk of breakouts shall be reduced:

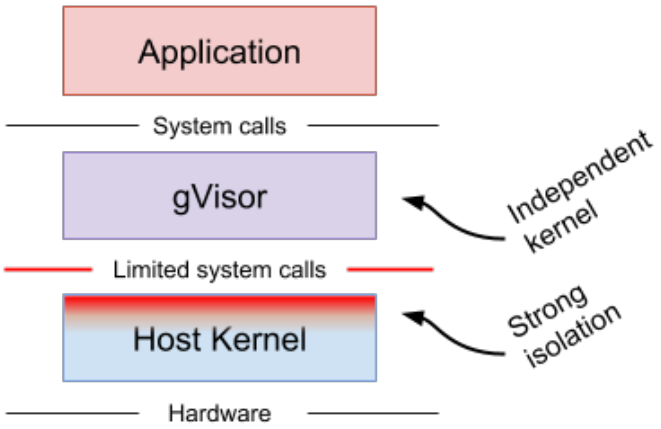


Figure 16: gVisor: Layers [gvl]

Through interception of system calls originating from containers, as illustrated as *Application* in the figure above, various mitigations can take place. For instance, system calls can be blocked based on rules and input can be sanitized before delegating calls to the *real* system kernel. However, *gVisor* does not aim to delegate all calls to the kernel layer. It rather implements the Linux primitives like signals, filesystems and memory management on its own. All these mechanisms lead to the fact that containers and their processes are not able to directly interact with the host's kernel. Of course, this design comes with a certain degree of overhead for each system call.

gVisor is compatible to the OCI specification and can therefore be used in conjunction with *Docker*. It's actively maintained at the time of writing this thesis and was published by *Google*. At the current state of the project not all applications may be containerized by *gVisor*. For example the database server *PostgreSQL* requires a system call, namely `sync_file_range`, that has not been implemented yet [gvp]. In the past vulnerabilities have been found in *gVisor* that allow processes to break out of the container environment [pzg] which eventually have been fixed.

2.7.6 rkt

This container engine works without a daemon and is able to run and manage OCI and `rkt`s own *appc* images. The term *appc* stands for *App Container* and describes a container image specification [apc]. This container engine was the first to have its own published image specification [rkc].

Containers spawned by `rkt` are application containers similar to *Docker* containers (see 2.7.1). By default the container data is saved on the host system, making the data storage persistent and independent of the container state. The `rkt` engine also brings its own default capability set [rka]. This container type can be created with the corresponding `rkt` application. The creation process is divided into three stages [rkd]:

1. **Stage 0**: Execution of the `rkt` binary which among other things prepares the filesystem for the two remaining stages
2. **Stage 1**: Creates the isolation according to the desired settings. There are three main flavors available:
   - `fly`: Creates a `chroot` environment
   - `systemd-nspawn`: Containerization using control groups and namespaces
   - `kvm`: Leverages full virtualization using the KVM approach
3. **Stage 2**: Execution of the containerized application process

The `rkt` container engine was first published by *CoreOS Inc.* which was then acquired by *Red Hat* [rko].

3 Kernel Primitives

This chapter focusses on kernel primitives that were discussed briefly in previous chapters. To properly understand the building blocks of containers it's necessary to provide an in-depth view of all involved mechanisms and their specific roles in containerization. By analyzing and explaining excerpts of the kernel source code this information is being substantiated. It also includes the description of short code snippets useful for demonstrating the presented primitives.

3.1 Namespaces

Being introduced first in Linux kernel version `2.4.19` (2002) [mgn], namespaces define groups of processes that share a common view regarding specific system resources. This ultimately *isolates* the view on a system resource a group of processes may have, meaning that a process can for instance have its own hostname while the *real* hostname of the system may have an entirely different value – as seen in the example 2.5.2: Every process in a *single* given UTS namespace shares the hostname with every other process in the namespace. There exist various types of namespaces, with each type isolating other aspects of the system. Namespaces can also be used in combination by making a process a member of multiple new namespaces at once.

Currently the following namespace types are available:

Table 1: Namespaces: Types

Flag	Purpose
`uts`	UTS namespace as previously described in 2.5.2
`mnt`	Mount namespace
`pid`	Process Identifier (PID) namespace to isolate the visible process ID number space
`net`	Network namespace, isolating the network stack
`ipc`	IPC namespace to isolate inter process communication interfaces
`cgroup`	Control group namespace
`user`	New user namespace (4)

It's important to note that by default every process is a member of a namespace of each type listed above. These namespaces are called *default*, *init* or *root* namespaces. In case no additional namespace configuration is in place, processes and all their direct children will reside in this exact namespace. This can be verified by executing `lsns` to list all namespaces in two different terminals and comparing the namespace identifiers which will be equal.

The isolation provided by namespaces is highly configurable and flexible. For instance, it's possible for a database application and a web application to share the same network

namespace, allowing both processes to communicate while other processes that reside in other network namespaces are not able to do so.

For namespaces two new system calls have been implemented [Iva17], namely:

- `unshare`: This disables sharing a namespace with the parent process, effectively *unsharing* a namespace and its associated resources. Please note that this system call changes the shared namespaces in-place without the requirement of spawning a new process – with the PID namespace being an exception in this case as discussed in a further section.
- `setns`: Attaches a process to an already existing namespace.

Another system call that's involved in creating namespaces is `clone`. Before explaining the involved system calls in detail, it's first necessary to discuss how namespaces are exposed from userland.

3.1.1 Filesystem Structure

The directory `/proc/<PID>/ns` holds symbolic links representing the namespace membership of each process:

```
1 lrwxrwxrwx 1 user user 0 Sep 21 15:33 cgroup -> 'cgroup:[4026531835]'
2 lrwxrwxrwx 1 user user 0 Sep 21 15:33 ipc -> 'ipc:[4026531839]'
3 lrwxrwxrwx 1 user user 0 Sep 21 15:33 mnt -> 'mnt:[4026531840]'
4 lrwxrwxrwx 1 user user 0 Sep 21 15:33 net -> 'net:[4026532009]'
5 lrwxrwxrwx 1 user user 0 Sep 21 15:33 pid -> 'pid:[4026531836]'
6 lrwxrwxrwx 1 user user 0 Sep 21 15:33 user -> 'user:[4026531837]'
7 lrwxrwxrwx 1 user user 0 Sep 21 15:33 uts -> 'uts:[4026531838]'
8 [...]
```

Listing 6: Namespaces: Directory Contents

The purposes of these symbolic links are the following [lnf]:

- Determining whether processes are members in the same namespace by issuing the same *inode* number to the respective symbolic links. This can also be determined using `readlink` which reports an identifier of the namespace.
- By opening or bind-mounting one of the symbolic links the associated namespace will be prevented from being destroyed in case the last process that's a member of the namespace exits. This allows namespaces to be persistent.

3.1.2 `clone`

Similar to the `fork` system call, `clone` is used to create child processes. There are multiple differences between the two calls: The most significant difference is that `clone` can be highly parametrized using flags. For example, it allows sharing the execution context, for instance the process memory space, with a child process. Therefore `clone` can also create threads and is more versatile than the legacy `fork` call. The `fork` call does not support most of this behavior. Instead, `fork` is essentially being used to create child processes as copies of a parent process. Before going into more detail about the differences and similarities it's first important to understand what's happening when `fork`, `clone` or a system call in general is being invoked in a C program.

By using one of these two calls in C code the actual code that will be executed is *not* the system call itself as defined in the system call table [sct]. The code that will be called instead is a wrapper around the actual system call [cfs] of the C library which is often named after the wrapped system call. These wrappers exist because using them is more convenient for developers than using system calls directly. For instance, to use a system call it's necessary to setup registers, switch to kernel mode, handle the call results and switch back the user mode [scd]. This can be simplified for by implementing a wrapper and doing these tasks in the wrapper's code.

When inspecting the `fork` wrapper function [fsc] it becomes clear that the actual `fork` system call that's supposed to be wrapped is not being used it all. Instead, the `ARCH_FORK` macro [fwc] gets called which is an inline system call to `clone` defined as:

```
1 #define ARCH_FORK() \
2   INLINE_SYSCALL (clone, 4, \
3       CLONE_CHILD_SETTID | CLONE_CHILD_CLEARTID | SIGCHLD, 0, \
4       NULL, &THREAD_SELF->tid)
```

Listing 7: `clone`: Inline System Call

This results in the `clone` and `fork` library functions calling the same system call, namely `clone`. This can be verified by compiling a C application containing a call to `fork` and using `strace` to trace the resulting system calls when executing the resulting binary: No calls to `fork` are present, only a call to `clone` with the `SIGCHILD` flag. This ultimately implements the legacy `fork` call with a call to `clone`. The reason for that results in `clone` being a more powerful and configurable call than `fork`, making it possible to replace `fork` entirely by `clone` to spawn processes *and* threads.

The `clone` system call [clm] accepts various flags to configure the process creation. For the usage of namespaces a subset of these flags can be used to specify the new namespaces a process will join. By default the child processes are being initialized with a modified copy of the parent's namespace configuration when supplying such a `NEW_*` flag. This takes the desired namespaces configuration into account and makes the child process a member of the new namespaces that are represented as flags. If a specific `NEW_*` flag of a namespace type is *not* specified, then the process is part of the parent's namespace for this specific type, providing no additional level of isolation. The flags responsible for the namespace creation are:

Table 2: Namespaces: Clone Flags

Flag	Namespace Type
CLONE_NEWCGROUP	Control group
CLONE_NEWNS	Mount
CLONE_NEWUTS	Unix time sharing (UTS)
CLONE_NEWIPC	Inter process communication (IPC)
CLONE_NEWPID	Process identifier (PID)
CLONE_NEWUSER	User
CLONE_NEWNET	Network

Please note that there are more namespaces being added to the Linux kernel over time and this list is subject to change. For example, a time namespace [tns] is expected to be rolled out in the near future.

With the exception of user namespaces, creating a new namespace requires the capability `CAP_SYS_ADMIN`. This aspect will be important when discussing unprivileged containers in chapter 4.

The `clone` call has the following prototype, allowing to specify the flags listed above:

```
1 int clone(int (*child_func)(void *), void *child_stack, int flags,
2     void *arg)
```

Listing 8: `clone`: Prototype

- `child_func`: Function pointer to the function being executed by the child process.
- `child_stack`: The downwardly growing stack the child will operate on.
- `flags`: Integer value representing all used flags as configured using an OR-Conjunction of flags.
- `arg`: Additional arguments

The following code snippet shows a minimal example of using `clone` to spawn a shell process in a new UTS namespace. Please note that this minimal example does not provide includes, error handling and does not check return values.

```
#define STACKSIZE 8192
char *childStack;
char *childStackTop;
pid_t childPid;

// Will be executed by the child process
int run(void *) {
    system("/bin/sh");
    return 0;
}

int main(int argc, char const *argv[]) {
    childStack = (char *)malloc(STACKSIZE);
    // stack grows downward
    childStackTop = childStack + STACKSIZE;
    childPid = clone(run, childStackTop,
        CLONE_VFORK | CLONE_NEWUTS, 0);
    return 0;
}
```

Listing 9: `clone`: Usage Example

After compiling and running this example a shell is being spawned. Similar to a previous example in section 2.5.2, the shell resides in a namespace with an isolated hostname. This makes it possible to change the hostname of the namespace without affecting the system's hostname.

3.1.3 `unshare`

This system call allows processes to disable sharing namespaces *after* they have been created. In contrast, `clone` causes child processes to be moved to namespaces while creating them. There's a CLI application available called `unshare` that has already been discussed in the example in section 2.5.2. This command line tool creates a new process, whereas using the system call in a C program isolates a process at runtime without allocating a new process:

```c
int main(int argc, char const *argv[]) {
    unshare(CLONE_NEWUTS); // No error handling
    // Print hostname
    system("hostname");
    // Set hostname
    system("hostname testname");
    // Print hostname again
    system("hostname");
    // Print the available namespaces
    system("lsns");
    // Spawn a shell
    execvp("/bin/sh", 0);
    return 0;
}
```

Listing 10: `unshare`: Usage Example

When compiling and running the program a similar output to the following listing content can be observed:

```
root@box:~# ./a.out
box
testname
[...]
NS         TYPE      NPROCS   PID USER            COMMAND
[...]
4026531835 cgroup    336        1 root            /sbin/init splash
4026531836 pid       293        1 root            /sbin/init splash
4026531837 user      293        1 root            /sbin/init splash
4026531838 uts       333        1 root            /sbin/init splash
4026531839 ipc       336        1 root            /sbin/init splash
4026531840 mnt       326        1 root            /sbin/init splash
4026532009 net       292        1 root            /sbin/init splash
[...]
4026532436 uts         3 27750 root            ./a.out
[...]
root@box:~# lsns | grep uts
4026531838 uts       134   803 user            /bin/sh
```

Listing 11: `unshare`: Example Output

As seen above, the created process is able to set its own hostname, ultimately isolating the value of this setting in-place. Additionally, the spawned process is present in *two* UTS namespaces. The parent process is only a member of one UTS namespace. According to the namespace IDs, one of these UTS namespaces is shared as it's the default UTS namespace. The other namespace is the one that has been created using the `unshare` call.

The command `lsns` gathers the displayed information by checking the contents of the `/proc/<PID>/ns` directory for each PID. In the context of containers it should *not* be possible to get information on parent namespaces like it's possible in this example (see 3.1.4). When creating a container the `/proc` directory or sensitive parts of it should therefore be isolated to prevent this information leak.

3.1.4 `setns`

To add processes to already existing namespaces, `setns` is being utilized. It disassociates a process from its original namespace and associates it with another namespace of the same type. The prototype of this system call is as follows:

```
1 int setns(int fd, int nstype)
```

Listing 12: `setns`: Prototype

- `fd`: File descriptor of a symbolic link representing a specific namespace as represented in `/proc/<PID>/ns`.
- `nstype`: This parameter is designated for checks regarding the namespace type. By passing a `CLONE_NEW*` flag, the namespace type of the first parameter is checked before entering the namespace. This makes sure that the passed file descriptor indeed points to the desired namespace type. To disable this check, a zero value can also be used for this parameter.

A simple example that invokes a command in a given namespace can be examined in the following code listing ([lnf] – modified):

```
1 [...]
2 // Get namespace file descriptor
3 int fd = open(argv[1], O_RDONLY);
4 // Join the namespace
5 setns(fd, 0);
6 // Execute a command in the namespace
7 execvp(argv[2], &argv[2]);
8 [...]
```

Listing 13: `setns`: Usage Example

The code above launches a child process that executes the specified command and resides in a different namespace as the parent.

To perform this from a CLI the `nsenter` application can be of use. Consider the shell process launched by executing the code present in listing 10 and the corresponding output of `lsns`: By executing `nsenter -a -t 1` the shell process is being moved to all namespaces originating from the system initialization process with PID `1`. This effectively reverts the call to `unshare`, making the *original* UTS namespace available to the shell process.

Now changing the hostname from the shell process will affect the hostname of the default namespace and therefore the system's hostname. As a result, it's important to prevent these types of `setns` calls by isolating the available namespaces as already described in 3.1.3. This illustrates that by using namespaces alone it may not be possible to prevent system modifications.

A possible use-case for `setns` is to enter already existing container environments, for instance using `docker exec`. This allows spawning shells in containers to perform command line based tasks in the context of a container. An easy way to test this is to execute `nsenter -a -t <PID>` with the container PID determined by issuing `lsns`. This also evades all potential limits applied by `cgroups` because the process created by `nsenter` does not enter control groups [nse].

In the following chapters all available namespaces that are implemented in the Linux kernel at the time of writing this thesis will be explained.

3.1.5 UTS Namespace

As previously mentioned in 2.5.2, this flag provides isolation for various host identifiers. For example, this affects a subset of the values returned by executing `uname -a`, namely the hostname and domainname that's available for a process. Later on in this chapter, the kernel view of namespaces will be explained using the example of the UTS namespace.

Container Engines make use of this namespace by providing each container with its own hostname to allow a distinction of containerized processes by default.

3.1.6 Mount Namespace

This namespace type was the first to be added to the Linux kernel `2.4.19` in 2002 [nsf]. This also explains the name of the `clone` flag `CLONE_NEWNS` in which *NS* stands for *new namespace*: It seems the kernel developers did not consider additional namespaces to follow up and therefore used a very generic name for this flag.

The goal of mount namespaces is to restrict the view of the global file hierarchy by providing each namespace with its own set of mount points. Before this namespace type was introduced, `chroot` environments had to be used to perform a subset of the tasks mount namespaces are able to perform [Iva17].

A newly created namespace initially uses a copy of the parent's mount tree. To add and remove mount points, the `mount` and `umount` commands are available. The implementation of these commands had to be modified in order to be aware of namespaces and work in combination with mount namespaces[nsf].

The initial implementation of mount namespaces caused a usability issue that ultimately reduced the efficiency of these namespaces: To make a device available for all or a subset of all namespaces it was required to execute one `mount` operation for each namespace. However, the optimal approach would require only a single mount operation to perform the same task. Additionally, a solution to manage mountpoints in all namespaces at once would provide more convenience. For these reasons, *shared sub-trees* have been introduced [sst].

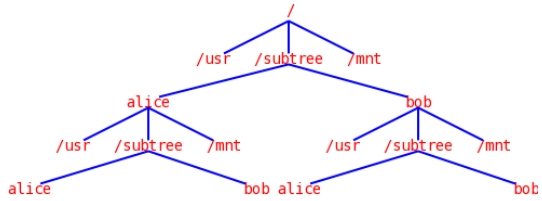
The basic idea of shared sub-trees is that a *propagation type* is associated with each mount point [nsf]. This configures how each `mount` operation within a mount point will be propagated to other related mount points. Internally the kernel uses *peer groups* to determine whether a `mount` event gets propagated to a specific mount point. A peer group consists of a set of mount points that share mount events with each other. Mount points get added to a peer group in case the specific *shared* mount point is being replicated by joining a new mount namespace or when bind-mounting a mount point. Peer group members are being removed in case an unmount operation is being issued or as soon as a mount namespace gets destroyed.

The following propagation types are available:

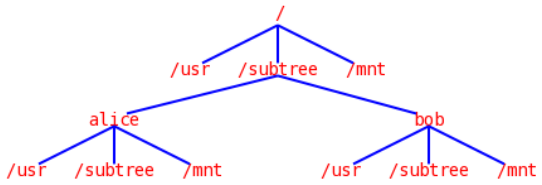
- `MS_SHARED`: This shares the mount with all mount points residing in the same peer group. Mount operations on a peer's mount will also be propagated to the original mount.
- `MS_PRIVATE`: No mount events are being propagated and no events are being received. Other mount namespaces will not be able to access the mount point.
- `MS_SLAVE`: Mounts of this type only receive events. They do not share events and effectively provide private mounts for mounts created using this propagation type.
- `MS_UNBINDABLE`: This type is similar to the private type with an addition: Mounts of this type can not be used to create bind mounts. An example usage of this type can be found below.

The types can be prefixed with an `r` to make its effect recursive for all child mounts of a mount point.

Consider the following scenario [nsf]: Two users, namely Alice and Bob, are sharing parts of the same file hierarchy and are being placed in their own mount namespaces. They are then being provided with their own view of the system directories. To perform this task, the system's directories have been bind-mounted into the user specific directories of the sub-tree, as shown on the left figure below. Note that the folders of Alice and Bob are replicated in the bind-mounted sub-tree directories which is not desired. By using the `MS_UNBINDABLE` flag this is prevented, effectively creating the hierarchy as can bee seen on the right in the figure below:



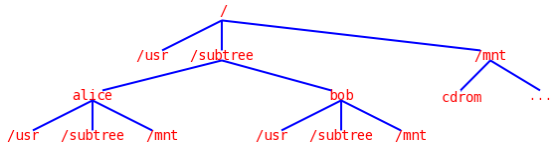
(a) Subtree Creation Using Bind Mounts



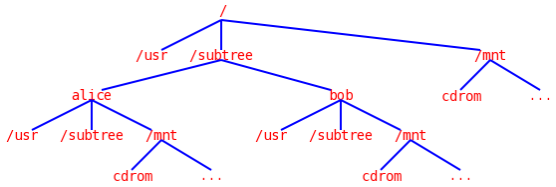
(b) Usage of Unbindable Mounts

Figure 17: Mount Namespace: Per-User Bind Mounts (Modified) [nsf]

To make use of the advantages provided by shared sub-trees, a `mount` now has to be shared with both users. As can be seen on the left figure below, a CD-ROM was inserted. However, it's only present outside of the mount namespaces of Alice and Bob. By executing a `mount` command with the `--make-shared` flag, the mount of the CD-ROM is now also present in the mount namespaces of both users:



(a) Bind-mounting a mount point



(b) Sharing a mount

Figure 18: Mount Namespace: Sharing Mounts (Modified) [nsf]

Moreover, every mount operation executed in the `/mnt` directory will now automatically be propagated to the mount namespaces of Alice and Bob.

This approach also enables Alice and Bob to have private mounts that are not being shared with other users. This can be achieved by either making the sub-tree mounts of the users `slaves` or by creating private mounts inside of the respective user directories.

