NET5038/NET5039 – Bibliographic review
Protection provided by operating system-level virtualization

Yassine Tioual & Thibaut Sautereau

Referent teacher: Christian BAC

June 6, 2016

Contents

Chapter 1

Introduction

Although hardware virtualization has proved its robustness and has been used for a long time, operating system-level virtualization (*ie* containers) is quickly growing in popularity. Indeed, containers are the building brick of PaaS (Platform as a Service) infrastructures, they are easily scalable, decrease power and storage costs, facilitate deployments for developers, reduce attack surface, etc. Moreover, as Linux is spread across many systems, from servers to embedded devices through mobile phones and laptops, the development of Linux containers is highly supported by many companies.

An easy way of seeing a container would be a lightweight virtual machine: it has its own process space, network interfaces, it can install packages, run services, run processes as root, etc. But as opposed to hardware virtualization, a container uses the host kernel, processes still run on the host machine, it cannot boot a different operating system, cannot have its own modules, does not need init as PID 1, etc. After all, operating system-level virtualization looks like an advanced chroot. So how do we ensure security in software virtualization?

Even if the basic principles are the same for almost all systems (or at least for UNIX systems), their implementation is OS specific. Therefore and as justified above, we chose to concentrate on Linux based mechanisms of isolation. At first, we will discuss the isolation mechanisms provided by the Linux kernel and then we will focus on how most popular operating system-level virtualization solutions implement them. Finally, we will analyze some security flaws of those solutions and how they are related to the previous mechanisms.

Chapter 2

Mechanisms of isolation

2.1 Cgroups

Control Groups[1], commonly referred to as cgroups, provide a mechanism for easily managing and monitoring system resources, by partitioning subsystems such as CPU time, system memory, disk and network bandwidth into groups, and then assigning tasks to these groups.

Cgroups have been around since 2006, and they have been merged into the Linux kernel mainline version 2.6.24 in 2008. This Linux kernel feature provides system administrators with fine-grained control over allocating, prioritizing, denying, managing and monitoring system resources. Other approaches were used in the past, like the `nice`[2] command or `/etc/security/limits.conf`[3], but they are not as flexible as cgroups.

And as with the Linux processes model, cgroups are also hierarchical: child cgroups inherit certain attributes from their parent cgroup. The main difference between cgroups and processes is that multiple hierarchies of cgroups can exist simultaneously on a given system. The reason behind having this kind of multiplicity is that each cgroup hierarchy is linked to one or more subsystems, a susbsystem[4] being the representation of a single system resource:

- **`blkio`**: The block input/output subsystem. It provides a way to limit and monitor input/output from and to block devices such as hard drives, USB drives, etc.
- **`cpu`**: This subsystem makes use of the scheduler to manage the CPU access to cgroups.
- **`cpuacct`**: Per-cgroup CPU usage reports are generated by this subsystem.
- **`cpuset`**: Provides a way to pin a group of processes to a one or multiple cores in a multi-core processor. This is useful to minimize context switch

[1] https://en.wikipedia.org/wiki/Cgroups
[2] http://linux.die.net/man/1/nice
[3] http://ss64.com/bash/ulimit.html
[4] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html-single/Resource_Management_Guide/index.html

in latency-sensitive environments or in NUMA[5] (Non Uniform Memory Access) systems for instance.

- **devices**: Defines access control to different devices by processes.
- **freezer**: The freezer subsystem can suspend or resume tasks in a cgroup. This is useful to preserve the state of a container during a migration.
- **memory**: Provides both memory limitation and accounting for cgroups.
- **net_cls**: In a similar way to VLANs, network packets originating from a particular cgroup set are tagged with a `calssid` (class identifier) so that the Linux `tc` (traffic controller) can identify them.
- **net_prio**: Aggregates network traffic of network interfaces based on priority.
- **ns**: The namespace subsystem.

Cgroups are easy to use, and there are different tools available online to do just that:

- On the fly creation/management tools like **cgcreate**, **cgexec**[6], etc.
- Systemd
- Container management tools such as LXC, Docker or Rkt

The first category of tools have the merit of being simple and easy to use. Their syntax is pretty straightforward, and a cgroup can be created as following:

```
> cgcreate -a <user> -g <subsystem>:<cgroup name>
```

Behind the scenes, what is really happening is that a new folder is created in the virtual file system `/sys/fs/cgroup`. Each folder of this virtual file system corresponds to one of the subsystems discussed earlier. An equivalent to the previous command would be:

```
> mkdir /sys/fs/cgroup/<subsystem>/<cgroup name>
> chown -R <user> /sys/fs/cgroup/<subsystem>/<cgroup name>
> chown root /sys/fs/cgroup/<subsystem>/<cgroup name>/tasks
#This file is owned by root when using cgcreate
```