The mount points available for a specific process of a namespace can be found along with their propagation types in the `/proc/self/mountinfo` file.

By default *Docker* uses the private propagation type [dpt] when mounting directories with the `-v` parameter. This can be verified by examining the output of `docker inspect <Containername>`:

```
[...]
"Mounts": [
    {
        "Type": "bind",
        "Source": "/tmp/test",
        "Destination": "/tmp/test",
        "Mode": "",
        "RW": true,
        "Propagation": "rprivate"
    }
],
[...]
```

Listing 14: Bind Mounts: Docker Example

This isolates the mount points of both the host and a container and no mount events are being propagated between a container and the host. Therefore mount points present on the host are not available for containerized processes and the other way around because separate mount namespaces are in place.

Consider sharing a host directory with a container after starting an initial process in it: In this scenario mounting takes place across two mount namespaces – the host namespace and the respective container namespace. The `nsenter` utility assists when performing this task [jpe] by using `setns` to join the container's mount namespace in order to allow creating mount points in the container. Because the capabilities in the host's mount namespace are being preserved, a folder governed by the host can therefore be mounted into the container, effectively sharing it between the host and the container.

For containers, host paths can also be *masked* to provide each container with its own version of a specific folder. For example by masking the `/proc/acpi` it's not possible for a containerized process to enable host interfaces like the Bluetooth device. Issues can be present in case no proper masking is taken into account [dic]. The *Docker* platform automatically masks certain paths under `/proc` and `/sys` to counter these potential issues.

During an assessment of the *Docker* platform two information leaks regarding the `/proc/asound` path were discovered in the OCI specification:

- **Leak of audio device status of the host**: When media is being played on the host, the `/proc/asound/card*/pcm*p/sub*/status` files may contain information regarding the status of media playback. Consider this command for a demonstration:

```
docker run --rm \
    ubuntu:latest bash -c \
        "sleep 5; \
        cat /proc/asound/card*/pcm*p/sub*/status | \
        grep state | \
        cut -d ' ' -f2 | \
        grep RUNNING || echo 'not running'"
```

Listing 15: Mount Namespace: Leaking Audio Device Information

  When playing a media file, watching a video or executing similar actions involving sound output on the host, the command above demonstrates that a containerized process is able to gather information on this status. Therefore a process in a container is able to check whether and what kind of user activity is present on the host system. Also, this may indicate whether a container runs on a desktop system or a server as media playback rarely happens on server systems.

  The scenario described above is in regard to media playback. Moreover, when examining the `/proc/asound/card*/pcm*c/sub*/status` files (`pcm*c` instead of `pcm*p`) this can also leak information regarding capturing sound, as in recording audio or making calls on the host system.

- **Leak of Monitor Type**: Monitors can also act as sound output devices when connected using a compatible interface like DisplayPort. The listing below illustrates how the model type of the connected monitors can be read from within a container:

```
docker run --rm \
    ubuntu:latest bash -c \
        "cat /proc/asound/card*/eld* | \
        grep monitor_name"

monitor_name SMS24A650 # (A Samsung monitor)
monitor_name SMS24A650
```

Listing 16: Mount Namespace: Leaking Monitor Type

  This information should not be accessible from within a container as it contains specific information on the host and its environment.

These issues have been reported [dix] to the *Docker* maintainers and are expected to be fixed in an upcoming release by adding `/proc/asound` to the list of paths that are being masked to make each container manage its own versions of the affected paths. This path list is part of the OCI specification. Therefore, the fix for these issues also propagates to `containerd` which also uses the OCI specification.

3.1.7 PID Namespace

By creating a PID namespace, the process ID number space gets isolated. This makes it possible that processes in a container have PIDs starting from the value `1` whereas the *real* PID outside of the namespace of the same process is an entirely different number. Therefore multiple processes can have the same PID value on a system while they reside in different PID namespaces. With this namespace type processes of a container can be suspended and resumed, making the PIDs of the processes unique and persistent within such a namespace. Furthermore, containers can have their own `init` processes with a PID value of `1` that prepare the container internals and reap child processes themselves rather than delegating this task to the system wide `init` process. Once the `init` process of a PID namespace terminates, all processes of the namespace will also be terminated using a `SIGKILL` signal. This can also be caused by a reboot of the system which terminates all `init` processes of the PID namespaces. Child processes with parents residing in other namespaces get reaped by their parents across namespace boundaries. This can be overridden by issuing a call to `prctl` with the `PR_SET_CHILD_SUBREAPER` flag (as already mentioned in 2.7.1).

Joining namespaces of this type limits process communication: It's not possible to interact with processes of other PID namespaces in a way that requires a process identifier when issuing system calls. For example, it's important to note that processes can only send signals to other processes that reside in their current namespace or in namespaces descending from their current namespace. This isolation is one of the main use-cases for this namespace type in containerization.

PID namespaces allow nesting up to a depth of 32, resulting in one PID in the original namespace and one in the new namespace for each process. This creates one PID value in each namespace that's a direct ancestor walking back to the root namespace. Changing PID namespaces is a one-way operation. This means that it's not possible to move a process up to an ancestor namespace.

Moving a process to its own PID namespace does *not* isolate the list of processes visible for that process. This can be verified by creating a shell in its own PID namespace and examining the process list:

```
1 user@box:~$ sudo unshare -fp /bin/bash
2 root@box:~# ps aux
3 USER  PID [...] COMMAND
4 [...]
5 user  5380 [...] /usr/lib/[...]/chromium-browser --enable-pinch
6 user  5388 [...] /usr/lib/[...]/chromium-browser --type=zygote
7 [...]
```

Listing 17: PID Namespace: Example Without `/proc` Mount

As can be seen in the listing above, processes residing in other namespaces are still visible. The process list is determined by processing the `/proc/<PID>` folders to gather the displayed information and this is not being isolated by joining a new PID namespace. By appending the option `--mount-proc` to the command used above, this issue is being

resolved by creating a new mount namespace and remounting the `/proc` directory using `mount -t proc proc /proc`. This effectively isolates the directory contents of `/proc` and prevents a container from accessing the information stored in the `/proc` directory for processes residing in other namespaces. Employing this remounting mechanism is a part of the default behavior of *Docker* and other container engines. Therefore, leaking information of host process and accessing parts of these processes by sharing `/proc` is prevented.

A simple C usage example for this namespace type can be examined below:

```
1  [...]
2  int run(void *) {
3      std::cout << "[Child] PID: " << getpid() << std::endl;
4      std::cout << "[Child] Parent PID: " << getppid() << std::endl;
5      system("/bin/sh");
6      return 0;
7  }
8
9  int main(int argc, char const *argv[]) {
10     [...]
11     childPid = clone(run, childStackTop, CLONE_VFORK | CLONE_NEWPID,
12         0);
13     std::cout << "[Parent] Child PID: " << childPid << std::endl;
14     [...]
15     return 0;
16 }
```

Listing 18: PID Namespace: Code Example

After compiling and executing this codes it becomes clear that the child is running with PID `1` in the isolated environment and a different, much higher identifier outside of the namespace.

A process in a new PID namespace can be created by providing the `CLONE_NEWPID` flag. When using the same flag in combination with `unshare` or when calling `setns`, a new PID namespace will be created. This namespace is present in the `/proc/<PID>/ns/pid_for_children` file. The process calling `unshare` or `setns` will *stay* in its current namespace [mpn]. Instead, the first child of this process will be placed in the new namespace with a PID value of `1`, making it the container's `init` process. This results from the assumption of many libraries and applications: PIDs of processes are not subject to change. By joining a new namespace at runtime, the identifier of a process would change. In fact, even C library functions like `getpid` cache the PIDs of processes [mpn].

Ever since PID namespaces (3.1.7) have been implemented, PIDs are represented by a kernel structure called `pid` [cgx]. In combination with the `upid` structure (`include/linux/pid.h`) it's possible to determine the correct `pid` structure as seen in a specific PID namespace.

3.1.8 Network Namespace

This namespace can enable processes to have their own private network stack, including interfaces, routing tables and sockets [nns]. The corresponding `clone` flag is `CLONE_NEWNET` and the `ip` CLI application is available to manage network namespaces easily. The `ip netns` command is able to create a new permanent network namespace. This is being accomplished by bind-mounting `/proc/self/ns/net` to `/var/run/netns/<Name of namespace>`. Using this, configuration can take place without moving processes to the namespace first.

To create a network namespace from the command line, the `ip netns add` is used as follows:

```
1 root@box:~# ip netns add one # Create a network namespace
2 root@box:~# strace ip netns add two # Trace the creation
3 [...]
4 unshare(CLONE_NEWNET) = 0
5 mount("/proc/self/ns/net", "/var/run/netns/two, [...]) = 0
6 [...]
7 root@box:~# ip netns exec one ip link # Execute command in NS
8 1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT [...]
9     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Listing 19: Network Namespace: Code Example

As seen in the output listed above, adding a new network namespace is accomplished by following these steps:

1. Entering a new network namespace by using `unshare` with the `CLONE_NEWNET` flag
2. Executing the mount operation mentioned above

Deleting a network namespace can be done analogous. However, a namespace only gets destroyed in case all processes residing in the namespace are terminated. Otherwise it's marked for deletion.

By default a new network namespace comes with its own local interface which is down by default. To perform the configuration of a namespace it may be convenient to spawn a shell in the desired network namespace with `ip netns exec <name> bash`. Among others, the following configuration scenarios are possible:

- Linking network namespaces in order to provide a way for different namespaces to communicate by creating a network bridge.
- Routing a network namespace through the host network stack to enable internet access.
- Assigning a physical interface to a namespace. Please note that each network interface belongs to exactly one namespace at a given point of time. The same is true for sockets.

To enable the communication between the default network namespace and another namespace, virtual interfaces can be of use. These interfaces come in pairs of two – In this example `veth0` and `veth1` allowing a pipe-like communication:

```
# Create the virtual interface pair
root@box:~# ip link add veth0 type veth peer name veth1
# Move veth1 to the namespace named `one`
root@box:~# ip link set veth1 netns one
# Set the IP for veth1 in the new namespace
root@box:~# ip netns exec one ifconfig veth1 10.1.1.1/24 up
# Set the IP in the default namespace
root@box:~# ifconfig veth0 10.1.1.2/24 up
```

Listing 20: Network Namespace: Usage of Virtual Interfaces

As seen above it's possible to move an interface to a different namespace. This can for example be used to enable the communication between containers, similar to the functionality provided by `docker --link`. Therefore private container networks can be built that are isolated from the host and other containers on a host. Also, internet access for containers can be provided this way. While the *Docker* platform does this by default using a bridge network, it may be necessary to configure parts of this aspect manually for other container engines.

Moving an interface back to the default network namespace is accomplished with the command `ip link set eth0 netns 1`: It detects the correct namespace according to the specified PID with value `1`, allowing administrators to identify namespaces by the process identifiers present in a given namespace. Moving interfaces between namespaces is implemented in the `dev_change_net_namespace` function of the Linux kernel [nnk]. Besides stopping and moving the desired interface it also notifies all processes using the interface in order to flush the message chains.

A common use-case for network namespaces is running multiple services binding to the same port on a single machine. This works by allocating a different port in the host's network namespace for each port that's being exposed by a container network namespace. Therefore all containerized web services can be configured to run on the default port `80` while they are being exposed on an entirely different port. Consequently applications do not require information on the real port that's being used to expose the service and the host system can map ports freely. This provides a certain degree of portability when deploying containers that use network functionality.

3.1.9 IPC Namespace

Similar to the namespaces that were discussed in previous sections, IPC namespaces also have the goal to isolate a certain aspect of a system: Message queues. The Linux kernel implements several ways by which an inter-process communication can be performed. Two of these mechanisms are *System V* message queues and POSIX message queues. With POSIX message queues being a newer concept [mqo] this section describes IPC namespaces by using message queues of this type as an example.

According to the Linux namespace documentation, IPC objects like message queues can be accessed using identifiers other than files and directories. This is the reason why a separate namespace for this resource type has been introduced [nso].

To get a better understanding of how message queues work, a short code snippet [mqe] is listed below:

```
1  void * queue_server(void * args) {
2      [...]
3      // Create the message queue
4      mq = mq_open(QUEUE_NAME, O_CREAT | O_RDONLY | O_NONBLOCK, [...]);
5      [...]
6      // Receive messages
7      bytes_read = mq_receive(mq, buffer, MAX_SIZE, NULL);
8      [...]
9  }
10
11 void * queue_client(void * args) {
12     [...]
13     // Open the queue
14     mq = mq_open(QUEUE_NAME, O_WRONLY);
15     [...]
16     // Send messages
17     mq_send(mq, buffer, MAX_SIZE, 0);
18     [...]
19 }
20
21 int main(int argc, char** argv) {
22     [...]
23     // Create two threads, one sender and one recevier
24     pthread_create(&server, NULL, &queue_server, NULL);
25     pthread_create(&client, NULL, &queue_client, NULL);
26     [...]
27 }
```

Listing 21: IPC Namespace: Usage Example

The basic structure of the example is as follows: A sending and receiving thread both have their own functions that are being executed. Using a shared queue name, namely `QUEUE_NAME`, the communication is being accomplished. The sender stores messages in the queue while the receiver reads messages out of the same queue one after another. The knowledge of the shared queue name is enough to let other processes access the same data structure these functions are using.

After compiling and running the example above, a file named according to the value of `QUEUE_NAME` in the code is being created [mqo] to represent the message queue. Without using IPC namespaces every process is able to read and write message from and to the queue just by specifying the queue's name. This is being prevented when using a IPC namespace.

3.1.10 Control Group Namespace

By entering a new *cgroup* namespace, the paths `/proc/<PID>/cgroup` and `/proc/<PID>/mountinfo` are being isolated from the parent's namespace. This prevents information leaks that can occur when viewing these files from a container as information regarding the host system may be present [cgm]. This could enable users in a container to determine the type of container engine that's in use or to gather information on valid user IDs of the host system:

```
1 root@container:~# cat /proc/self/cgroup
2 12:pids:/user.slice/user-1000.slice/session-2.scope # UID leak
3 [...]
4 9:memory:/user/root/0
5 7:devices:/user.slice
6 0::/user.slice/user-1000.slice/session-2.scope
7 [...]
```

Listing 22: CGroup Namespace: Information Leak

While the internals of *cgroups* are discussed in section 3.2, it nevertheless becomes clear that otherwise not accessible information is being leaked. In contrast, when entering a new control group namespace with `unshare -C` these paths are being changed to appear as the root directory or relative to the control group namespace root in order to prevent leaking information [cgm]. Relative paths in the *cgroup* file allow processes not only to be moved downwards in a control group hierarchy but also upwards [cgz]. The ultimate effect is that the control group hierarchy in a container is the result of bind-mounting the container's *cgroup* sub-tree at the *cgroup* root in the container [cgz].

An added advantage of control group namespaces is the additional abstraction layer of the *cgroup* paths which may allow migrating containers more easily. It also prevents processes from accessing ancestor control groups by making the respective directories inaccessible.

3.1.11 User Namespace

This namespace type introduces mapping user and group IDs and the isolation of capabilities per-namespace. For instance, a process can run with a non-zero UID outside of a user namespace while having a UID of zero in a namespace. This ultimately enables unprivileged users to have root privileges in isolated environments. Moreover, this is the only namespace type not requiring the `CAP_SYS_ADMIN` capability, allowing unprivileged users to create this namespace type.

By mapping the UID in the namespace to the original UID, all changes and actions introduced by a user will be mapped to the set of privileges a user holds in the parent namespace. This prevents actions in the global system context that the user normally would not be allowed to perform. This includes sending signals to processes outside of the current namespace and accessing files [unz]. For example, a user with root privileges in a namespace is not able to read the `/etc/shadow` file because the original privileges do

not allow to do so. The same applies to the mapping of Group Identifier (GID) values. Applied changes, for example creating new files, will be shown as originating from the user outside of the namespace. Likewise, calling `stat` will map IDs in the opposite direction in order to create a correlation in regard to the applied mappings.

This namespace type can also be nested, similar to PID namespaces (see 3.1.7) . Using `ioctl` operations it's possible to inspect the relationships between different user namespaces [ion].

By default a process only has a specific capability in a user namespace in case it's a member of the user namespace and the capability is present in the corresponding capability set [unz] (see 2.3).

Linux capabilities are aware of user namespaces as can be seen in the function `has_ns_capability` of `kernel/capability.c`: With this function the kernel is able to check whether a process, as represented by a `task_struct` in the kernel code, has a capability in a specific user namespace. By default the first process in a new user namespace is granted a full set of capabilities in that namespace. While having all capabilities in the user namespace, the process does not have *any* capabilities in the parent user namespace. This means:

- A process with all capabilities in a user namespace is able to perform privileged operations on resources that are solely governed by this specific user namespace.
- The same process is not able to perform changes to resources governed by the parent namespace, the initial namespace or outside of namespaces. For example even with `CAP_SYS_TIME` in the user namespace it's not possible to alter the system clock because this is not governed by a namespace type as of now and therefore `CAP_SYS_TIME` in the initial namespace is required in this example.

In case a process has the `CAP_SYS_ADMIN` capability it's possible for it to enter an arbitrary already existing user namespace using `setns`. In that case, this process also gains a full capability set in the entered namespace. Also, if a process possesses a capability in a parent user namespace, it also has this capability in all child namespaces. All processes that run with the same UID as the process that created a user namespace have a full capability set in a namespace and all of its children. This can be verified by creating two namespaces as siblings. Processes in the parent namespace are able to use `setns` to switch to each of the namespaces. However, it's not possible for one of the child namespaces to do switch to a sibling namespace because of the lack of `CAP_SYS_ADMIN` in the target namespace.

In the scenario of containers it's common practice to reduce the set of available capabilities a container and all its processes may have. The reduced capability set that's in place by default for *Docker* containers is present in the OCI specification's source code [dgc]. Missing capabilities can be added and removed on demand when starting a new container – for simple use-cases the pre-defined set can serve as a secure base. All capabilities can be granted to a process with the `privileged` flag although its use should be evaluated carefully in terms of security.

Without reducing the capability set or by granting all privileges to a container, drastic changes to a system and its state may be employed in case of compromise. This includes:

- Loading custom kernel modules
- Tracing arbitrary processes to interfere with their program flow or to leak sensitive information
- Changing the ownership of arbitrary files
- Send arbitrary signals to processes
- Spoof network packets using raw sockets

As the list above suggests, reducing the capability set via user namespaces is crucial for containerization. In cases where untrusted code is being executed in containers or compromised containers are part of the threat model, changes to the host system must not be possible without explicitly allowing this kind of modification.

Every process in a user namespace has two files which can be used to perform a UID and GID mapping:

- `/proc/<PID>/uid_map`
- `/proc/<PID>/gid_map`

By default, these files are empty. The kernel expects lines following the format `ID-inside-ns␣ID-outside-ns␣length` to be present in these files. The length parameter is used to create a range of possible IDs, starting from `ID-inside-ns` respectively `ID-outside-ns` with the maximum value according to the `length` field value. In case no mapping is being performed for an ID, the value of `/proc/sys/kernel/overflow{u,g}id` is used. This causes an ID to be mapped to the `nobody` ID. Also, the kernel silently prevents elevating privileges when set-user-ID binaries are being executed [unm]. Please note that an ID mapping can only be performed *once* and is limited to a maximum of 340 lines [unm].

Consider the listing below which helps to understand the mapping process:

```
1  int run(void *) {
2      while (true) {
3          std::cout << "[Child] EUID/EUID "
4              << geteuid()
5              << "/" << geteuid()
6              << std::endl;
7      }
8      return 0;
9  }
10
11 int main(int argc, char const *argv[]) {
12     [...]
13     childPid = clone(run, childStackTop, CLONE_NEWUSER, 0);
14     std::cout << "[Parent] Child PID: " << childPid << std::endl;
```

```
15      sleep(5);
16
17      // echo "0 1000 1" >> /proc/<PID>/uid_map
18      std::string cmd = "echo '0 " +
19                        std::to_string(getuid()) +
20                        " 1' >> /proc/" +
21                        std::to_string(childPid) +
22                        "/uid_map";
23
24      system(cmd.c_str());
25      while (true) { [...] }
26      [...]
27      return 0;
28  }
```

Listing 23: User Namespace: Usage Example

The code spawns a child process that prints its effective UID and GID values in an endless loop. After five seconds have passed, the parent process will perform a UID mapping in order to map the root user in the new namespace to the user executing this code. This produces the following output:

```
1  [Parent] Child PID: 22338
2  [Child] EUID/EUID 65534/65534
3  [...] // five seconds pass, mapping is done
4  [Child] EUID/EUID 0/0
5  [...]
```

Listing 24: User Namespace: Output of Usage Example

This produces the same UID mapping as executing `unshare -r /bin/bash` does. With the new mapping values it's now possible for the kernel to check the privileges a user possesses in the namespace by mapping the ID values back to the original values and performing permission checks with the values outside of the namespace. One of the use-cases for this is to allow unprivileged users on the host to be privileged in a container. After some additional configuration, this for example enables these users to configure parts of the container and install software packages. Container engines like *Docker* perform this task by default.

There exist certain additional rules [unz] for applying UID/GID mappings:

- The UID mapping file is being owned by the user that created the namespace. Because of this, only this user and `root` are able to write to this file. However, there's an exception as mentioned below.
- To write to the mapping files, the capability `CAP_SET{U, G}ID` has to be present in the context of the target process.

- If the data to be written to one of the files only contains a single line: The initial process in a user namespace is allowed to map its *own* effective UID/GID values from within the namespace.
- Otherwise: Arbitrary mappings can be added by a process residing in the parent namespace.

Depending on the namespace a process resides in, there exist differences in how the `ID-outside-ns` values are being interpreted. Consider the figure below illustrating this scenario. A user with UID `1000` creates two namespaces with different ID mappings. After that, the initial processes of both namespaces read the applied mappings of each other:

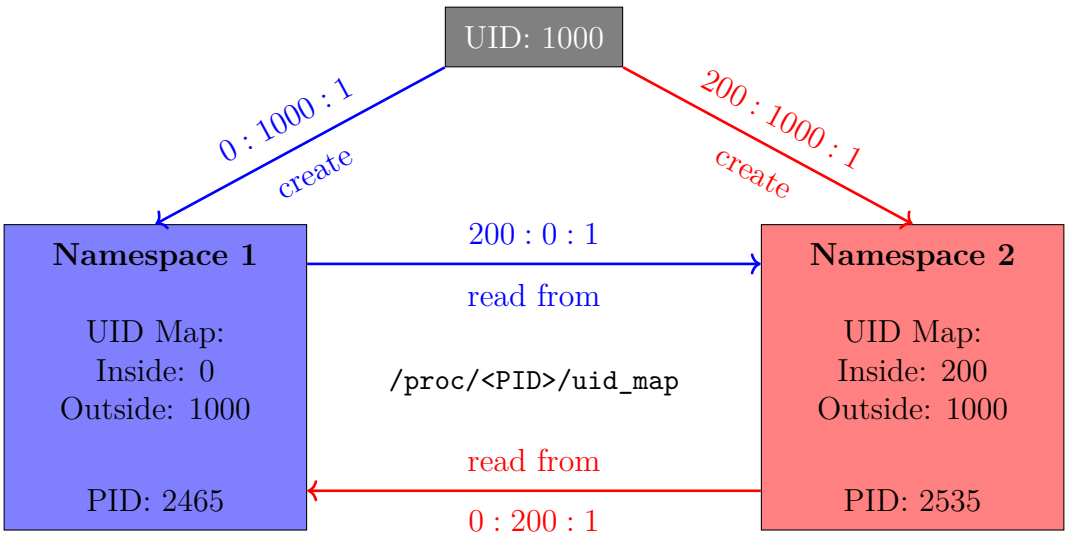


Figure 19: User Namespace: UID Mappings (With Information from [unz])

Consider the initial process writing to its own mapping file: To map the user identifier `1000` to its `root` user, a namespace has to use the same configuration as the one which namespace `1` receives above. This results in the `ID-outside-ns` value having to be interpreted as a UID of the parent namespace in case two processes reside in the same namespace.

However, if they are in different namespaces, as seen above, something else can be observed: `ID-outside-ns` is interpreted as being relative to the namespace the process that's being read from is being placed in. Namespace `1` reads `200:0:1` from namespace `2` – this means that the UID `0` in namespace `1` corresponds to the ID `200` of namespace `2` and both originate from the user identifier `1000`. The same applies to namespace `2` the other way around.

It's possible to combine user namespaces with other namespace types. This enables users to create namespaces that would require `CAP_SYS_ADMIN` without being privileged. To accomplish this, one can use `clone` and a combination of `CLONE_NEWUSER` and other `clone` flags. The kernel processes the flag to create a new user namespace first and processes the remaining `CLONE_NEW*` flags inside of the new user namespace.

Being privileged in a user namespace does *not* imply superuser access on the whole system. Nevertheless the ability to perform actions an unprivileged user would not be able to execute without user namespaces also broadens the attack surface. For example unprivileged users are able to execute certain mount operations that can be the target of kernel exploits [cua]. There may remain more potential security issues regarding user namespaces to be uncovered in the future. For example, it was previously discovered that the combination of user namespaces and the `CLONE_FS` flag can lead to a privilege escalation issue [unv]. More recently it was discovered that when the limit for the number UID/GID mappings a namespace can have was increased from 5 to 340, a security issue [jhz] was introduced: When switching to different data structures to store the mappings in the kernel once the number of mappings exceeds five, this data is being accessed in a wrong way. This results in processes of nested user namespace being able to access files being owned by other namespaces, e.g. `/etc/shadow` of the initial user namespace.

3.1.12 Kernel View

This section covers aspects of the Linux kernel code regarding the implementation of namespaces. For this, various points, such as *tasks*, *credentials* and *namespace proxies* are being considered. In the kernel source code, these are all represented by C structures:

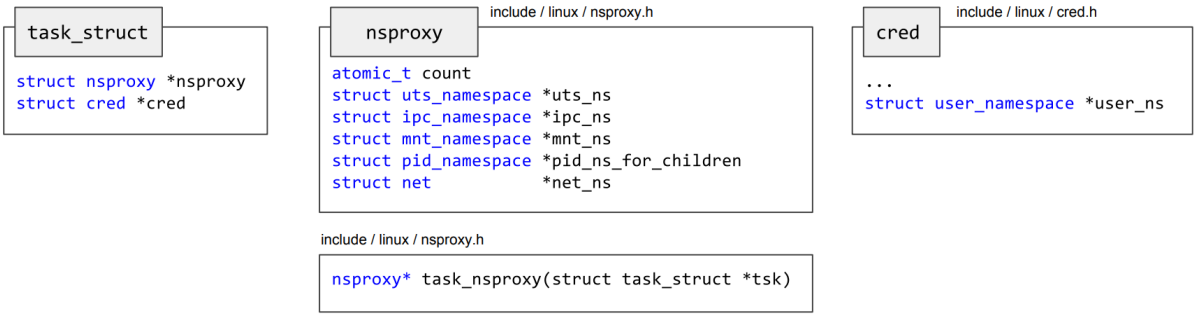


Figure 20: Namespace Kernel View: Overview [kvs]

From a kernel point of view, processes are often called *tasks* [nsi]. The kernel keeps track of them by storing task information in a doubly linked list. This list is called *task list* and contains elements of the `task_struct` type, as defined in `linux/sched.h`. These structures act as task descriptors and contain all relevant information on a task, for example:

- The process identifier (`pid_t pid`)
- Managed file descriptors (`files_struct *files`)
- Pointer to the parent task (`task_struct *parent`)
- Namespace proxy (`nsproxy *nsproxy`)
- Task credentials (`cred *cred`)

Using the credential information listed above, the Linux kernel manages the associated ownership information of certain objects, such as files, sockets and tasks. Upon requesting an action to be performed on an object, for example writing to a file, the kernel executes a security check in order to determine whether a user is allowed to perform an action in regard to the task's associated credentials. The `cred` structure is defined in `linux/cred.h` and contains a `user_namespace` pointer. This is required because credentials are relative to the specific user namespace that's currently active.

To associate a task to the namespace it currently runs in, the `nsproxy` kernel structure contains a pointer to each per-process namespace. The PID and user namespace are an exception: The `pid_ns_for_children` pointer is the PID namespace the *children* of the process will use. The currently active PID namespace for a task can be found in the `task_active_pid_ns` pointer. Moreover, the user namespace is a part of the `cred` structure as seen above.

For an example of the linked kernel structures, consider the association of a task to a specific UTS namespace in the figure below:

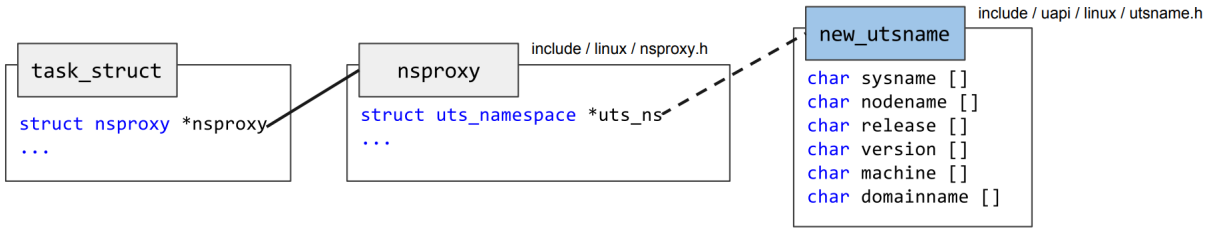


Figure 21: Namespace Kernel View: UTS Data Structures [kvs]

This implies that other parts of the kernel code also have to be aware of namespaces, e.g. when returning a hostname upon accessing this information from a task. This is implemented by modifying the original code of `gethostname` in order to read the hostname of the current namespace and not from the system's hostname:

```
1 [...]
2 i = 1 + strlen(
3     system_utsname.nodename);
4 [...]
5 if (copy_to_user(
6     name,
7     system_utsname.nodename,
8     i))
9 [...]
```

Listing 25: Namespaces Kernel View: `gethostname` System Call (Namespace Unaware)

```
1  [...]
2  u = utsname();
3  i = 1 + strlen(u->nodename);
4  [...]
5  memcpy(tmp, u->nodename, i);
6  [...]
7  if (copy_to_user(name, tmp,
8      i)
9  )
10 [...]
```

Listing 26: Namespaces Kernel View: `gethostname` System Call (Namespace Aware)

The call to `utsname` is being used in namespace aware versions of the `gethostname` call to determine the desired hostname value using the `nsproxy` kernel structure mentioned above:

```
1 static inline struct new_utsname *utsname(void)
2 {
3     return &current->nsproxy->uts_ns->name;
4 }
```

Listing 27: Namespaces Kernel View: Using `nsproxy` to Read the Hostname

Upon creating a child process, the elements of the `nsproxy` structure will be copied as the child will reside in the exact same namespaces the parent lives in. This can be observed in the `copy_process` function called in `do_fork` [cns] [vsc]. By using `copy_namespaces` [nsc], the members of the parent's `nsproxy` structure are being duplicated:

```
1  static struct nsproxy *create_new_namespaces(unsigned long flags,
2    struct task_struct *tsk, struct user_namespace *user_ns,
3    struct fs_struct *new_fs) {
4      struct nsproxy *new_nsp;
5      int err;
6      // Create a new namespace proxy
7      new_nsp = create_nsproxy();
8      [...]
9      // Copy the existing UTS namespace (among others)
10     new_nsp->uts_ns = copy_utsname(flags, user_ns,
11         tsk->nsproxy->uts_ns);
12     [...]
13 }
```

Listing 28: Namespaces Kernel View: Copying Namespaces

Upon using `unshare`, the `ksys_unshare` function [cns] is being executed. This calls `unshare_nsproxy_namespaces` [nsc] which effectively leads to the `nsproxy` structure being copied, modified according to the `unshare` call and replaced as soon as one element of the structure is being modified.

Every `nsproxy` structure contains a `count` element. This counter keeps track of how many tasks refer to the same `nsproxy`. After a task terminates, the `count` value of the associated `nsproxy` gets decremented and in case there's no other task using a specific `nsproxy`, it gets freed along with all contained namespaces:

```
1 void free_nsproxy(struct nsproxy *ns) {
2     // Check if the namespace is still in use, free if unsued
3   if (ns->mnt_ns)
4     put_mnt_ns(ns->mnt_ns);
5   if (ns->uts_ns)
```

```
6       put_uts_ns(ns->uts_ns);
7     if (ns->ipc_ns)
8           put_ipc_ns(ns->ipc_ns);
9
10    if (ns->pid_ns_for_children)
11      put_pid_ns(ns->pid_ns_for_children);
12    put_cgroup_ns(ns->cgroup_ns);
13    put_net(ns->net_ns);
14    kmem_cache_free(nsproxy_cachep, ns);
15  }
```

Listing 29: Namespaces Kernel View: Freeing Namespaces

The `put_*_ns` functions seen above are responsible for destroying unused namespaces of a specific type. Of course, this only happens in case no other task uses the particular namespace anymore.

When entering a namespace using `setns`, the function `switch_task_namespaces` is being used:

```
1   void switch_task_namespaces(struct task_struct *p,
2     struct nsproxy *new) {
3       struct nsproxy *ns;
4       [...]
5       // lock the task before switching namespace
6       task_lock(p);
7       // switch namespace
8       ns = p->nsproxy;
9       p->nsproxy = new;
10      task_unlock(p);
11
12      // delete the old namespace proxy if unused
13      if (ns && atomic_dec_and_test(&ns->count))
14          free_nsproxy(ns);
15  }
```

Listing 30: Namespaces Kernel View: Switching Namespaces

Moving a task to a specific namespace can be as simple as setting the respective pointers of the `nsproxy` structure.

3.1.13 Usage in Containerization

Both LXC and *Docker* apply a standard namespace configuration in case no further configuration is supplied. This involves setting up a new namespace for each type that has been discussed in this chapter and that's also supported by the kernel. Specific namespace configuration can be applied using the *Docker* CLI and LXC configuration files.

Additional configuration may include supplying a specific namespace a container is joining. For example, the `--net=host` option disables creating a new network namespace for a *Docker* container, allowing it to join the initial network namespace. Therefore no network isolation is in place and for instance `localhost` in a container points to `localhost` of the host, creating a behavior as if a process is running without containerization in regard to its available network environment. This allows flexible setups by allowing the user to choose the isolated resources freely.

Docker configures user namespaces in a way that maps a non-existent UID as `root` in a container. Therefore, escalating to another user namespace by abusing a potential vulnerability does not add any privileges [dds]. This is being performed by using the `/etc/sub{u, g}id` files. By creating a *subordinate* ID mapping in these files, a range of non-existing IDs is assigned to each real user of the host. This ensures that the ranges do not overlap and are in fact disjunct. Using this range non-existing users are being mapped into a container, starting from a zero value which corresponds to `root`.

3.2 Control Groups

The goal of *cgroups* is to enable fine-grained control over resources consumed by processes additionally to resource monitoring. Before this Linux kernel feature was available, other mechanisms such as `nice` or `setrlimit` had to be used [cgt] to replicate a subset of the features that are being offered directly by today's kernels. However, without the ability to group processes and restore previously applied settings this was not as convenient as using control groups is today. Next to namespaces this kernel feature is one of the main primitives that are being used to build container environments. Limiting and monitoring of containers with the control groups described in this section can be applied to processes as well as containers.

By organizing control groups in a virtual filesystem called `cgroupfs`, taking advantage of the hierarchical structure of control groups in implementations becomes possible. As seen in the example in section 2.5.2, control groups are created, deleted and modified by altering the structure and files of this filesystem. Also, sub-groups are possible that allow the inheritance of *cgroup* attributes. This implies that limits in parent groups can also apply in child groups. It's also possible that this affects child processes – for example a `fork` call can cause the newly created process to be affected by the same limits as the parent. Please note that, depending on the type of control group that's in use, these aspects may or may not be applicable.

Consider the following use-case: On Ubuntu `systemd` prevents systems from being crashed by fork bombs by automatically creating a default control group for each user. This behavior results from the effect of the `.slice` sub-group that's limiting the number of processes a single user may create, preventing users from spawning processes in an infinite loop. By executing `systemctl status user-$UID.slice` the current limit is shown.

Specific parts of the offered control group configuration, like the limit described above, are configurable in a granular way using *controllers*. Controller types are also called *subsystems* of *cgroups* and control the aspect of resource usage that may be limited or monitored. Throughout this chapter, various controller types will be discussed. The kernel code regarding *cgroups* is responsible for *grouping* processes whereas the individual

controller implementations takes care of the resource monitoring and limiting functionality itself.

Modifications to control groups can be applied once the virtual filesystem of the controller to use is mounted. One may choose to mount a single controller type or mount every available type at once [mcg]. In any case, mounting controllers usually takes place in `/sys/fs/cgroup` where at least one folder for each controller is being created by calling `mount`. It's important to note that it's not possible to mount an already mounted controller to a different location without also mounting all other controllers that are already mounted to prevent ambiguity. The following figure illustrates the mounting procedure:

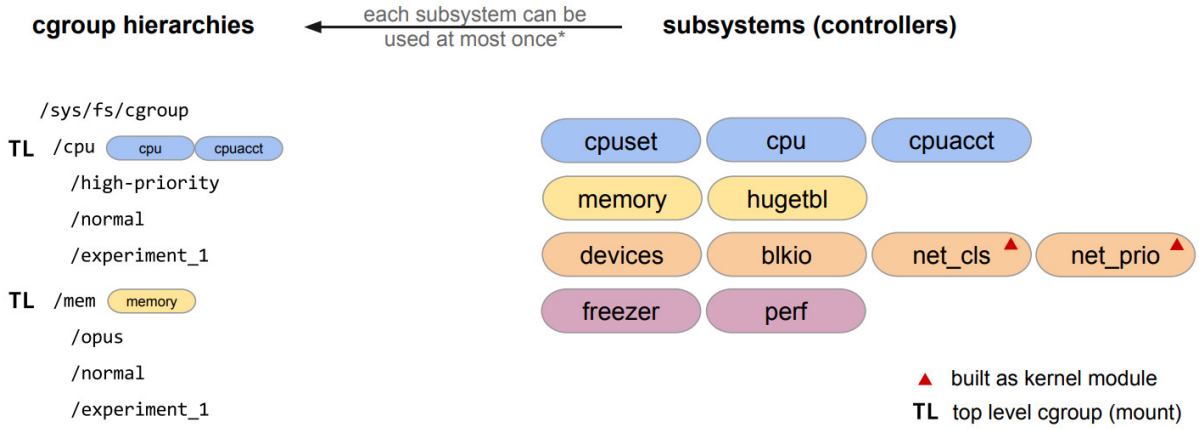


Figure 22: Control Groups: File System Hierarchy [kvs]

Initially, all processes belong to the root control group. Creating a new *cgroup* is being accomplished by creating a directory for the control group under `/sys/fs/cgroup/<controller type>/<cgroup name>`. To configure the newly created *cgroup*, two tasks have to be performed:

1. Adding *cgroup* configuration by creating an attribute file with desired values. For example, to set a memory limit of 100MB for all processes in the control group `mem`, the file `/sys/fs/cgroup/memory/mem/memory.limit_in_bytes` has to be created with content `100000000` [awc]. For each controller, there exist multiple possible attribute files that can be used to configure a control group.

2. Moving a process to a control group by writing the target PID to `/sys/fs/cgroup/memory/mem/cgroup.procs`.

The `cgroup.procs` file mentioned above may be utilized by any process with write access to the respective file to move another process into a control group. However, when a PID is written to the file, an additional check is being performed to determine whether the writing user is privileged or the same user that's the owner of the process that's about to be moved.

Control groups can also be nested by creating sub-directories under already existing *cgroup* folders. It should be noted that changes made to processes and the control group configuration are *not* persistent. After rebooting a machine, the configuration will be lost

in case `cgconfig` or similar mechanisms have not been used to make the changes persistent. Persistence is achieved by re-applying control group configurations upon startup. Re-mapping processes of a specific user to a control group works by using the daemon `cgrulesengd` in combination with a rule set. To remove a *cgroup*, the associated directory has to be removed which renders the control group invisible. As soon as all processes of the *cgroup* terminate or are considered zombies, the control group will finally be removed by the kernel.

Due to the way control groups were originally implemented, many inconsistencies were introduced and the code was rather complex in regard to the offered features [mcg]. Unfortunately, the development of *cgroups* was not coordinated in an optimal manner [mcg] which lead to the problematic first version of control groups. Since Linux `4.5`, the *cgroups* `v2` implementation is available with the intent to replace the original `v1` implementation. However, as of now not all resource controllers of the original implementation are available in the newer version. Because of this, a parallel usage of both versions has to be taken into account in case specific `v1` features are required. Please note that a single controller can not be used in both `v1` and `v2` simultaneously. Some major differences between `v1` and `v2` are listed in the table below:

Table 3: Control Groups: Major Differences Between v1 and v2

v1	**v2**
Multiple *cgroup* hierarchies possible.	Only a single, unified hierarchy is allowed.
Distinguishes threads and processes. Threads of a single process can be mapped into different cgroups.	No control of single threads. Therefore no `tasks` file. Exception: Thread mode.
Processes can be mapped to *cgroups* disregarding whether a group is a leaf node of the hierarchy or not.	No internal processes rule for non-root *cgroups*: A group can not have controllers enabled and have child groups and processes mapped to it at the same time.
The `release_agent` can be used in order to get notified in case a group gets empty.	The events file can be used for the same purpose.
Implements own mount options.	Deprecates all `v1` mount options.
Uses `/proc/cgroups` to keep track of all controllers supported by the kernel.	Uses the `cgroup.controllers` file for the same purpose.

Additionally to the *cgroups* file mentioned above, the following files are present to organize the control group functionality:

- `/sys/kernel/cgroup/delegate`: Since Linux `4.15` it's possible to delegate the management of a control group to an unprivileged user. This file lists all *cgroup* version 2 files that the delegatee should own after a delegation process. Delegating a control group sub-tree is a privileged action. This will be important when discussing unprivileged containers in 4.
- `/sys/kernel/cgroup/features`: This file enables user-space applications to list all control group features the kernel supports.

- `/proc/<PID>/cgroup`: This contains information on all *cgroups* a process resides in. In the listing below the contents of this file are listed as an example. It consists of colon-separated attributes organized in lines:

```
1 root@box :~# cat /proc/$$/cgroup
2 12:cpuset:/
3 11:net_cls,net_prio:/
4 [...]
5 0::/user.slice/user-1000.slice/session-2.scope
```

Listing 31: Control Groups: Parallel Usage of v1 and v2

This shows the parallel usage if version 1 and 2 of the control group feature: For `v1`, the first number of each line represents the hierarchy ID number that can be matched to the `/proc/cgroups` file. For the `v2` implementation, this field always shows a value of zero. The second field represents the associated resource controllers for `v1`, whereas this field is empty for the newer implementation. At last, the mount point relative path to the individual control group path is listed.

3.2.1 Thread Mode

The initial control group implementation allows distributing the threads of a process into individual control groups. At first, this seemed to provide a maximum of flexibility. However, there are cases where this resulted in problems: Threads of a single process are running in the same memory space. When limiting the available memory of one thread, this also affects all other threads that may have other memory restrictions imposed by other *cgroup* memberships. Undefined behavior results as multiple settings are active at the same time for the threads of a single process. Also, when moving a process into a control group after starting it, it's often necessary to move all child threads too. A helper utility for this task is `cgexec` which automatically moves all threads of the created process in the correct groups.

After removing the ability to manage control groups on a per-thread basis in `v2`, some important use-cases could not be fulfilled anymore. This includes the ability of the `cpu` controller to manage threads. Because of that, Linux `4.14` added the *thread mode* [mcg] to relax the restrictions imposed by the transition to `v2`.

This new mode allows to use *threaded sub-trees*, enabling threads of multiple processes to be mapped into different control groups within such a sub-tree. There are now two types of resource controllers:

- Domain: Controllers of this type are not thread aware, meaning that they can only work on process level. All threads of a process have to reside in the same control group. This controller type can not be enabled in threaded sub-trees.
- Threaded: This is the new controller type that allows control group management on a thread-level. It can be used in combination with threaded sub-trees.

A *cgroup* can be transformed in thread mode by inserting the string `threaded` into the `cgroup.type` file.

3.2.2 Network (v1)

This section covers resource controllers: `net_cl` and `net_prio`. Both cause identifiers to be attached to sockets once they are created by a process that's being managed by one of the two controllers. The difference between the two controllers is that `net_prio` assigns an ID that's unique for each control group whereas `net_cl` uses a specified identifier that does not have to be unique for each *cgroup*, allowing flexible tagging of sockets in classes [cgt]. Adding these identifiers allows quick checks to determine whether a socket originates from the same control group or class. This is more efficient than searching in the control group tree, for example using the *cgroup* function `is_descendant()` to perform these checks – especially if this has to be performed regularly.

There are multiple use-cases for these additional socket attributes. Among others, some of them are [cgt]:

- Setting the priority of network packets originating from a specific socket or device by using the network priority set by `net_prio`.
- Using `iptables` in combination with the `net_cl` class identifier to filter and route packets based on the control group membership.
- Scheduling network packets based on class identifiers.

A simple usage example of `net_cls` that drops all IP based traffic for all processes not present in a specific control group can be seen below [nie]:

```
root@box :~# mkdir /sys/fs/cgroup/net_cls # Create mountpoint
root@box :~# mount -t cgroup \ # Mount the controller
    -o net_cls net_cls /sys/fs/cgroup/net_cls
root@box :~# mkdir /sys/fs/cgroup/net_cls/IPAllowed # Create cgroup
root@box :~# echo 0x100001 > \ # Set the fixed class identifier
    /sys/fs/cgroup/net_cls/IPAllowed/net_cls.classid
root@box :~# tc qdisc add dev <interface> root handle 10: htb
root@box :~# tc class add dev <interface> parent 10: classid 10:1 \
        htb rate 40mbit
root@box :~# tc filter add dev <interface> parent 10: protocol ip \
        prio 10 handle 1: cgroup
root@box :~# iptables -A OUTPUT -m cgroup ! \
        --cgroup 0x100001 -j DROP # Disallow IP for all non-members
root@box :~# echo $$ > \ # Add process to cgroup
        /sys/fs/cgroup/net_cls/IPAllowed/cgroup.procs
-- Filtering active --
root@box :~# tc qdisc del dev <interface> root; \ # Revert settings
        tc qdisc add dev <interface> root pfifo
```

Listing 32: Network Control Groups: Using `net_cls`

The `tc` (*Traffic Control*) commands listed above are being used to set up a Queueing-Discipline (QDisc) that uses the fixed control group class to classify the traffic originating from a control group on a network interface by assigning it to a handle called `cgroup`. This filtering is accomplished by using a Hierarchical Token Bucket (HTB) filter. With `iptables` it's then possible to use the `cgroup` handle to add rules for a control group, e.g. allowing network access.

This controller type is an example where child control groups are not automatically affected by the `net_*` controllers, meaning that this setting is not inherited throughout the hierarchy.

3.2.3 Block IO (v1/v2)

As discussed with an example in 2.5.2, the `blkio` (`v1`)/`io` (`v2`) controller is being utilized to enable I/O resource usage policies. The most common use-cases to limit these aspects are:

- Specifying upper bandwidth limits, for example in the `blkio.throttle.read_bps_device` file to specify the maximum bandwidth for a device in bits per second. Alternatively, the `rbps` parameter in conjunction with the `io.max` file is the equivalent for version 2.
- Denying access to a specific device.
- Limiting with proportional time based division: This allows settings weights for various control groups that will be used to prioritize all device accesses when performing I/O operations. The `blkio.weight` file is present for this purpose in *cgroup* `v1` whereas this is configured with `io.weight` in version 2.

Enforcing bandwidth limits is implemented in `blk_throtl_bio` which resides in `block/blk-throttle.c`. This function makes use of `throtl_charge_bio` to ultimately charge for the data volume used in an I/O operation. Depending on the resource usage, an I/O operation can be executed directly or may have to be delayed using a queue to meet the resource limitations. For delayed operations, a dispatcher function will then cause pending operations to be executed using pre-calculated timers in order to throttle requested operations. With `throtl_trim_slice` the required time limiting is calculated, yielding the time slice the operation may be executed in.

To allow or deny accessing a specific device, functions of `security/device_cgroup.c` come to use. When passing *cgroup* configuration strings to the files present in the virtual control group file system, as seen in the previous example (2.5.2), `devcgroup_update_access` parses this information and configures the control group accordingly, e.g. by setting flags indicating whether accessing a device is allowed or denied for processes of a *cgroup*. This builds an exception list as seen in the listing following below. Upon accessing a block device, `__blkdev_get` (`fs/block_dev.c`) is being called which performs access checks prior to allowing access. To perform these checks, `__devcgroup_check_permission` (`security/device_cgroup.c`) is called, resulting in the following checks being performed using the internal exception list:

```
1  // current is the current task_struct
2  dev_cgroup = task_devcgroup(current);
3  if (dev_cgroup->behavior == DEVCG_DEFAULT_ALLOW)
4      // perform checks based on the exception list
5      rc = !match_exception_partial(&dev_cgroup->exceptions,
6          type, major, minor, access);
7  else
8      rc = match_exception(&dev_cgroup->exceptions, type, major,
9          minor, access);
10 if (!rc)
11     return -EPERM; // deny access
```

Listing 33: Control Groups Kernel View: `blkio` Access Checks

The default scheduler for I/O operations in the Linux kernel is the CFQ scheduler [tkc]. It was extended to support the I/O related *cgroup* controllers after control groups have been introduced in the kernel. This makes it possible to account and constrain processes regarding their consumed I/O resources, for example using pre-defined weights. The CFQ I/O scheduler is implemented in `block/cfq-iosched.h` – not to be confused with `kernel/sched/fair.c` where process-related CFQ scheduling is implemented. The kernel structure `cfq_group` maps various settings per *cgroup*-device relationship. This includes applied policies and weights which are considered in order to schedule I/O operations.

3.2.4 Memory (v2)

Processes can be accounted and limited regarding their memory usage. The following types of memory usages are currently being tracked [cfo]:

- Consumed memory in user-space
- Memory usage in kernel-space, as in kernel data structures
- TCP socket buffers

The memory consumed by a control group can be read using the `memory.current` file. One of the most common ways to limit memory usage includes setting the `memory.max` value to define an upper memory limit for all processes residing in a control group.

A *cgroup* is charged for its memory usage when allocating memory. In turn, this accounting gets removed once `free()` or similar mechanisms to free previously allocated memory are being used. When moving a process that has allocated memory in the name of a control group to another group, the *original* group is being charged for the allocated memory since these mappings are not being transferred [cfo].

3.2.5 CPU (v1/v2)

Various controllers exist for version 1 and 2 of the control group implementation to manage CPU utilization. There are three controllers for `v1`:

With `cpu` a control group is supplied with a guaranteed minimum time of CPU utilization. The `cpuset` controller allows specifying a set of processors a process is allowed to be executed on. For the current process this can be examined with `cat /proc/$$/status | grep Cpus` . Changes to this setting propagate to all descendants in the control group hierarchy. Accounting is performed with `cpuacct`, for example the file `cpuacct.usage` gives information on the consumed CPU time of all processes in a control group in nanoseconds.

Control group version 2 allows weight and absolute CPU limiting models with the `cpu` subsystem. In contrast to `v1`, the newer control group version does *not* support real-time processes. Therefore, all real-time processes have to be moved to the root control group first before activating the `cpu v2` subsystem in the *cgroup* tree [cfo].

3.2.6 Freezer (v1)

This control group does not limit or account resource usage – it rather allows *freezing* a process. Freezing a process ultimately stops the execution and suspends it. This allows analyzing the current state of a process with the ability to unfreeze it afterwards and continue the execution without side effects. By creating a checkpoint with the `freezer` subsystem it's also possible to move an entire running process, including its children, to another machine or restart a process from a specific state [cfm].

For this, the virtual file `freezer.state` exists that can receive either `FROZEN` or `THAWED` as input values to freeze and unfreeze a process. This works by walking down the control group hierarchy and marking all descendants of a process with the desired state. Additionally, all processes managed by the affected groups have to be moved in or out of the `freezer` group, depending on the desired freezing state. Freezing itself is done by sending a signal to the affected processes. Also, the `freezer` has to follow all child process of the affected processes that may result from calling `fork` and freeze these as well to prevent freeze escapes [cgt].

3.2.7 Devices (v1)

As already briefly discussed in 2.5.2, this controller type allows to implement access controls for devices. One can use whitelist and blacklist approaches to only block or allow very specific accesses by defining exceptions. Child control groups are forced to have the exact same or a subset of the exception list of the parent. This results in faster checks whether a rule can be added to the exception list because only the list of the child has to checked and *not* the whole group tree. This controller is one of the few that makes use of the hierarchical organization in order to pass configuration information to its child groups.

3.2.8 Kernel View

In the kernel source code, control groups are represented by the `cgroup` structure defined in `linux/cgroup-defs.h`. Every `cgroup` includes a unique ID, starting from the value `1`, always using the smallest value possible. When applying changes to the control group hierarchy, checks have to be performed on a regular basis to determine whether a group is a descendant of another group. To avoid the requirement to traverse in the control group tree, an integer value `level` is present to solve this problem using numerical comparisons.

The logic to initialize a control group is implemented in `cgroup_init` (`kernel/cgroup/cgroup.c`):

```
1  [...]
2  BUG_ON(cgroup_setup_root(&cgrp_dfl_root, 0, 0));
3  [...]
4  for_each_subsys(ss, ssid) {
5      [...]
6      cgroup_init_subsys(ss, false);
7      [...]
8      css_populate_dir(init_css_set.subsys[ssid]);
9      [...]
10 }
11 [...]
12 WARN_ON(sysfs_create_mount_point(fs_kobj, "cgroup"));
13 WARN_ON(register_filesystem(&cgroup_fs_type));
14 WARN_ON(register_filesystem(&cgroup2_fs_type));
15 WARN_ON(!proc_create_single("cgroups", 0, NULL,
16     proc_cgroupstats_show));
17 [...]
```

Listing 34: Control Groups Kernel View: Initializing a Control Group

The function `cgroup_setup_root` initially sets up the control group root which is represented by a `cgroup_root` structure. Internally this includes the `cgroup` structure discussed above. The setup routine is also responsible for creating the `kernfs` – the virtual filesystem exporting the files residing in `/sys/fs/cgroup`. After that, all control group subsystems are being enabled. With `init_and_link_css` called in `cgroup_init_subsys`, pointers to the respective children, siblings and parent nodes are created. The abbreviation `css` stands for *cgroup subsystem state* in this context and is being used to map a specific thread to a set of control groups [cgx]. The function `css_populate_dir` creates a virtual filesystem for each controller in the `kernfs` created before. Finally, the `kernfs` is mounted in `sysfs_create_mount_point`. For each control group version a filesystem is registered and the virtual `/proc/cgroups` file is being created.

There exists a global array of all subsystems, called `cgroup_subsys` which is defined using an `include` directive for `cgroup_subsys.h` as can be seen below. This is the file holding all available controllers supported by the kernel.

```
1 structure cgroup_subsys *cgroup_subsys[] = {
2     #include <linux/cgroup_subsys.h>
3 };
```

Listing 35: Control Groups Kernel View: Control Group Subsystem Array

The controllers are implemented using another kernel structure: `cgroup_subsys`. This structure provides a common interface for all implemented resource controllers. Another common interface for all subsystems is the `cftype_ss` structure which enables all controllers to define own virtual files to export data. For example the `cpuset` controller exports these files:

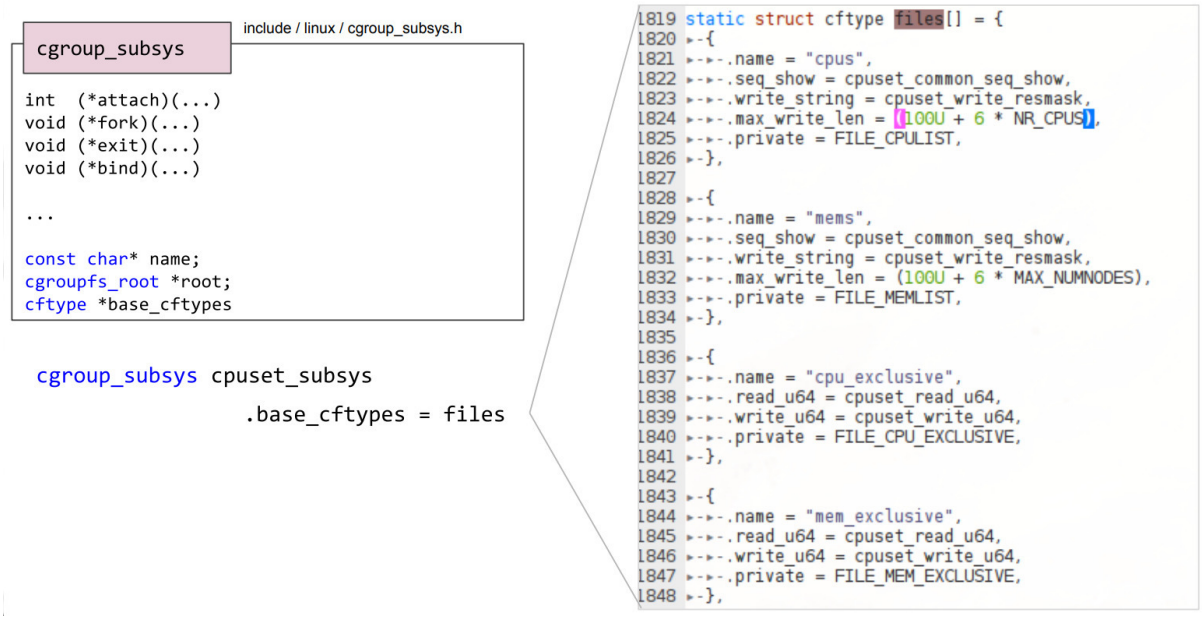


Figure 23: Control Group Kernel View: Exporting Virtual Files [kvs]

Similar to the implementation of namespaces in the kernel, the `task_struct` structure also holds information regarding control groups:

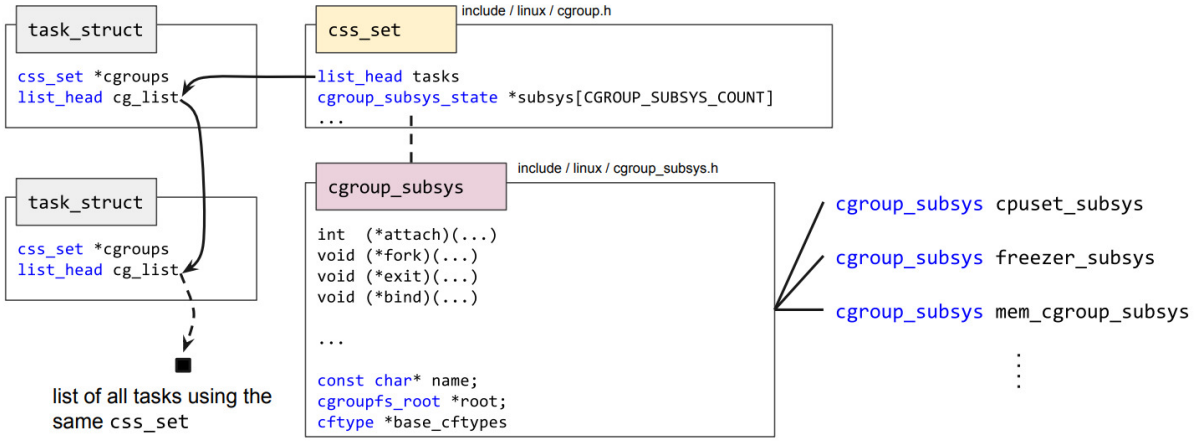


Figure 24: Control Group Kernel View: Kernel Structures (Simplified) [kvs]

As can be observed in the figure above, the `css_set` structure associates a set of control group subsystems to a task. Every task with the same *cgroup* subsystem set has a pointer to the same `css_set`. This is being used to save space in the task structure which effectively speeds up `fork` calls [lcs].

Internally there's a `MxN` relationship between `cgroup`s and `css_set`s. To link the kernel structures efficiently, the following link structure is in place:

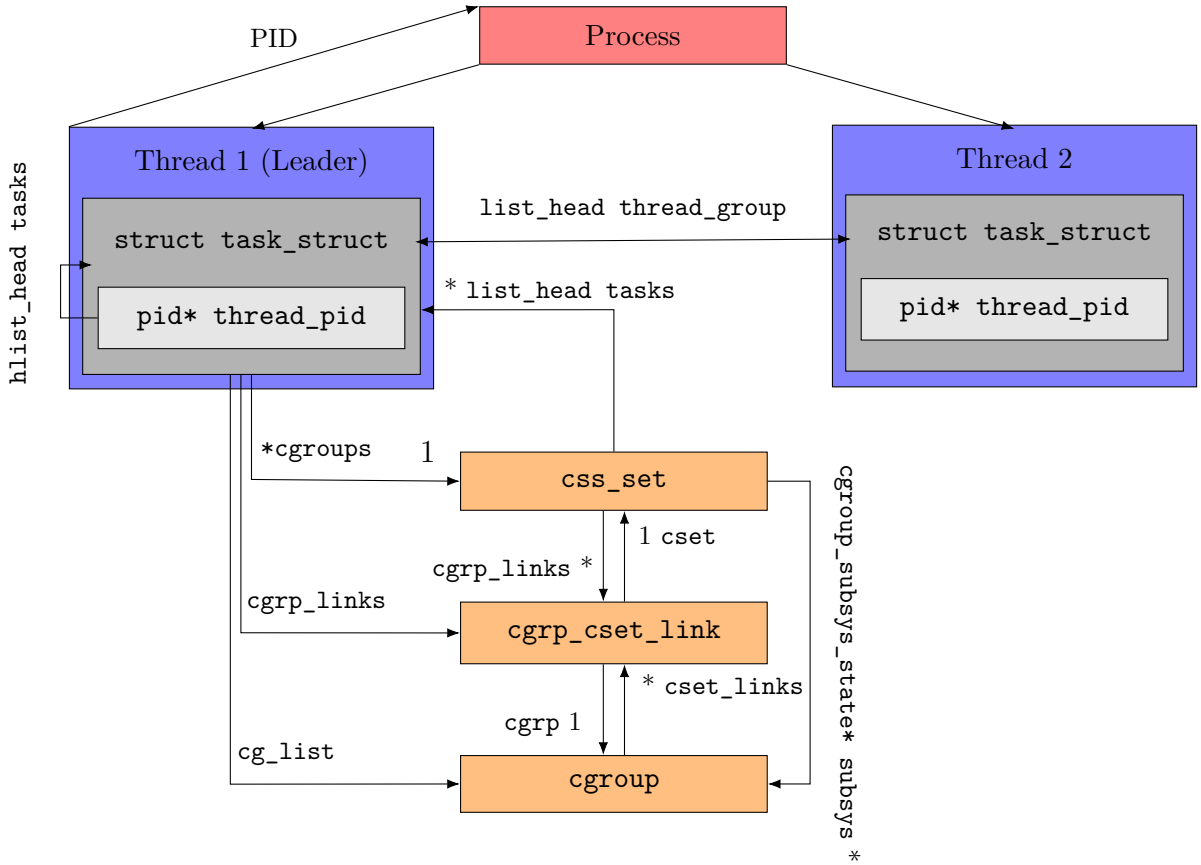


Figure 25: Control Groups Kernel View: Kernel Structures (Simplified)

For each process, there exists exactly one *leader* thread whose thread ID is equal to the PID of the whole process. It's possible that a `css_set` is being linked to multiple control groups because every single task can be present in various *cgroups*. For this reason and to be able to traverse the link structure the other way around, beginning from a `cgroup`, the linking structure `cgrp_cset_link` associates both kernel structures. The labels of the arrows are to be interpreted as Unified Modeling Language (UML) associations. As shown above, there also exist multiple shortcuts in the structure linkage to allow an efficient direct access to associated structures without having to traverse multiple lists. For example the link from `css_set` to `cgroup` bypasses the linking structure in between.

3.2.9 Usage in Containerization

Container engines like LXC and *Docker* support the configuration on control group settings. Similar to the configuration of namespaces, it's possible to supply command line parameters to the *Docker* CLI while configuration files are being used for LXC.

When starting a *Docker* container without any additional control group configuration, a `docker` group is created for each controller type mentioned in this chapter using permissive default values. Additional configuration can then be applied after starting a container using the mechanisms that have already been discussed. Furthermore the command line options allow *cgroup* configuration without the requirement of interacting with the virtual control group filesystem. For example, the amount of CPUs a container may use is being configured by passing a numerical value along with the `--cpus` option. Another convenient feature is the ability to integrate a container into an existing parent control group. Therefore control groups are able to be prepared in order to use it as parent group for a container later on. With the ability to use persistent *cgroups*, containers can be restricted in an automated way upon booting a machine by assigning a parent control group.

3.3 Seccomp

A considerable part of the kernel's attack surface is being exposed by system calls. The *Secure Computing* (*seccomp*) facility provides a way to reduce this attack surface by limiting the set of system calls a process is allowed to use. In case a process gets compromised the risk of any further system damage, for example risk imposed by a privilege escalation attack requiring a specific system call is therefore being reduced. The first version of *seccomp* has been merged into the Linux kernel in 2005 (version `2.6.12`). Starting from that point many additions to this mechanism found their way into the upstream version of the kernel.

3.3.1 Operation Modes

Currently there are two modes of operation [lws]:

1. **Strict Mode** (Mode 1): By making use of a fixed whitelist only the system calls `read`, `write`, `exit` and `sigreturn` are allowed for a process. The usage of any other system call results in `SIGKILL` being sent to the violating process [mps].

2. **Filter Mode** (Mode 2): Arbitrary system call black- and whitelisting can be performed using this mode. Internally a Berkeley Packet Filter (BPF) processes the requested system calls along with their arguments and allows or denies the execution based on prior configuration. Please note that the BPF originally is a construct for network packets to handle packet filtering in kernel-space [lws]. However, the BPF can be generalized in a way that it's also applicable for system calls, making it a useful utility in this scenario that also saves from introducing more logic into the kernel code. The filtering can be configured using the corresponding BPF language that defines the BPF program used to verify whether a call is allowed or not. The patch implementing this *seccomp* mode was published 2012 [lcd] [lcp].

The current *seccomp* mode of a process can be examined using the file `/proc/<PID>/status` and by making use of the *Seccomp* entry. A zero value indicates that *seccomp* is disabled for this process.

3.3.2 Internals

Installed filters are evaluated for each system call an associated process is requesting to use. This also introduces a performance impact for the execution of system calls. For example, a simple application that issues the `getppid` system call is slowed down by up to 25% because of the additional *seccomp* checks in place [lws]. Depending on the system call filter that's in place, additional filter rules can be added. Always the lastly added rule is being evaluated first. Of course, adding more rules also impacts the performance.

The return value of a *seccomp* filter determines the process behavior in regard to the requested system call and applied filter. Among others, it's possible to return:

- `SECCOMP_RET_KILL_PROCESS`: This kills the entire process among all its threads.
- `SECCOMP_RET_KILL_THREAD`: This only kills the thread that requested the execution of a blocked system call. All other threads of the same process are not killed.
- `SECCOMP_RET_ALLOW`: To allow the system call.

The wrapper of the `seccomp` system call has the following prototype:

```
1 int seccomp(unsigned int operation, unsigned int flags, void *args)
```

Listing 36: Seccomp: System Call Prototype

Among others, the `operation` parameter accepts the values `SECCOMP_SET_MODE_STRICT` or `SECCOMP_SET_MODE_FILTER` in order to activate one of the two operation modes listed above. By providing specific flags it's possible to synchronize all threads of a process in order to apply the *seccomp* to all threads properly. The last parameter, `args`, accepts pointers to `sock_fprog` structures holding the BPF configuration to apply in their `filter` value of type `sock_filter` [lsf].

While it's possible to write the BPF filter code manually in the corresponding language it's more convenient to use the programming library `libseccomp` for this task:

```
#include <seccomp.h>
[...]
// Create custom seccomp filter
scmp_filter_ctx ctx;
// Default action is allow --> blacklist
ctx = seccomp_init(SCMP_ACT_ALLOW);
// Setup blacklist
seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(clone), 0);
// build and load the filter
seccomp_load(ctx);
[...]
```

Listing 37: Seccomp: `libseccomp` Usage Example

The listing above defines a blacklist in order to deny using the `clone` system call.

Applied filters are being preserved across calls to `fork`, `exec` and `execve` and can not be removed. In order to apply a filter one of the following points has to be true:

- The `CAP_SYS_ADMIN` capability is available in the current user namespace.
- The `PR_NO_NEW_PRIVS` flag has to be set using `prctl`. This can also be set by a process itself. By setting this flag it ensures that a process can not gain any new privileges, e.g. through `set{u,g}id` and file capabilities. Therefore, calls to change the UID/GID no longer change the respective value and file capabilities do not add new values to the permitted set [nnp]. Once this per-thread flag is being set it will be inherited to all children and can not be unset afterwards. By making this flag being set mandatory it's assured that adding new filters does not result in privilege escalation. Consider the following scenario: The `setuid` system call is being used with a non-zero value to drop privileges. A malicious filter may prevent `setuid` from being executed and effectively causes the elevated privileges to be preserved for an otherwise unprivileged process.

Please note that the filtering takes place using the system call number as can be seen in the kernel structure [lss] listed below. It's being used to store the execution state information:

```
struct seccomp_data {
    int nr; // system call number
    __u32 arch; // system call convention (architecture)
    __u64 instruction_pointer; // at syscall execution
    __u64 args[6]; // syscall arguments
};
```

Listing 38: Seccomp: `seccomp_data` Structure

This results in BPF filters being platform dependent because it's possible to encounter different system call numbers in other Unix systems.

Please note that there exist various pitfalls [mps]: If for instance calls to the `fork libc` wrapper function must be blocked, it's not correct to block the `fork` system call since `clone` is being invoked instead (see 3.1.2). Also, one wrapper function on varying architectures can employ a varied set of system calls which may break the filtering.

The following figure illustrates the kernel structure linkage for the *seccomp* facility:

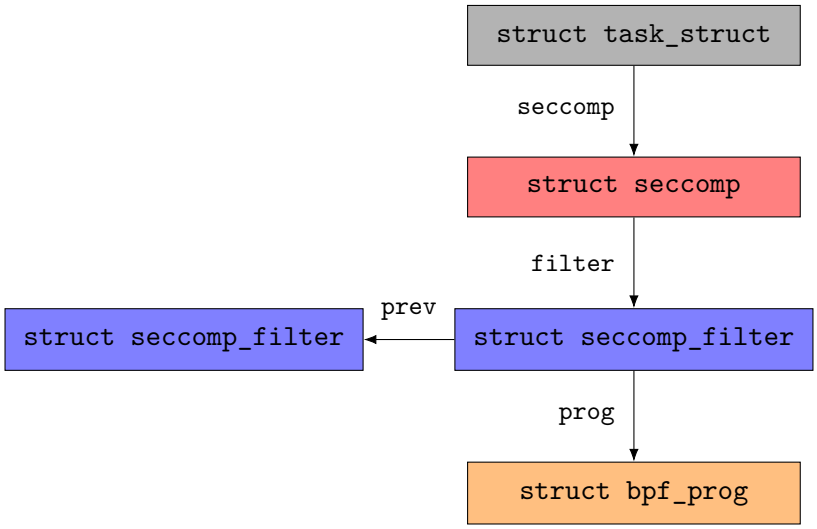


Figure 26: Seccomp: Kernel Structures (Simplified)

Each task is connected to a `seccomp` structure in order to perform the system call filtering on a task level. Each `seccomp` structure is tied to a linked list of filters. Each filter uses a `bpf_prog` structure that holds the associated BPF program.

Seccomp is commonly used to provide additional security for services and fault injection tests. In the latter scenario it's being used to simulate failing system calls to test the behavior of applications under certain circumstances. The *OpenSSH* project is using *seccomp* to harden the server process by filtering all unneeded system calls [sss].

Determining a minimal set containing all system calls an application may require to operate is not trivial. In general, there are two ways to accomplish this:

1. Runtime analysis, for example using `strace`. For this to be successful every possible code path of an applications has to be reached, requiring a high coverage when determining a minimal set of system calls using unit tests or similar approaches.

2. Static analysis, for example by making use of the information that's generated by compilers on build-time. An example of this is using the *Go* compiler that can be used to maintain a list of all system calls used by the compiled application [jfk].

The methodologies described above can be problematic in certain scenarios. For example invoking another binary from C code using the `system` function can not be covered when analyzing one specific binary.

3.3.3 Usage in Containerization

Similar to reducing the set of available capabilities described in 3.1.11, limiting the usage of system calls can also be used to prevent containerized processes from accessing or modifying resources governed by the host system. Additionally, potentially dangerous or vulnerable system calls are prevented from being employed to limit the kernel's attack surface from inside a container. Fewer permitted system calls result in a lower risk of an attacker being able to successfully use a kernel exploit to leverage a container breakout.

The *Docker* project ships with its own default *seccomp* profile [dps] that's based on a whitelist. The goal of using this profile is to have a sane default set of permitted system calls while providing a moderate security level without breaking existing containers and applications at large scale [dsm]. By default it also protects from using system calls that can affect resources that are not protected by namespaces like the kernel keyring [jfr]. After blocking the `keyring` and any related system call it's therefore not possible to access the host's keyring data from inside a container. While it's recommended to use the default *Docker* profile it's also possible to provide a custom profile to limit the set of available system calls even further or to add specific calls to be allowed instead.

3.4 Linux Security Modules

An additional level of security for container environments can be provided with kernel extensions called Linux Security Module (LSM). These modules are commonly used to implement a Mandatory Access Control (MAC) mechanism [lsm] that enforces certain security policies. A LSM defines multiple hooks that may run before accessing resources, enabling the kernel to manage access according to a pre-defined policy [lsh]. In the context of containerization the goal is to protect the host from potentially malicious processes inside containers and to protect containers from one another. Two of these modules are *SELinux* and *AppArmor*.

3.4.1 SELinux

This LSM works by attaching a *label* to each managed resource [sel]. Next to files and directories these resources can also be network ports and devices. The container engine *Docker* is being shipped with an associated *SELinux* default policy that can be activated manually. Among others, the policy blocks write access to files under `/etc` and `/usr` [sem]. One can verify this by creating a file in one of the directories and mounting it in a container without the read-only volume option. Any write access to this mounted file from the container are then blocked by *SELinux* while read access remains possible.

Docker provides its own volume options that may relabel files and directories:

- `z` (lowercase): Relabels content in a way that allows sharing it across *all* containers
- `Z`: Relabels content as private so that it can only be mounted into one specific container

The latter option assigns a label to a resource that's unique for the specific container and prevents any access to it from other containerized processes.

3.4.2 AppArmor

In contrast to *SELinux* that's working with labels, *AppArmor* is path based. This means that granting or denying access to paths are managed via a policy file that whitelists or blacklists full paths, optionally using regular expressions.

When activated, the following *AppArmor* default profile is being used by *Docker* to prevent specific write operations to files under `/proc` [aad]:

```
1 [...]
2 deny @{PROC}/* w,   # deny write for all files directly in /proc
3 # (not in a subdir)
4 # deny write to files not in /proc/<number>/** or /proc/sys/**
5 deny @{PROC}/{[^1-9],[^1-9][^0-9],[^1-9s][^0-9y][^0-9s], \
6     [^1-9][^0-9][^0-9][^0-9]*}/** w,
7 [...]
```

Listing 39: AppArmor: Docker Default Profile Template

These rules were helpful to prevent write operations to `/proc/acpi/ibm/bluetooth` and `/proc/acpi/ibm/kbdlight` from inside a container that could have been used to enable Bluetooth or manipulate the screen brightness settings on the host from a containerized environment [dic]. This issue was effectively caused by not adding `/proc/acpi` to the masked paths (similar to the issue described in 3.1.6) in the OCI specification. As a last resort, *AppArmor* prevented this issue using this default policy.

3.5 Combination of Kernel Primitives

In containerization, layering of security primitives is a common way to prevent attacks – even theoretical ones. Container engines like *Docker* can be used in combination with existing security mechanisms like *AppArmor*. The goal is to prevent container escapes that may enable attackers to execute arbitrary commands on the host system or leak sensitive information by making it necessary to bypass multiple security mechanisms at the same time for a single prohibited operation. This works by combining the kernel primitives that have been described in the previous sections.

A few examples of the layering of security primitives are:

- `mount` operations can be prohibited using *AppArmor* profiles [jfk]. Also, the Linux kernel enforces that the capability `CAP_SYS_ADMIN` has to be present in order to use the `mount` system call. On top of that, *seccomp* is able to block these calls by blacklisting `mount`. The default *Docker* profile for *seccomp* does this by default.
- The same *seccomp* profile blocks `clone` calls to prevent creating new namespaces in which otherwise denied operations could potentially be performed. To create new namespaces `CAP_SYS_ADMIN` is required additionally, except for user namespaces. The same applies to `unshare`.

- Unprivileged processes are required to have the *No New Privileges* flag being set to prevent employing malicious *seccomp* filters (see 3.3.2) that may allow elevating privileges.

To demonstrate that this layering is in fact secure and reliable, the `http://contained.af` website provides a shell with the `root` user logged in that's available in public on the internet. The terminal process is restricted using the described security primitives to, among other things, deny using raw sockets. *If* it's possible for a site visitor to modify the system in a way that enables the usage of the `ping` command, it suffices as a proof that privilege escalations or container escapes are in fact possible from within a container. As of now, it has not been demonstrated that this is in fact possible.

4 Unprivileged Containers

In the context of containers, the term *unprivileged* or *rootless* refers to containers being created and managed entirely with the permissions of an unprivileged user – e.g. without elevating privileges with `sudo` or a user being a member of a group that's allowed to execute certain commands with higher privileges. Please note that this does not refer to the user that's being used inside of a container to run services because the actual part that requires elevated privileges on the host system – creating a container along with namespace and control group configuration – has already finished running at this point.

4.1 Current State

The fewer privileges a process owns while still remaining operable, the better it is in terms of security. The same is true for containers: To create a running container, `root` privileges are required at certain points. This is why for instance the *Docker* daemon `dockerd` is running as `root` user. Also, LXC requires SUID binaries or the `lxd` daemon [ldu] running with `root` privileges. Because of this design, there's a privileged process running, acting as a *permission gateway*.

From a security perspective this requirement is not desired as exploiting the privileged component may lead to privilege escalation issues. Also, corporate restrictions may apply that prevent people from being able to execute commands with elevated privileges in certain environments. On top of that, build servers should be able to build container images in an unprivileged way to prevent potential compromises of the infrastructure. Therefore the usage of containerization is being hindered under certain circumstances solely because of the required permissions.

There are two reasons for that with the first one being the networking setup. By default, a new network namespace only possesses a loopback interface pointing to its own network namespace (see 3.1.8). In order to link network namespaces, in this scenario the host's network namespace and the container's namespace, privileged actions are necessary that require `CAP_NET_ADMIN` in both the host and the container network namespace [asn]. There exist certain possibilities [rlc] that either involve kernel patches or mechanisms to create a network bridge, for example using `tap` devices. Currently network namespaces are being omitted for unprivileged containers. Unprivileged containers that make of us privileged helpers can use an SUID binary or a privileged daemon prior to running the unprivileged container itself [gdn] – however the disadvantages already described have to be taken into account.

With user namespaces being available, creating, setting up and joining namespaces with user privileges is not an issue. However, creating control groups with user privileges is still an ongoing issue [asu]. By default, the control group directories are owned by the `root` user with permissions that deny write access from other users. Therefore no sub control group can be created or joined with user privileges because this behavior is not part of the permission model of control groups [asn]. The usage of delegated sub-trees, as described in 3.2, would be ideal to use in the context of container managers but the action of delegating a sub-tree itself is privileged. On top of that, the current implementation

of the control group namespace (see 3.1.10) only isolates the *cgroup* virtual directory structure and does not allow unprivileged modifications to the hierarchy.

A proposed kernel patch [cgk] has been created to solve this issue: By unsharing the control group namespace a new delegated subtree is being created in every control group the current process is residing in. After moving the process to the newly created sub-groups, it's free to alter the *cgroup* configuration. Because this is directly implemented in the kernel, no special privileges are required to perform this action. Also, this patch uses the existing control group namespace and does not implement new mechanisms to achieve the same goal.

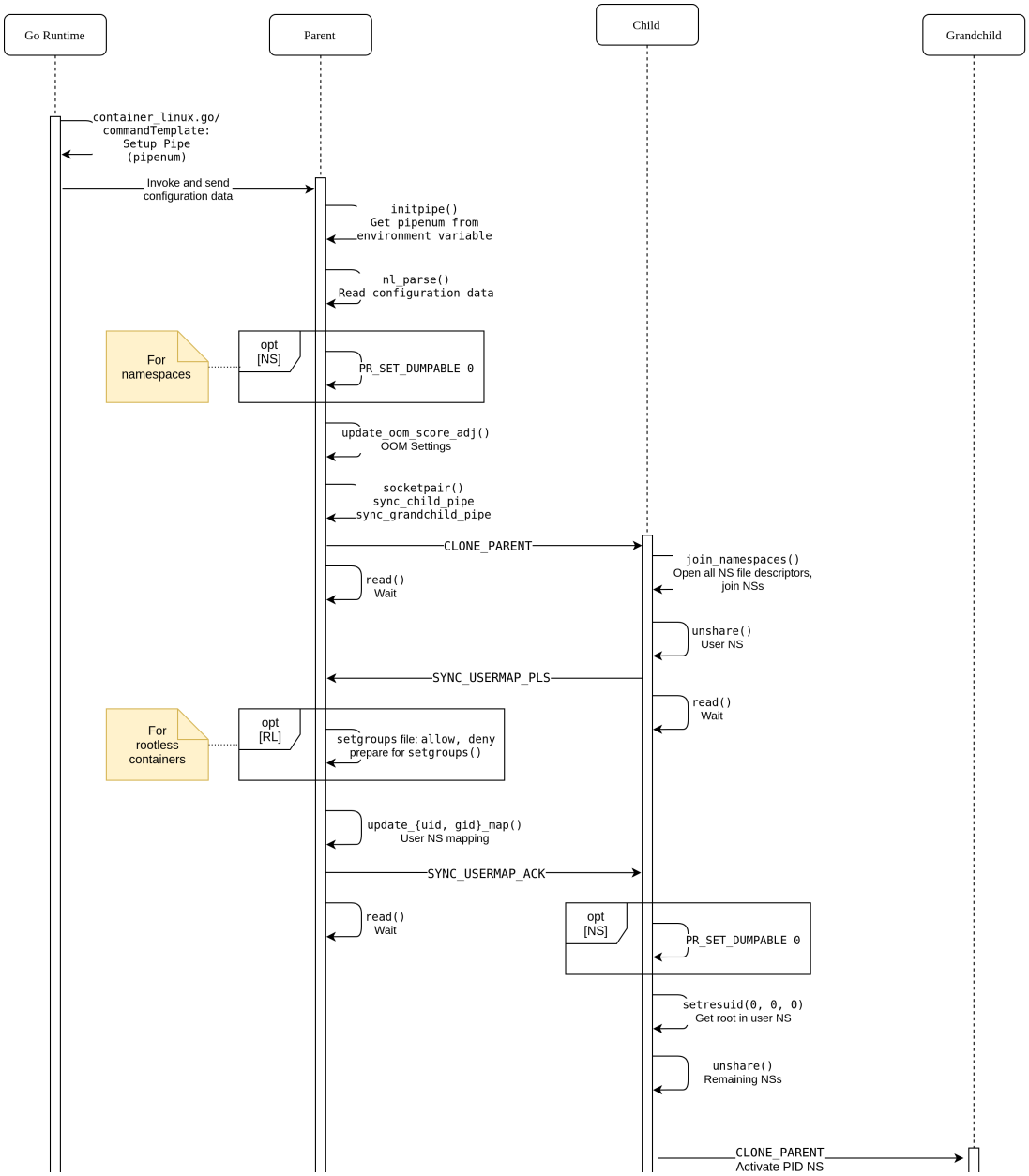
However, the current maintainer of the control group system questioned whether the proposed patch would lead to issues with the current control group subsystem [ckf]. One major problem is that various control group `v1` controllers do not expect unprivileged users to be able to modify the *cgroup* structure and configuration. For example, the `v1` controllers `net_cl` and `net_prio` (see 3.2.2) do not strictly act according to the control group hierarchy, hence it may be possible to escape constraints imposed by ancestor groups by being able to write to a *cgroup* subdirectory of an ancestor [ckh]. This and various other reasons lead to the patch not being merged into the Linux kernel [ckg]. Currently it's not known whether there will be further attempts to implement a mechanism of this kind into the kernel that's conform to the existing *cgroup* system and its controllers.

There exist workarounds [wmp] that involve `chown` calls to create delegated sub-trees prior to spawning unprivileged container daemons. This partially mitigates the existing issues with privileged control groups in case no previous user input is required that may affect these calls. Another possibility is to omit using control groups at all. However, then only a limited set of possibilities regarding resource usage constraining and monitoring are given and no fair resource usage across all containers is guaranteed.

While this can be an issue for running containers, building images without special control group and network namespace settings may be feasible for scenarios that involve CI/CD pipelines and build servers. With an unprivileged container image builder, such as `img` [img], it's possible to build images on a central node without requiring any special privileges. Internally this uses `runC` in unprivileged mode to execute build instructions. As stated in 2.7.1, `runC` is also part of *Docker* and is responsible for the container creation process. How this works in detail in regard to unprivileged containers is described in the following section.

4.2 Creation of Unprivileged Containers with runC

The `runC` project [rcg] packages `libcontainer` that includes a *Go* package called `nsenter` [rns]. A part of the package, namely `nsexec.c`, is written in the C programming language that's being imported and executed before the respective *Go* code runs in order to prepare the environment for a containerized process. This helper also supports creating unprivileged containers. The following sequence diagram illustrates the necessary steps to create an unprivileged container in `runC 1.0-rc6`:


Go Runtime
Parent
Child
Grandchild
container_linux.go/
commandTemplate:
Setup Pipe
(pipenum)
Invoke and send
configuration data
initpipe()
Get pipenum from
environment variable
nl_parse()
Read configuration data
For
namespaces
opt
[NS]
PR_SET_DUMPABLE 0
update_oom_score_adj()
OOM Settings
socketpair()
sync_child_pipe
sync_grandchild_pipe
CLONE_PARENT
join_namespaces()
Open all NS file descriptors,
join NSs
read()
Wait
unshare()
User NS
SYNC_USERMAP_PLS
read()
Wait
For
rootless
containers
opt
[RL]
setgroups file: allow, deny
prepare for setgroups()
update_{uid, gid}_map()
User NS mapping
SYNC_USERMAP_ACK
read()
Wait
opt
[NS]
PR_SET_DUMPABLE 0
setresuid(0, 0, 0)
Get root in user NS
unshare()
Remaining NSs
CLONE_PARENT
Activate PID NS

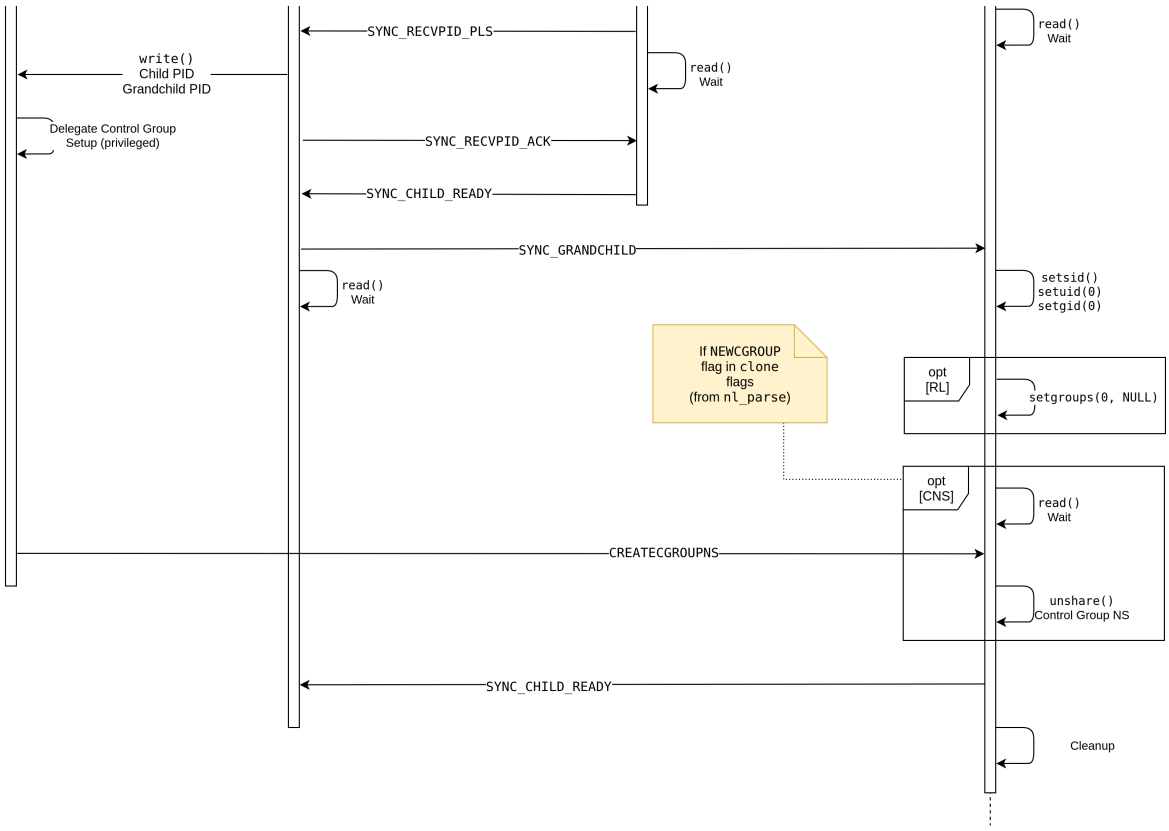


Figure 27: Unprivileged Containers: runC `nsenter` Invocation

As can be seen above, three newly created processes are necessary in order to setup a containerized process that's using the appropriate namespace settings. The process that's creating the `Parent` process is the process that's executing the code of the *Go* runtime. This runtime offers functionality to manage the container lifecycle and can be executed by an unprivileged user. It passes configuration data to its child process using environment variables and netlink messages.

By using the `PR_SET_DUMPABLE` `prctl` operation, the `Parent` and `Child` processes are being protected from being traced with `ptrace`. The main use-case for the *dumpable* flag is to control whether core-dumps are being generated for a specific process upon receiving certain signals [srp]. For containers the second effect of this flag is important: If the flag's value is `0`, no other process can attach to this process using `ptrace`. A `runC` vulnerability [cva] illustrated the risk imposed by not making the `runc init` process non-dumpable: The first process inside of a `runC` container is the parent of all other processes residing in the same container. Upon calling `runc exec` from the host to execute a command in a container, the containerized `runc init` process is able to `ptrace` the process spawned by `runc exec`. This allows the `runc init` process to access file descriptors pointing to locations on the host while `runc exec` is setting up the process that's then executing the user specified command. As long as this setup process is running, it's also possible to use `/proc/<PID>/exe` to rewrite the `runC` binary of the host to execute commands on the host from a container [cme]. The value of the *dumpable* flag is not being inherited across child processes, hence child processes of `runc init` can be traced. This results in the ability to trace all containerized processes, e.g. for debugging purposes. However, there

exists one slight disadvantage: `runC` itself can not be traced using `ptrace` because of the *dumpable* setting. This is a trade-off between protecting the host system and the ability to diagnose bugs via tracing.

To properly synchronize the state of all child processes, a socket pair is in use. Because various configuration aspects can only be set by a process itself or only by a parent process, it's therefore required to notify other process that the correct configuration is present at a given time. This allows a process to continue its execution that's depending on and waiting for further configuration.

An example of this is setting the correct UID/GID mappings of a user namespace. The first chid process, namely `Child` in the diagram above, is requesting the parent to apply the correct mappings (`SYNC_USERMAP_PLS`) in order to use a UID/GID value of zero and the thereto related privileges in its new user namespace. As this mapping can not be set by a process itself, the `Parent` process sets the correct values and notifies its child (`SYNC_USERMAP_ACK`) after the work is done. The `Grandchild` process will then also be executed with the same UID because it's residing in the same user namespace as `Child`.

The PID of both the `Child` and `Grandchild` processes are being reported to the *Go* runtime that manages these processes. After `Grandchild` has finished its setup routine, the created namespace setup can then be used by the runtime. The purpose of `Grandchild` is properly joining the new PID namespace. As stated in 3.1.7, the value of the PID of a process as reported by the `getpid` system call must not change during its lifecycle. For this reason, PID namespaces can only be joined at the time of process creation. This results in the namespace setup not being complete until the `Grandchild` process is created and finished with its own setup routine.

At the time of writing this thesis, the setup of control groups, respectively creating a delegated sub-tree, is a privileged action. Therefore the *Go* runtime delegates this task to a privileged external component that may be a SUID binary or a daemon running with the required permissions.

For unprivileged containers its common practice to issue a call to `setgroups(0, NULL)` to drop the list of currently available supplementary groups. These groups can be observed by listing the contents of `/etc/group` and are used to grant additional privileges by making use of group membership [lwg]. This is a part of the mechanisms used to drop privileges for a containerized process. By default calls to `setgroups` are blocked until an explicit GID mapping from the parent user namespace to the current namespace has been established. This prevents privilege escalation vulnerabilities [cvg] that may occur by unprivileged users being able to drop arbitrary supplementary groups.

The kernel supports denying access to resources like files by making use of an Access Control List (ACL). In case accessing a file is denied for a specific supplementary group, an unprivileged user could elevate its privileges by dropping this group from its credentials with `setgroups` and access this blocked file. As an additional security measure, the string `deny` is written to `/proc/<PID>/setgroups` by the `Parent` process when spawning an unprivileged container to block the `setgroups` system call entirely. GID re-mappings are denied by the kernel for unprivileged processes until `setgroups` is disabled using the described measure.

As the `Grandchild` process is the last one out of all child processes to exist, it's responsible

for the cleanup process. This includes the socket pairs and data structures used to pass configuration data from the *Go* runtime to `nsexec`.

Among others, the following limitations have to be addressed that make the overall process more complex:

- To join the respective namespaces in `Child`, the file descriptors for all namespaces have to be opened first because after joining a `mnt` namespace the namespace paths might be no longer be accessible. Additionally, the first child has to use `setns` when joining namespaces to make sure the reported PID values are in the scope of the right PID namespace.
- The Out Of Memory (OOM) settings have to be adjusted before setting `PR_SET_DUMPABLE` to `0` because otherwise `/proc/self/oom_score_adj` would not be writable anymore.
- Unsharing all namespaces at once may cause issues for older kernel versions. For example various versions do not handle `CLONE_NEWUSER` first when passing multiple flags at once.
- Unsharing the user namespace before calling `clone` causes several issues: Since `unshare` drops all capabilities in the old namespace, only *one* UID/GID mapping can be used because of a kernel limitation.

As described throughout this chapter, the implementation of unprivileged containers is mostly hindered by the current control group implementation and its security restrictions. Applying appropriate kernel patches regarding this aspect is still an ongoing topic – perhaps implementing unprivileged containers that can use all available containerization features will be possible in the future.

5 Survey: Utilization of systemd Isolation Directives

Services managed by *systemd* are able to be executed in an isolated environment, as discussed briefly in 2.7.4. To configure aspects of these environments unit files come to use. Among others, several of the available unit directives can be used to apply settings concerning capabilities, namespaces and control groups. This chapter gives an overview of the usage extent in terms of incorporating the available isolation directives into services offered in *Ubuntu* packages.

To perform this task, two separate software projects have been created:

- Package downloader and parser: With the `apt-file` utility it's possible to gather a list of all software packages that contain a *systemd* unit file. These files are identified using the file extensions in combination with basic checks of the file contents. Every identified package is being downloaded with `apt-get` and extracted afterwards. Since `apt-file` also yields the full path of the unit files in every package, collecting the relevant files of a package is as simple as following the respective paths in the extracted software packages. To perform efficient searches and analysis procedures, every unit file is imported into an *Elasticsearch* collection. This Not Only SQL (NoSQL) database offers an API that's accessible using `curl` and Python.
- Analyzer: With the combination of the `elasticsearch` and `matplotlib` Python modules, statistics are being built on-the-fly. This makes use of the *Elasticsearch* API and its *Painless* scripting support in queries. Therefore, complex statistics can often be built using only a single query. All unit files that are marked as internal *systemd* tests have been excluded from the analysis.

At the time of writing, 2146 out of all *Ubuntu* packages contained *systemd* unit files. Out of these, 181 (less than 9 percent) use one or more isolation directives covered in the generated statistics. A total of 99 packages include two or more directives at once (less than 5 percent). Every covered isolation directive that's being used at least once is listed in the next diagram:

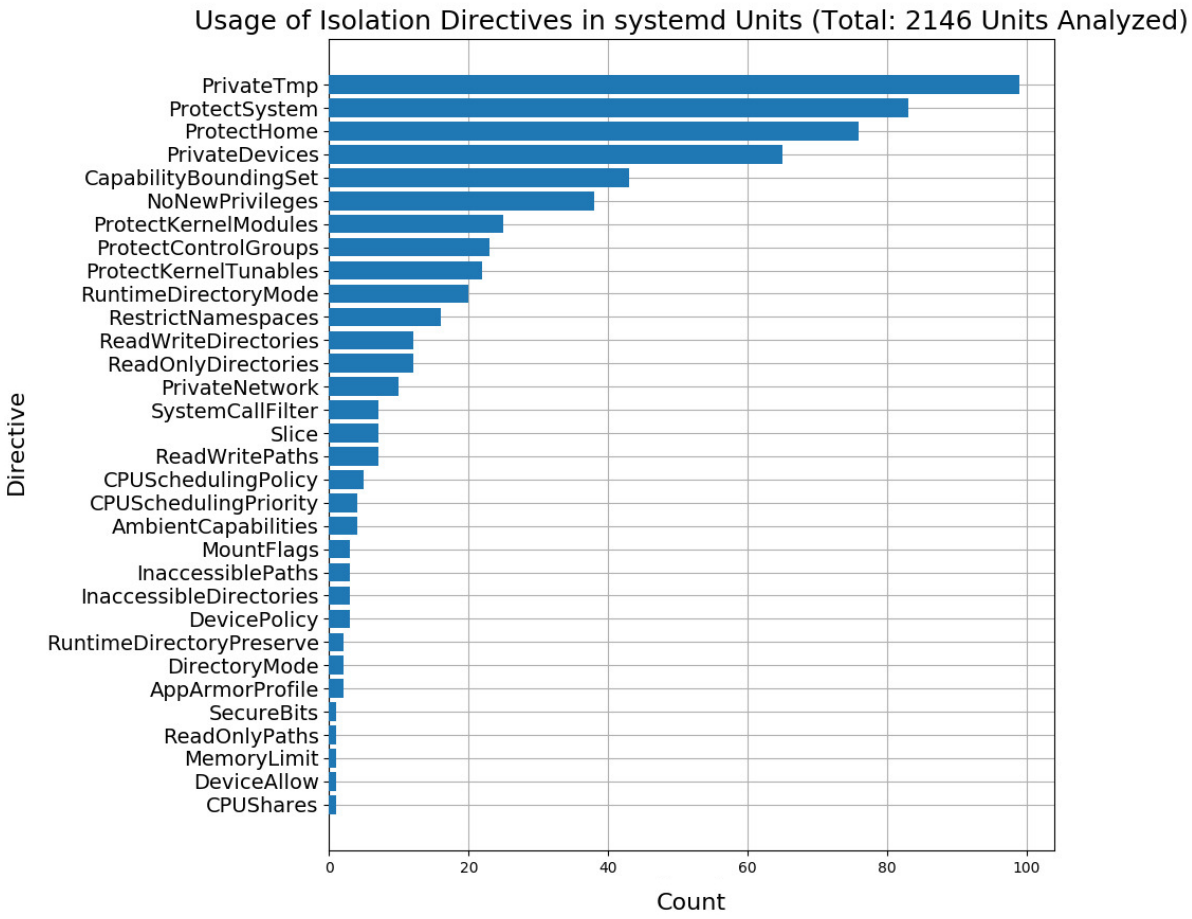


Figure 28: Statistic: Usage of Isolation Directives in *systemd* Units

Note: A list of all covered isolation directives is shown in the appendix (8).