The next step is printing the limitation or boolean value to the right file (called tunable) under the newly created cgroup directory. For example, to activate CPU accounting for a `test` cgroup:

```
> echo 1 > /sys/fs/cgroup/cpu/cpuacct.stat
```

The last step is to attach processes to the cgroup. This can be done with `cgexec`:

```
> cgexec -g <subsystem>:<cgroup name> <command>
```

Or by attaching already running processes:

```
> echo <PID> >> /sys/fs/cgroup/<subsystem>/<cgroup name>/cgroup.procs
```

[5] https://fr.wikipedia.org/wiki/Non_Uniform_Memory_Access

[6] libcgroup^{AUR} for Arch Linux, cgroup-tools for Debian

2.2 Namespaces

Like cgroups, namespaces are also a Linux kernel feature, inspired by the more general namespace functionality used by Plan 9 from Bell Labs[7]. The first Linux namespace (the **Mount** one) was introduced in kernel 2.4.19 from 2002. As of kernel 3.8, the implementation of current namespaces was considered finished. Namespaces are the building block of isolation, and therefore of containers. Basically, every process is associated with a namespace, of each kind, and can only see the resources associated with that namespace. One could say it is more a *view limitation* than a *do limitation* as with cgroups. Namespaces can be created, joined and left. They can also be bundled together to obtain a really customizable isolation filter.

For each process, symbolic links can be found in **`/proc/<pid>/ns/`**, which point to inode numbers. These inode numbers are the same for every single process belonging to a given namespace (readlink). Three syscalls directly manipulate namespaces:

- **`clone(2)`**, especially with its `CLONE_NEW` flag.
- **`unshare(2)`** and **`setns(2)`**, which were specifically added for security concerns.

There are six kinds of namespaces[8], each of which is discussed below. It is important to keep in mind that even if namespaces are often used altogether to create a container, they can also be used separately to add isolation in other cases where "full containers" are unneeded.

2.2.1 Mount namespace

A mount namespace is the set of filesystem mounts that are visible to processes belonging to that namespace. This was the first namespace to be introduced in the Linux kernel. It uses the `CLONE_NEWNS` from `clone(2)` so the child process gets a copy (instead of the traditionnal pointer) of its parent's mounted filesystem data, which it can change without affecting the parent's mount namespace.

2.2.2 IPC namespace

As its name indicates, the IPC (Inter-Process Communications) namespace allows creating objects shared by every process belonging to that namespace. It uses the `CLONE_NEWIPC` flag.

2.2.3 UTS

This namespace is as simple as it is necessary: it adds support for separate hostname and domain names values. The `CLONE_NEWUTS` flag needs to be set for a child process to use this namespace.

[7]http://www.cs.bell-labs.com/sys/doc/names.html
[8]http://man7.org/linux/man-pages/man7/namespaces.7.html

2.2.4 PID namespace

The PID namespace is sometimes considered the most important one as it helps create containers by preventing cross-application attacks, information leaks and other weaknesses. This namespace can be seen as a chroot in the process identifier tree: by calling `clone(2)` with the `CLONE_NEWPID` flag, the `getpid()` function called from within the resulting child process will return 1.

2.2.5 Network namespace

Processes within a network namespace have their own network stack (own lo, eth0 and routing tables for instance). This can be used for example to isolate a process from the network, or to apply a specific routing for a group of processes. Network namespaces can be manipulated with the **`ip netns`** command from the iproute2 collection of utilities. The corresponding `clone(2)` flag is `CLONE_NEWNET`.

2.2.6 User namespace

The user namespace is the most recent one and is crucial for containers security. It basically consists of a remapping of UIDs and GIDs: for instance, UID 9 on host in mapped to UID 0 in another user namespace. This way, processes from within a container think they are operating as root, but they are still low-privileged from the host's point of view. They also open the road to containers created and run by unprivileged users, although this has been responsible for a huge amount of critical vulnerabilities in modern containers. The corresponding `clone(2)` flag is `CLONE_NEWUSER`.

2.3 Capabilities

On Unix-like systems such as Linux, the **root** user (uid 0) has full power over the whole system. Exploitation of setuid-root binaries or simply of applications running as root (and therefore ignoring the *least privilege* principle) in the first place has been a recurring issue since the birth of that operating system. A major reason for that is that the frontier between unprivileged and privileged users often seems to be an annoying gap for most systems administrators.