A first conclusion is that most directives are being used rarely or not at all. Less than 50 units use directives regarding limiting capabilities and elevating privileges. The most used option, `PrivateTmp`, is present 103 times. This directive, along with several others, could potentially be enabled for a majority of all units. For instance isolating the temporary folder only produces conflicts in case data is being shared with other processes using this folder. In every other case enabling this options would most likely not interfere with the program flow and should therefore be considered for security purposes. Similar directives can be used to protect system folders, home directories and devices. It should be evaluated whether enabling these options is possible, considering many `systemd` services run with elevated privileges. Also, the *systemd* developers recommend activating these directives for long-running services unless there exist justifiable reasons against their usage [sde].

Less than 20 units use options for namespaces, control groups and system call filtering with *seccomp*. While it may not be possible to provide a default control group setting that's useful for every use-case, applying additional namespace and *seccomp* configuration may be more feasible. Especially since the developers of an application also maintain the corresponding unit file there's a possibility that generating a minimal *seccomp* whitelist for an application is in fact viable. The `PrivateUsers` directive that manages the creation of a separate user namespace for a service is not used at all by any unit – however it may

be possible to lower the risk imposed by a potentially vulnerable service by applying this kind of configuration.

Out of 46 units that are using the `CapabilityBoundingSet` directive, 24 specify the `CAP_NET_BIND_SERVICE` capability. This is being used to allow services running as unprivileged users to bind to arbitrary ports:

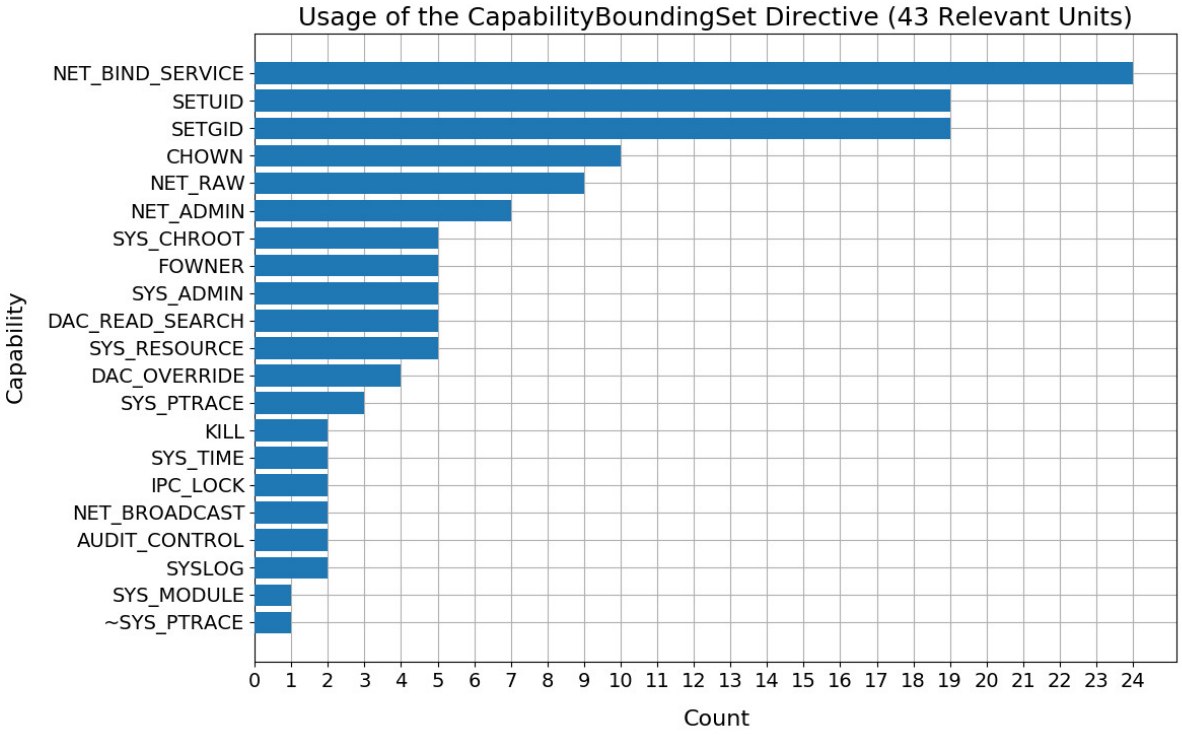


Figure 29: Statistic: Usage of the CapabilityBoundingSet Directive in *systemd* Units

Please note that the tilde symbol can be used to specify a list of negated capabilities. For example, `~CAP_CHOWN` puts all capabilities except `CAP_CHOWN` into the capability bounding set, as shown in the diagram above. Only five units explicitly require `CAP_SYS_ADMIN` in their bounding set. However, this does not mean that the respective service is not restricted from performing actions governed by a capability since it may also be executed as `root` user.

Seven unit files include a directive to perform system call filtering using *seccomp*:

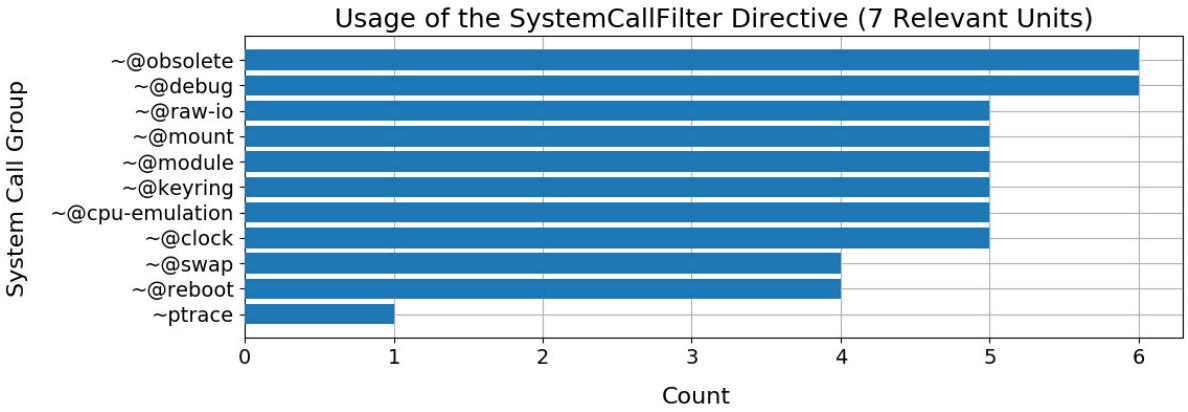


Figure 30: Statistic: Usage of the SystemCallFilter Directive in systemd Units

The parameters for this directive are system call *groups*. As seen in [sde], all available system calls are grouped by their purpose and can be blacklisted or whitelisted on a group-basis. It's noticeable that all groups are used in the negated form. This means that the directive does in fact provide a blacklist instead of a whitelist which would be the default behavior.

For various services the choice of blocked system call groups seems arbitrary and incomplete in detail. For example, the `dbus-org.freedesktop.network1.service` unit file is executing a networking service as `systemd-network` user with extended capabilities regarding networking. The easiest way to provide a minimal system call filter would be to whitelist `@network-io` instead of blacklisting a list of selected system call groups. This blocks access to all unnecessary system calls and provide a certain degree of maintainability at the same time.

The few services that use some of the presented isolation directives may not use them to their full potential. Considering the small amount of units that are using these directives at all, it serves as an adequate starting point.

6 Challenges

During the work on this thesis several challenges had to be met. This chapter discusses these and, if applicable, describes possible solutions.

The most significant challenge was that the kernel code is subject to regular changes. This becomes manifest in two challenges:

- It's not guaranteed that the current state of the kernel code regarding namespaces and control groups will stay in the same state for a longer period of time. Implementation details can change or be replaced entirely. This means that aspects described in this thesis are outdated as soon as significant parts of the kernel code are changed.
- There are several documentation sources for the Linux kernel code. It's not always evident whether information is still applicable or outdated. This has to be checked manually by examining the code and matching the current code and used information. This is one of the reasons why suitable documentation is not available for every topic. An example of this is the *cgroup* `freezer` subsystem for which no practical usage information has been found.

The aspects above also affect the control group documentation for version 1 and 2. It was not always clear which aspect, limitation or implementation detail corresponds to which *cgroup* version. This results in the information having to be verified using the kernel code first. As not every detail of the Linux kernel is documented, much information had to be retrieved by reading the code itself.

The progress of implementing unprivileged containers is well documented. However, certain implementation details and the idea behind them were found to be not documented at all. This was partially solved by contacting the developers themselves and partially by analyzing the related code manually. By creating a sequence diagram the process has been illustrated to get a better understanding of the internals.

7 Conclusion

Profiting from efficient and cost-saving computing environments becomes possible by deploying Linux containers. As of today they are being adopted widely across the industry and existing VM IT landscapes are being transformed into lightweight container deployments. A few of the main use-cases are on-demand scaling of the available infrastructure and optimizing the development workflow, as well as saving cost.

The Linux kernel does not implement containers all by itself. It rather offers several features that can be combined to form a construct that's commonly referred to as a *container*. The two main features in use are namespaces and control groups.

Namespaces define the view on a system's resources for a process. There exist various namespace types which are used in combination to allow a fine-grained control. One of the most important namespaces in terms of security is the user namespace which allows unprivileged users to be privileged in an own designated namespace. Therefore, the risk of damage imposed by container breakouts is mitigated to a certain extent by making privilege escalations necessary prior to gaining elevated privileges on the host.

Monitoring and constraining of resource usage is being accomplished with control groups. As the name suggests, processes are able to be grouped with *cgroup* policies being attached. There are two versions of the control group implementation available that may be used in combination. Limits are being configured using *cgroup* controllers of which various types are available.

Container managers like *Docker* and LXC combine and configure the kernel primitives in a user-friendly way. It's also possible that sane defaults are being provided that make it easier to spawn containers with a fair level of security. It has been pointed out using own analyses that there's always a possibility of insufficient protection by default profiles or feature specifications. This can also be the case for kernel features themselves – for example user namespaces were also affected by security issues in the past.

As of today spawning containers with all namespace and control group features in place still requires elevated privileges. Most container managers solve this issue by running a privileged daemon or by providing SUID binaries. This may introduce certain security issues, for example in case user-supplied code can be executed in the context of such a privileged component. For this reason there's still ongoing work and discussion regarding unprivileged containers. A large part of the container spawning procedure is able to run without elevated permissions – except the control groups and networking configuration.

The survey on *systemd* unit files showed that as of now only a small number of services are being isolated from the host system using features that are already available. Most of the isolation directives are not in use at all or only used to a small extent.

8 Future Work

Considering all the advantages of containerization it's likely that containers will be increasingly important for Platform as a Service (PaaS) providers that offer to host environments of this type in the cloud. This may result in an increased adoption and new technologies being developed that are built on containerization. For example, a new approach named Secure Linux Containers on Intel SGX (SCONE), with SGX standing for Software Guard Extensions, aims to combine containers with secure enclaves in the cloud to allow secure execution of containers in untrusted environments. Secure enclaves are separate microprocessors that are able to store and protect secret data in a way that even prevents users with elevated privileges to read these secrets. This may hinder attackers from leveraging further attacks on the infrastructure after compromising an application container and gaining elevated privileges. Also, the trust that has to be put into the security of a cloud provider can be minimized.

With all the benefits gained from migrating to a container-based infrastructure, it's however unlikely that VM deployments will disappear in the future – both in the cloud and on the desktop. The ability to run separate kernels of various architectures and other operating systems on a single machine is indispensable for many scenarios. However, there are and will be new challenges regarding containerization and managing container landscapes: For example to prevent using outdated and vulnerable container images, it's necessary to implement proper processes into the software supply chain.

One approach that may receive an increased attention in the future is *Kata Containers*, as briefly described in 2.7.3. It's built with a specific trade-off in mind: Providing additional security by running a separate kernel for each container while making compromises regarding performance. It's still to be determined how this type of container performs in large-scale deployments – support for new technologies is constantly being implemented at the moment [fca].

The kernel primitives used for containerization may be improved and extended in the future. There are plans to implement new namespaces, such as the time namespace [ena], device namespace [enb] and several others [enc] [end] [ene]. Depending on the progress regarding the discussion on unprivileged and delegated control groups, the *cgroup* facility may be incorporated in the process of creating unprivileged containers in the future.

While the security of container environments can be improved even more in the future, it's also possible that container breakouts may happen. As with most other technologies, there's no guarantee that containerization is not prone to vulnerabilities that can affect the host system or other containers on a machine. It has been shown that breakouts of privileged *Docker* containers can happen in case of misconfiguration [pwh] – for unprivileged containers this still has be demonstrated.

It's possible that in the future more software projects will use container features in order to execute applications in isolated environments commonly referred to as *sandboxes*. This reduces the risk of damage imposed by a potentially vulnerable process. The *Chrome* browser already makes use of these features [csb] by splitting the functionality in multiple processes and applying appropriate isolation. For example, the part that renders the website layout processes untrusted data and is therefore a component that's prone to being exploited. To provide a better security level, this process is being sandboxed and only possesses a limited privilege level. *Chrome* uses a layered approach to perform this task in order to isolate the potentially vulnerable process itself and reduce the kernel's attack surface at the same time. As can be observed by visiting `chrome://sandbox`, namespaces and *seccomp* come to use.

The effort that's being put into the *Chrome* sandbox and similar approaches of other projects can serve as an example other developers can follow. It's often possible to reduce the attack surface of an application and the underlying system by using primitives offered by an operating system. Operating systems and drivers, mobile applications, the automotive sector and similar segments where privilege separation and isolation are important could therefore definitively benefit from the ideas of containerization.

List of *systemd* Isolation Directives

The following directives have been considered for the survey in chapter 5:

AmbientCapabilities	AppArmorProfile	BlockIOAccounting
BlockIODeviceWeight	BlockIOReadBandwidth	BlockIOWeight
BlockIOWriteBandwidth	Capabilities	Capability
CapabilityBoundingSet	CPUAccounting	CPUAffinity
CPUQuota	CPUSchedulingPolicy	CPUSchedulingPriority
CPUSchedulingResetOnFork	CPUShares	DeviceAllow
DevicePolicy	DirectoryMode	DropCapability
ExecStart	ExecStartPost	InaccessibleDirectories
InaccessiblePaths	JoinsNamespaceOf	MemoryAccounting
MemoryLimit	MountFlags	NoNewPrivileges
Private	PrivateDevices	PrivateMounts
PrivateNetwork	PrivateTmp	PrivateUsers
PrivateUsersChown	ProtectControlGroups	ProtectHome
ProtectKernelModules	ProtectKernelTunables	ProtectSystem
ReadOnlyDirectories	ReadOnlyPaths	ReadWriteDirectories
ReadWritePaths	RestrictNamespaces	RootDirectory
RootDirectoryStartOnly	RuntimeDirectoryMode	RuntimeDirectoryPreserve
RuntimeDirectorySize	RuntimeKeepFree	RuntimeMaxFiles
RuntimeMaxFileSize	RuntimeMaxUse	SecureBits
SELinuxContext	SELinuxContextFromNet	Slice
SystemCallFilter		

References

[aad] Moby Contributors. *template.go*. Retrieved: 12/18/2018 11:00. URL: https://github.com/moby/moby/blob/master/profiles/apparmor/template.go.

[aib] Moby Contributors. *mkimage-alpine.sh*. Retrieved: 09/13/2018 09:40. URL: https://github.com/moby/moby/blob/master/contrib/mkimage-alpine.sh.

[alp] Alpine Linux Development Team. *Alpine Linux*. Retrieved: 09/13/2018 09:42. URL: http://alpinelinux.org/.

[alr] Alpine Linux Development Team. *Alpine Edge x86 Releases*. Retrieved: 09/13/2018 09:42. URL: http://dl-cdn.alpinelinux.org/alpine/edge/releases/x86_64/.

[apc] App Container Contributors. *App Container Specification*. Retrieved: 09/17/2018 16:47. URL: https://github.com/appc/spec.

[asn] Aleksa Sarai. *Rootless Containers with runC*. Retrieved: 12/10/2018 07:33. URL: https://www.cyphar.com/blog/post/20160627-rootless-containers-with-runc.

[asu] Aleksa Sarai. *Rootless Containers (Pull Request)*. Retrieved: 12/03/2018 14:14. URL: https://github.com/opencontainers/runc/pull/774.

[awc] Arch Wiki Contributors. *cgroups*. Retrieved: 10/23/2018 14:55. URL: https://wiki.archlinux.org/index.php/cgroups.

[awd] Arch Wiki Contributors. *systemd-nspawn*. Retrieved: 12/06/2018 07:42. URL: https://wiki.archlinux.org/index.php/systemd-nspawn#Creating_private_users_(unprivileged_containers).

[aws] ArchWiki Contributors. *systemd-nspawn*. Retrieved: 09/18/2018 10:13. URL: https://wiki.archlinux.org/index.php/systemd-nspawn.

[cdg] containerd contributors. *containerd: An open and reliable container runtime*. Retrieved: 09/17/2018 09:02. URL: https://github.com/containerd/containerd.

[cfm] Linux Control Groups v1 Documentation Contributors. *freezer-subsystem*. Retrieved: 11/05/2018 11:16. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt.

[cfo] Linux Control Groups v2 Documentation Contributors. *cgroup-v2*. Retrieved: 11/05/2018 13:27. URL: https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

[cfs] Thorsten Ball. *Where did fork go?* Retrieved: 09/21/2018 09:45. URL: https://thorstenball.com/blog/2014/06/13/where-did-fork-go/.

[cgd] Tejun Heo. *Device Whitelist Controller*. Retrieved: 09/13/2018 17:01. URL: https://github.com/torvalds/linux/blob/master/Documentation/cgroup-v1/devices.txt.

[cgk] Aleksa Sarai. *[PATCH v4 0/2] cgroup: allow management of subtrees by new cgroup namespaces*. Retrieved: 12/03/2018 14:39. URL: https://lkml.org/lkml/2016/5/13/576.

[cgm] CGroup Namespaces Manual Contributors. *cgroup namespaces - overview of Linux cgroup namespaces*. Retrieved: 09/25/2018 15:39. URL: http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html.

[cgt] Neil Brown. *Control groups, part 3: First steps to control*. Retrieved: 10/23/2018 11:43. URL: https://lwn.net/Articles/605039/.

[cgx] Neil Brown. *Control groups, part 6: A look under the hood*. Retrieved: 10/29/2018 09:55. URL: https://lwn.net/Articles/606925/.

[cgz] Jake Edge. *Control group namespaces*. Retrieved: 09/26/2018 08:37. URL: https://lwn.net/Articles/621006/.

[ckf] Tejun Heo. *Re: [PATCH v4 0/2] cgroup: allow management of subtrees by new cgroup namespaces (1)*. Retrieved: 12/03/2018 14:49. URL: https://lkml.org/lkml/2016/5/20/522.

[ckg] Tejun Heo. *Re: [PATCH v4 0/2] cgroup: allow management of subtrees by new cgroup namespaces (3)*. Retrieved: 12/04/2018 07:59. URL: https://lkml.org/lkml/2016/5/31/752.

[ckh] Tejun Heo. *Re: [PATCH v4 0/2] cgroup: allow management of subtrees by new cgroup namespaces (2)*. Retrieved: 12/04/2018 09:34. URL: https://lkml.org/lkml/2016/5/20/550.

[clm] Clone Manual Contributors. *clone - create a child process*. Retrieved: 09/20/2018 16:00. URL: http://man7.org/linux/man-pages/man2/clone.2.html.

[cme] Michael Crosby. *Securing Containers, One Patch at a Time*. Retrieved: 12/11/2018 10:46. URL: https://www.youtube.com/watch?v=jZSs1RHwcqo.

[cmp] Linux Capabilities Manual Contributors. *Capabilities - Overview of Linux Capabilities*. Retrieved: 10/30/2018 15:24. URL: http://man7.org/linux/man-pages/man7/capabilities.7.html.

[cng] Jake Edge. *Inheriting capabilities*. Retrieved: 11/12/2018 08:06. URL: https://lwn.net/Articles/632520/.

[cnh] Jonathan Corbet. *Namespaced file capabilities*. Retrieved: 11/12/2018 08:08. URL: https://lwn.net/Articles/726816/.

[cns] Linux Contributors. *fork.c*. Retrieved: 10/22/2018 11:06. URL: https://github.com/torvalds/linux/blob/master/kernel/fork.c.

[cpf] *Performance Evaluation of Container-based Virtualization for High Performance Computing Environments*. Retrieved: 09/13/2018 10:18. URL: https://www.researchgate.net/profile/Fabio_Rossi10/publication/261266481_Performance_Evaluation_of_Container-Based_Virtualization_for_High_Performance_Computing_Environments/links/5500c3200cf2aee14b581172/Performance-Evaluation-of-Container-Based-Virtualization-for-High-Performance-Computing-Environments.pdf.

[cra] ArchWiki. *Change root*. Retrieved: 09/14/2018 13:41. URL: https://wiki.archlinux.org/index.php/change_root.

[cre] Filippo Valsorda. *Escaping a chroot jail/1*. Retrieved: 09/14/2018 13:59. URL: https://filippo.io/escaping-a-chroot-jail-slash-1/.

[crh] The Linux Foundation. *A Brief Look at the Roots of Linux Containers*. Retrieved: 09/14/2018 13:54. URL: https://www.linuxfoundation.org/blog/2017/05/a-brief-look-at-the-roots-of-linux-containers/.

[crs] Chroot Manual Contributors. *chroot - change root directory*. Retrieved: 09/19/2018 09:58. URL: http://man7.org/linux/man-pages/man2/chroot.2.html.

[csb] Google Chrome Developer Team. *Linux sandboxing*. Retrieved: 01/14/2019 10:14. URL: https://chromium.googlesource.com/chromium/src/+/master/docs/linux_sandboxing.md#User-namespaces-sandbox.

[cua] Jason A. Donenfeld. *kernel: tmpfs use-after-free*. Retrieved: 09/27/2018 11:57. URL: https://lwn.net/Articles/540083/.