This is what **Linux capabilities** are trying to address: introduced in kernel 2.2, they bring more granularity to every process, thread or file through a map of permission bits (bitmap) enforced by the kernel. Capabilities are passed to processes and threads *via* a structure given to the **`clone(2)`** and **`fork(2)`** syscalls, or dedicated syscalls like **`capset(2)`** and **`capget(2)`**.

As for files, capabilities are set using CLI utilities such as **`setcap(8)`** and stored as extended file attributes, if available on the filesystem. There are three different capability sets for threads, which are:

- **Effective**: this set is the one used by the kernel to check permissions when needed.
- **Permitted**: this set is the limiting superset for the previous one. A capability has to appear in this set if a process wants to add it to its *Effective* set.

- **Inheritable**: those capabilities *can be* preserved when calling `execve(2)` (see below).

In addition to the above sets, the **Ambient** set has been introduced: a capability belonging to this set is always preserved across an `execve(2)` as long as the calling process has it in its *Permitted* and *Inheritable* sets.

For files, the sets have a different meaning:

- **Effective**: in this case it is just a single bit telling if the new permitted capabilities should directly been set in the effective set of the new thread.
- **Permitted**: these capabilities are inevitably set for the new thread.
- **Inheritable**: this set is ANDed with the calling thread's inheritable set to get the new permitted capabilites for the resulting thread.

Among capabilities are for instance `CAP_NET_BIND_SERVICE` (useful for a webserver, as it only needs privileges to bind to a <1024 port) or the very dangerous `CAP_SYS_MODULE` (allowing arbitrary kernel modules loading and unloading). One should notice that the User namespace and capabilities are intended to work together and without conflicts[9]: any process creating or joining a namespace is given a new set of capabilities that do not apply in the parent/previous User namespace.

Although first implemented quite a long time ago, Linux capabilities still need to improve as they are a very complex feature and can quickly lead to privilege escalation[10].

2.4 MAC

Like the well-known DAC (Discretionary Access Control), MAC (Mandatory Access Control) is an access control model. DAC was largely implemented by UNIX systems through the read, write and execute rights for owners, groups and others. The key thing here is that each resource in DAC has a list of users who can access it: DAC provides access by identity of the user and not by permission level.

On the contrary, when using MAC, an admin creates a set of levels and each user is linked with a specific access level. He can then access all the **labeled** (ie mapped to a security context) resources that are not greater than his access level[11]. Although possibly quite cumbersome to configure, MAC gives admins the means to apply security policies to an entire system.

The two most popular native methods of MAC enforcement for Linux are discussed below.

2.4.1 SELinux

SELinux (Security-Enhanced Linux) is a LSM (Linux Security Module) originally developped by the NSA to *separate enforcement of security decisions from*

[9]http://man7.org/linux/man-pages/man7/user_namespaces.7.html
[10]http://man7.org/linux/man-pages/man7/capabilities.7.html
[11]https://en.wikipedia.org/wiki/Mandatory_access_control

the security policy itself[12]. It was merged to the Linux kernel mainline in the 2.6 series of the Linux kernel.

Although its policy language is quite complex[13], SELinux relies on a simple principle: every single user or process is given a context consisting of a username, a role and a domain. Most of the time, all real users share the same SELinux username and the difference is made by the domain. That is how security policies are defined: for instance, if a user wants to be granted access to a device, he must possess the required domain. Each policy is compiled and then loaded into the kernel.

SELinux provides the `-Z` option to several shell commands like `ps` or `ls` for seeing the security context of a ressource.

2.4.2 App Armor

AppArmor is also a LSM and was included in the Linux kernel mainline in version 2.6.36[14]. It is expected to be an easier alternative to SELinux.

Like SELinux, AppArmor allows for manually creating profiles but it also provides a learning mode (violations are only logged) which is very helpful to build adapted profiles[15]. Many profile generators are also available to make that easier, such as aa-genprof[16], one of the main drawbacks being that profiling requires the targeted application or container to be run long enough (no static analysis). Furthermore, AppArmor works with file paths instead of filesystems inodes, which makes it easier to configure but vulnerable to path modification attacks.

2.5 Pivot_root

Pivot_root() is a mechanism for changing the root file system of the current process to the directory *put_old* and making the *new_root* directory the new root of the current process. The **`pivot_root(2)`** syscall is needed to enter a container environment (to *pivot* in it)[17].

Other typical uses of **Pivot_root** are changing the root file system from one storage device to another, or mounting a new root file system over NFS[18].