[cva] National Vulnerability Database. *CVE-2016-9962*. Retrieved: 12/11/2018 11:08. URL: https://nvd.nist.gov/vuln/detail/CVE-2016-9962.

[cvg] MITRE (CVE List). *CVE-2014-8989*. Retrieved: 12/10/2018 10:16. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8989.

[dce] Docker Inc. *Docker CE*. Retrieved: 09/17/2018 08:30. URL: https://github.com/docker/docker-ce.

[dcr] Docker Inc. *Docker Structure: containerd and runC*. Retrieved: 09/17/2018 09:19. URL: http://www.linux-magazin.de/wp-content/uploads/2016/05/dockerrunc.png.

[dds] Docker Inc. *Isolate containers with a user namespace*. Retrieved: 12/04/2018 10:14. URL: https://docs.docker.com/engine/security/userns-remap/#disable-namespace-remapping-for-a-container.

[ded] Chris Swan. *Docker drops LXC as default execution environment*. Retrieved: 09/17/2018 13:16. URL: https://www.infoq.com/news/2014/03/docker_0_9.

[deo] Docker Inc. *Docker Overview: Docker Engine*. Retrieved: 09/14/2018 16:05. URL: https://docs.docker.com/engine/images/engine-components-flow.png.

[dgc] Moby/OCI Contributors. *defaults.go: Default Capability Set*. Retrieved: 11/26/2018 14:41. URL: https://github.com/moby/moby/blob/master/oci/defaults.go.

[dhb] Docker Inc. *Docker Hub*. Retrieved: 09/17/2018 13:06. URL: https://hub.docker.com/.

[dhs] Docker Inc. *5 Years later, where are you on your docker journey?* Retrieved: 09/17/2018 13:09. URL: https://blog.docker.com/2018/03/5-years-later-docker-journey/.

[dic] Antonio Murdaca. *Add /proc/acpi to masked paths (GitHub Pull Request*. Retrieved: 11/27/2018 16:46. URL: https://github.com/moby/moby/pull/37404.

[dil] Preethi Kasireddy. *A Beginner-Friendly Introduction to Containers, VMs and Docker*. Retrieved: 09/12/2018 16:38. URL: https://medium.freecodecamp.org/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b.

[dix] Philipp Schmied. *Minor Information Leaks in /proc/asound*. Retrieved: 11/28/2018 14:27. URL: https://github.com/moby/moby/issues/38285.

[dkr] Docker Inc. *Docker: Build, Manage and Secure Your Apps Anywhere. Your Way.* Retrieved: 09/13/2018 09:51. URL: https://docker.com/.

[dps] Docker/Moby Contributors. *Default Seccomp Profile: default.json*. Retrieved: 11/26/2018 10:38. URL: https://github.com/moby/moby/blob/master/profiles/seccomp/default.json.

[dpt] Docker Documentation Contributors. *service create*. Retrieved: 11/27/2018 11:16. URL: https://github.com/docker/cli/blob/master/docs/reference/commandline/service_create.md.

[drc] Arnaud Porterie. *Docker 1.11: The First Runtime Built on containerd and Based On OCI Technology*. Retrieved: 09/17/2018 08:59. URL: https://blog.docker.com/2016/04/docker-engine-1-11-runc/.

[dsd] Docker Inc. *About storage drivers*. Retrieved: 09/14/2018 10:12. URL: https://docs.docker.com/storage/storagedriver/.

[dsm] Docker Inc. *Seccomp security profiles for Docker*. Retrieved: 11/26/2018 10:58. URL: https://docs.docker.com/engine/security/seccomp/.

[dvl] Daniel Shapira. *Escaping Docker container using waitid() – CVE-2017-5123*. Retrieved: 09/12/2018 15:02. URL: https://www.twistlock.com/2017/12/27/escaping-docker-container-using-waitid-cve-2017-5123/.

[elm] Linux execl Manual Contributors. *execl(3)*. Retrieved: 11/09/2018 09:47. URL: https://linux.die.net/man/3/execl.

[ena] Jonathan Corbet. *Time namespaces*. Retrieved: 01/14/2019 09:44. URL: https://lwn.net/Articles/766089/.

[enb] Jake Edge. *Device namespaces*. Retrieved: 01/14/2019 09:46. URL: https://lwn.net/Articles/564854/.

[enc] Martijn Coenen. *Symbol namespaces*. Retrieved: 01/14/2019 09:46. URL: https://lwn.net/Articles/759928/.

[end] Richard Guy Brigg. *namespaces: log namespaces per task*. Retrieved: 01/14/2019 09:46. URL: https://lwn.net/Articles/644081/.

[ene] Jonathan Corbet. *Namespaced file capabilities*. Retrieved: 01/14/2019 09:47. URL: https://lwn.net/Articles/726816/.

[fca] Sebastian Grüner Ulrich Bantle. *Kata Container unterstützen Amazons Firecracker-Hypervisor*. Retrieved: 01/29/2019 10:23. URL: https://www.golem.de/news/virtualisierung-kata-container-unterstuetzen-amazons-firecracker-hypervisor-1901-139008.html.

[fjl] Matteo Riondato. *Jails*. Retrieved: 09/14/2018 13:26. URL: https://www.freebsd.org/doc/handbook/jails.html.

[fsc] glibc Contributors. *fork.c*. Retrieved: 09/21/2018 10:15. URL: https://github.com/lattera/glibc/blob/416bf844227d37b043b16be28c9523eeaecd3de3/nptl/sysdeps/unix/sysv/linux/fork.c.

[fwc] glibc Contributors. *fork.c (x86-64)*. Retrieved: 09/21/2018 10:23. URL: https://github.com/lattera/glibc/blob/416bf844227d37b043b16be28c9523eeaecd3de3/nptl/sysdeps/unix/sysv/linux/x86_64/fork.c.

[gdn] Cloud Foundry. *Rootless containers in Garde*. Retrieved: 12/05/2018 11:00. URL: https://github.com/cloudfoundry/garden-runc-release/blob/43a4aaf771a18173f1b57a8aec749d16433ad2a7/docs/rootless-containers.md.

[grp] gRPC Authors. *gRPC: A high performance, open-source universal RPC framework*. Retrieved: 09/17/2018 13:52. URL: https://grpc.io/.

[gvi] Google LLC. *gVisor: Container Runtime Sandbox*. Retrieved: 09/18/2018 11:40. URL: https://github.com/google/gvisor.

[gvl] gVisor Contributors. *gVisor: Layers*. Retrieved: 09/18/2018 12:07. URL: https://github.com/google/gvisor/blob/master/g3doc/Layers.png?raw=true.

[gvp] gVisor Contributors. *postgres needs sync file range*. Retrieved: 09/18/2018 12:15. URL: https://github.com/google/gvisor/issues/88.

[hhv] Robert P. Goldberg Gerald J. Popek. *Formal Requirements for Virtualizable Third Generation Architectures*. Retrieved: 09/11/2018 16:23. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.4815&rep=rep1&type=pdf.

[hvv] Virtuatopia. *An Overview of Virtualization Techniques*. Retrieved: 09/12/2018 10:45. URL: https://www.virtuatopia.com/index.php/An_Overview_of_Virtualization_Techniques.

[hyp] Microsoft Corporation. *Introduction to Hyper-V on Windows 10*. Retrieved: 09/11/2018 16:29. URL: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/.

[img] Jessica Frazelle et al. *Standalone, daemon-less, unprivileged Dockerfile and OCI compatible container image builder*. Retrieved: 12/05/2018 10:03. URL: https://github.com/genuinetools/img.

[ion] ioctl Manual Contributors. *ioctl ns - ioctl() operations for Linux namespaces*. Retrieved: 09/27/2018 14:52. URL: http://man7.org/linux/man-pages/man2/ioctl_ns.2.html.

[ioz] IOZone Project Contributors. *IOzone Filesystem Benchmark*. Retrieved: 09/13/2018 13:17. URL: http://www.iozone.org/.

[isd] Brian Christner. *Docker Base Image OS Size Comparison*. Retrieved: 09/12/2018 14:44. URL: https://www.brianchristner.io/docker-image-base-os-size-comparison/.

[Iva17] Konstantin Ivanov. *Containerization with LXC*. Birmingham: Packt, 2017. ISBN: 978-1-78588-894-6.

[jfk] Jessica Frazelle. *Keynote: Containers aka crazy user space fun*. Retrieved: 09/17/2018 08:36. URL: https://www.youtube.com/watch?v=7mzbIOtcIaQ.

[jfr] Jessica Frazelle. *Two Objects not Namespaced by the Linux Kernel*. Retrieved: 11/26/2018 11:16. URL: https://blog.jessfraz.com/post/two-objects-not-namespaced-linux-kernel/.

[jhz] Jann Horn. *Linux: broken uid/gid mapping for nested user namespaces with >5 ranges*. Retrieved: 11/16/2018 07:09. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1712.

[jpe] Jérôme Petazzoni. *Attach a volume to a container while it is running*. Retrieved: 11/27/2018 10:51. URL: https://jpetazzo.github.io/2015/01/13/docker-mount-dynamic-volumes/.

[kcs] OpenStack Foundation. *What is Kata Containers?* Retrieved: 09/17/2018 13:54. URL: https://katacontainers.io/static/katacontainers/img/kata-explained1%402x.png.

[krp] Dejan Lukan. *Introduction to Kernel Debugging*. Retrieved: 09/12/2018 09:07. URL: https://resources.infosecinstitute.com/introduction-to-kernel-debugging/.

[kvs] Lucian Carata. *Linux Containers: Basic Concepts*. Retrieved: 09/27/2018 15:37. URL: https://www.cl.cam.ac.uk/~lc525/files/Linux_Containers.pdf.

[lcd] Will Drewry. *[RFC,PATCH 2/2] Documentation: prctl/seccomp filter*. Retrieved: 11/23/2018 09:18. URL: https://lwn.net/Articles/475049/.

[lcp] Will Drewry. *dynamic seccomp policies (using BPF filters)*. Retrieved: 11/23/2018 09:17. URL: https://lwn.net/Articles/475019/.

[lcs] Linux Contributors. *cgroup-defs.h*. Retrieved: 10/26/2018 10:45. URL: https://github.com/torvalds/linux/blob/master/include/linux/cgroup-defs.h#L176.

[ldr] Twain Taylor. *Docker vs. LXC—The Similarities and Differences*. Retrieved: 09/17/2018 13:43. URL: http://blog.wercker.com/what-is-the-difference-between-lxc-and-docker.

[ldu] LXD Contributors. *cgroup.go*. Retrieved: 12/03/2018 13:05. URL: https://github.com/lxc/lxd/blob/master/lxd/cgroup.go#L56.

[lnf] Michael Kerrisk. *Namespaces in operation, part 2: the namespaces API*. Retrieved: 09/21/2018 15:41. URL: https://lwn.net/Articles/531381/.

[lpk] Julie Langou Jack Dongarra Bonnie Toy. *Linpack*. Retrieved: 09/13/2018 11:18. URL: https://github.com/2000nickels/linpackc/blob/master/linpack.c.

[lsf] Linux Contributors. *uapi/linux/filter.h: Linux Socket Filter Data Structures*. Retrieved: 11/22/2018 14:53. URL: https://github.com/torvalds/linux/blob/master/include/uapi/linux/filter.h.

[lsh] *Linux Security Module Framework*. Retrieved: 12/18/2018 09:29. URL: http://www.kroah.com/linux/talks/ols_2002_lsm_paper/lsm.pdf.

[lsm] Linux Kernel Development Community. *Linux Security Module Usage*. Retrieved: 12/18/2018 09:20. URL: https://www.kernel.org/doc/html/v4.16/admin-guide/LSM/index.html.

[lss] Linux Contributors. *uapi/linux/seccomp.h*. Retrieved: 11/22/2018 14:54. URL: https://github.com/torvalds/linux/blob/master/include/uapi/linux/seccomp.h.

[lwg] Jonathan Corbet. *User namespaces and setgroups()*. Retrieved: 12/10/2018 10:07. URL: https://lwn.net/Articles/626665/.

[lws] Jake Edge. *A seccomp overview*. Retrieved: 11/22/2018 11:45. URL: https://lwn.net/Articles/656307/.

[lxg] Linux container projects. *LXC*. Retrieved: 09/17/2018 13:36. URL: https://github.com/lxc/lxc.

[mcd] Michael Crosby. *Containerd- Building a Container Supervisor - Black Belt Track*. Retrieved: 09/17/2018 12:50. URL: https://www.youtube.com/watch?v=VWuHWfEB6ro.

[mcg] Linux Control Group Contributors. *Linux control groups*. Retrieved: 10/23/2018 14:29. URL: http://man7.org/linux/man-pages/man7/cgroups.7.html.

[mgn] Marcus Gelderie. *Advanced Linux Security Course: Namespaces*. Retrieved: 11/07/2018 10:56.

[mpn] Michael Kerrisk. *Namespaces in operation, part 4: more on PID namespaces*. Retrieved: 09/24/2018 11:20. URL: https://lwn.net/Articles/532748/.

[mps] Linux Seccomp Manual Contributors. *seccomp - operate on Secure Computing state of the process*. Retrieved: 11/22/2018 11:45. URL: https://lwn.net/Articles/656307/.

[mqe] Gustavo RV 86. *queue example threads*. Retrieved: 09/25/2018 15:09. URL: https://gist.github.com/gustavorv86/9e98621b44222114524399a3b4302ddb.

[mqo] MQ Manual Contributors. *mq overview - overview of POSIX message queues*. Retrieved: 09/25/2018 14:53. URL: http://man7.org/linux/man-pages/man7/mq_overview.7.html.

[msc] util-linux Contributors. *mount.c*. Retrieved: 09/24/2018 17:10. URL: https://github.com/karelzak/util-linux/blob/master/sys-utils/mount.c.

[nie] Linux Control Group Documentation Contributors. *Network classifier group*. Retrieved: 10/24/2018 07:30. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/net_cls.txt.

[nmd] Tim Bozarth (Netflix Technology Blog) Andrew Spyker Andrew Leung. *The Evolution of Container Usage at Netflix*. Retrieved: 09/10/2018 11:31. URL: https://medium.com/netflix-techblog/the-evolution-of-container-usage-at-netflix-3abfc096781b.

[nms] Tony Mauro (NGINX Inc.) *Adopting Microservices at Netflix: Lessons for Architectural Design*. Retrieved: 09/10/2018 11:41. URL: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[nnk] Linux Kernel Contributors. *dev.c*. Retrieved: 09/25/2018 12:15. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/core/dev.c?h=master#n9098.

[nnp] Linux prctl Documentation Contributors. *No New Privileges*. Retrieved: 11/26/2018 13:38. URL: https://www.kernel.org/doc/Documentation/prctl/no_new_privs.txt.

[nns] Jake Edge. *Namespaces in operation, part 7: Network namespaces*. Retrieved: 09/25/2018 11:11. URL: https://lwn.net/Articles/580893/.

[npp] *netpipe*. Retrieved: 09/13/2018 13:42. URL: https://linux.die.net/man/1/netpipe.

[nsc] Linux Contributors. *nsproxy.c*. Retrieved: 10/22/2018 11:10. URL: https://github.com/torvalds/linux/blob/603ba7e41bf5d405aba22294af5d075d8898176d/kernel/nsproxy.c.

[nse] Jerome Petazzon. *nsenter*. Retrieved: 09/24/2018 09:28. URL: https://github.com/jpetazzo/nsenter.

[nsf] Michael Kerrisk. *Mount namespaces and shared subtrees*. Retrieved: 09/24/2018 16:57. URL: https://lwn.net/Articles/689856/.

[nsi] Shichao An. *Process Management*. Retrieved: 10/22/2018 09:55. URL: https://notes.shichao.io/lkd/ch3/.

[nso] Namespace Manual Contributors. *namespaces - overview of Linux namespaces*. Retrieved: 09/25/2018 15:01. URL: http://man7.org/linux/man-pages/man7/namespaces.7.html.

[ntd] NTP Project. *ntpd.c*. Retrieved: 11/12/2018 07:37. URL: https://github.com/ntp-project/ntp/blob/47e9cc7fd5b31417ac34b1b97b7f72fad0fb03dc/ntpd/ntpd.c#L938.

[oci] Open Container Initiative Contributors. *Open Container Initiative*. Retrieved: 09/17/2018 09:11. URL: https://www.opencontainers.org/.

[osc] S. K. Tripathi Dev Ras Pandey Bharat Mishra. *Operating System Based Virtualization Models in Cloud Computing*. Retrieved: 09/19/2018 09:39. URL: https://pdfs.semanticscholar.org/46c9/d9d971f1a0e8f658e3fd6293c1061e5209b3.pdf.

[pmp] PID Namespace Contributors. *PID namespaces - overview of Linux PID namespaces*. Retrieved: 09/24/2018 15:55. URL: http://man7.org/linux/man-pages/man7/pid_namespaces.7.html.

[ppd] Evan Sarmiento. *The Jail Subsystem*. Retrieved: 09/14/2018 14:34. URL: https://www.freebsd.org/doc/en/books/arch-handbook/jail.html.

[ppe] Evan Sarmiento. *The Jail Subsystem - Restrictions*. Retrieved: 09/14/2018 14:34. URL: https://www.freebsd.org/doc/en/books/arch-handbook/jail-restrictions.html.

[pwh] Nimrod Stoler. *How I Hacked Play-with-Docker and Remotely Ran Code on the Host*. Retrieved: 01/16/2019 07:38. URL: https://www.cyberark.com/threat-research-blog/how-i-hacked-play-with-docker-and-remotely-ran-code-on-the-host/.

[pzg] Jann Horn. *gVisor runsc guest->host breakout via filesystem cache desync*. Retrieved: 11/19/2018 11:04. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1631.

[rcg] Open Container Initiative. *runc: CLI tool for spawning and running containers according to the OCI specification*. Retrieved: 09/17/2018 09:00. URL: https://github.com/opencontainers/runc.

[rka] App Container Contributors. *App Container Executor*. Retrieved: 09/17/2018 16:37. URL: https://github.com/appc/spec/blob/master/spec/ace.md.

[rkb] Rkt Contributors. *rkt pods should not be given CAP SYS ADMIN*. Retrieved: 09/17/2018 16:40. URL: https://github.com/rkt/rkt/issues/576.

[rkc] LWN. *Demystifying container runtimes*. Retrieved: 09/17/2018 16:45. URL: https://lwn.net/Articles/741897/.

[rkd] rkt Contributors. *rkt architecture*. Retrieved: 09/18/2018 08:48. URL: https://github.com/rkt/rkt/blob/master/Documentation/devel/architecture.md.

[rko] CoreOS Inc. *CoreOS has joined the Red Hat family*. Retrieved: 09/18/2018 09:10. URL: https://coreos.com/.

[rlc] Aleksa Sarai. *Rootless Containers*. Retrieved: 12/05/2018 10:57. URL: https://rootlesscontaine.rs/.

[rns] runc/libcontainer Contributors. *nsenter*. Retrieved: 12/06/2018 10:45. URL: https://github.com/opencontainers/runc/tree/master/libcontainer/nsenter.

[scd] Michael Kerrisk. *Glibc and the kernel user-space API*. Retrieved: 09/21/2018 10:09. URL: https://lwn.net/Articles/534682/.

[scf] Filippo Valsorda. *Searchable Linux Syscall Table for x86 and x86-64*. Retrieved: 09/20/2018 14:05. URL: https://filippo.io/linux-syscall-table/.

[sct] Linux Contributors. *64-bit system call numbers and entry vectors*. Retrieved: 09/20/2018 14:01. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/entry/syscalls/syscall_64.tbl.

[scv] Michael Eder. *Hypervisor- vs. Container-based Virtualization*. Retrieved: 09/11/2018 13:44. URL: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2016-07-1/NET-2016-07-1_01.pdf.

[sdd] Debian Wiki Contributors. *systemd - system and service manager*. Retrieved: 09/18/2018 10:04. URL: https://wiki.debian.org/systemd.

[sde] Systemd Contributors. *systemd.exec — Execution environment configuration*. Retrieved: 12/20/2018 14:55. URL: https://www.freedesktop.org/software/systemd/man/systemd.exec.html.

[sdu] Systemd Manual Contributors. *systemd.unit — Unit configuration*. Retrieved: 09/18/2018 10:37. URL: https://www.freedesktop.org/software/systemd/man/systemd.unit.html.

[sel] Daniel J Walsh. *Your visual how-to guide for SELinux policy enforcement*. Retrieved: 12/18/2018 09:49. URL: https://opensource.com/business/13/11/selinux-policy-guide.

[sem] Juan Antonio Osorio. *SELinux and docker notes*. Retrieved: 12/18/2018 10:12. URL: http://jaormx.github.io/2018/selinux-and-docker-notes/.

[slc] Major Hayden. *Securing Linux Containers*. Retrieved: 09/12/2018 10:08. URL: https://major.io/wp-content/uploads/2015/08/Securing-Linux-Containers-GCUX-Gold-Paper-Major-Hayden.pdf.

[srp] Linux Programmer's Manual. *prctl - operations on a process*. Retrieved: 09/17/2018 11:37. URL: http://man7.org/linux/man-pages/man2/prctl.2.html.

[sss] Will Drewdry / OpenSSH Contributors. *sandbox-seccomp-filter.c*. Retrieved: 11/23/2018 09:30. URL: https://github.com/openssh/openssh-portable/blob/master/sandbox-seccomp-filter.c.

[sst] corbet. *Shared subtrees*. Retrieved: 09/25/2018 08:53. URL: https://lwn.net/Articles/159077/.

[str] Ph.D. John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Retrieved: 09/13/2018 12:54. URL: https://www.cs.virginia.edu/stream/.

[szd] *Virtualization and Namespace Isolation in the Solaris Operating System (PSARC/2002/174)*. Retrieved: 09/14/2018 14:47. URL: https://us-east.manta.joyent.com/jmc/public/opensolaris/ARChive/PSARC/2002/174/zones-design.spec.opensolaris.pdf.

[tkc] Georg Schönberger. *Linux I/O Scheduler*. Retrieved: 11/06/2018 12:58. URL: https://www.thomas-krenn.com/de/wiki/Linux_I/O_Scheduler.

[tns] Scott McCarty. *The Business Case for Running a Container-based Infrastructure*. Retrieved: 09/10/2018 09:08. URL: https://thenewstack.io/containers-enable-companies-just-less/.

[tts] Inc. Netflix. *Titus*. Retrieved: 09/10/2018 12:00. URL: https://netflix.github.io/titus/.

[unm] User Namespace Manual Contributors. *user namespaces - overview of Linux user namespaces*. Retrieved: 09/27/2018 14:40. URL: http://man7.org/linux/man-pages/man7/user_namespaces.7.html.

[unv] Michael Kerrisk. *Anatomy of a user namespaces vulnerability*. Retrieved: 09/27/2018 13:25. URL: https://lwn.net/Articles/543273/.

[unz] Michael Kerrisk. *Namespaces in operation, part 5: User namespaces*. Retrieved: 09/26/2018 09:58. URL: https://lwn.net/Articles/532593/.

[vbc] Niklas Baumstark. *CVE-2018-3055,3085*. Retrieved: 09/19/2018 09:11. URL: https://github.com/niklasb/3dpwn/tree/master/CVE-2018-3055%2B3085.

[vbx] Oracle Corporation. *VirtualBox*. Retrieved: 09/11/2018 16:31. URL: https://www.virtualbox.org/.

[vsc] jancorg. *Linux Kernel Namespaces Pt I*. Retrieved: 10/22/2018 11:07. URL: http://jancorg.github.io/blog/2015/01/05/linux-kernel-namespaces-pt-i/.

[wam] Richard Li. *What are microservices? (A pragmatic definition)*. Retrieved: 09/10/2018 11:53. URL: https://www.datawire.io/what-are-microservices-a-practical-definition/.

[wmp] Will Martin. *The Route to Rootless Containers*. Retrieved: 12/04/2018 11:48. URL: https://www.youtube.com/watch?v=DvZNsZs35qo.

[xen] Xen Project. *Xen Project*. Retrieved: 09/11/2018 16:26. URL: https://github.com/xen-project.

[xhs] Xen Project. *History*. Retrieved: 09/12/2018 15:50. URL: https://www.xenproject.org/about/history.html.

[xvl] Sean Michael Kerner. *Xen Patches Hypervisor Breakout Risk Without Breaking the Cloud*. Retrieved: 09/12/2018 11:17. URL: http://www.eweek.com/security/xen-patches-hypervisor-breakout-risk-without-breaking-the-cloud.