Pivot_root is also used at system boot time by the **`/sbin/init`** program to mount the *real* root file system and pivot with the previously mounted initial ramdisk image.

2.6 Seccomp

Seccomp (short for SECure COMPuting mode) is a sandboxing mechanism implemented in the 2.6.12 Linux kernel. Basically, the first version of Sec-

[12]https://en.wikipedia.org/wiki/Security-Enhanced_Linux
[13]http://cecs.wright.edu/~pmateti/Courses/7900/Lectures/Security/NSA-SE-Android/Figs/selinux%20architecture.png
[14]https://en.wikipedia.org/wiki/AppArmor
[15]https://www.suse.com/documentation/sles11/singlehtml/apparmor_quickstart/apparmor_quickstart.html
[16]http://manpages.ubuntu.com/manpages/precise/en/man8/aa-genprof.8.html
[17]http://man7.org/linux/man-pages/man2/pivot_root.2.html
[18]http://man7.org/linux/man-pages/man8/pivot_root.8.html

comp (SECCOMP_MODE_STRICT) set a process into a *secure* state (one-way transition) from which it could not make any syscalls except **`_exit(2)`**, **`sigreturn(2)`** and **`read(2)/write(2)`** to already-open file descriptors. Kernel would then send `SIGKILL` to any process attempting other syscalls.

It obviously lacked flexibility so a patch was developped in 2012 by Will Drewry[19]: it takes advantage of the Berkeley Packet Filter to evaluate syscalls and their arguments, and was therefore called seccomp_BPF. However, generating the correct and minimal syscall filter is a complex task with potentially disastrous consequences. Furthermore, seccomp_BPF can quickly bring performance issues, especially when using a lot of rules[20].

Still, Seccomp should not be considered a sandbox itself: it is just a tool to build sandboxes since it only reduces the kernel attack surface[21]. For instance, it is used by latest OpenSSH versions to handle pre-authentication network traffic, and also by vsftpd or the Google Chrome browser[22]. Seccomp can be invoked using **`prctl(2)`** or directly *via* **`seccomp(2)`**.

[19]https://lwn.net/Articles/475019/
[20]https://lwn.net/Articles/656307/
[21]https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt
[22]https://github.com/docker/docker/blob/master/docs/security/seccomp.md

Chapter 3

Implementation in operating system-level virtualization solutions

In this chapter, we will take a quick look at how LXC, Docker and CoreOS Rocket implement the security mechanisms we discussed above.

3.1 LXC

There are two types of LXC containers:

– Privileged containers

– Unprivileged containers

The former type refers to containers in which uid 0 is mapped to the host's uid 0. It means that there is no real isolation between the host and the guest, and such containers should not be used unless the container's root user is trusted. Furthermore, no security mechanisms are implemented by default by LXC in privileged containers, even though host protection can always be reached *via* isolation techniques discussed in Chapter 1 (MAC, Seccomp filters, etc.)[1].

In real world scenarios where containers' users are not to be trusted, this type of containers should not be used, and we will not be discussing it in this paper.

3.1.1 Security in unprivileged LXC containers

Unpriviliged containers are used to encapsulate untrusted environments and were designed with this idea in mind. The security model relies mainly on namespaces for isolation and cgroups for resources management. As a consequence, processes within a LXC container cannot interact with other processes (on the host or on any other LXC container), nor access the root filesystem or special devices unless configured otherwise. The Linux kernel implementation of

[1]https://linuxcontainers.org/fr/lxc/security

namespaces is somehow incomplete, and some parts of special filesystems (e.g. `/proc` & `/sys`) are not namespaced. This makes LXC alone vulnerable to attacks such as denials of service or data leaks. This is why LXC is shipped with AppArmor in Ubuntu (12.10 and greater) and other Debian-based distributions. Other security mechanisms are used alongside with AppArmor to ensure better security and reduce the attack surface:

- **Seccomp-BPF** which has been a supported option in LXC for a long time. It comes by default with a minimal blacklist, but whitelists can be used as well.
- **User namespace** support, which has been provided since LXC 1.0.
- **SELinux** to define a Mandatory Access Control.
- **Grsecurity** patch which can be configured manually on the host kernel.

It is hard to assert that a LXC implementation is secure, since security depends mainly on context, and custom configuration is always required to attain a good level of security. For example, access to */dev/random* and */dev/urandom* is granted by default to processes running inside a LXC container. Access to such devices is not considered as a security threat, unless it can be used to retrieve system state or data that may have been generated by the host system using randomness (e.g. keys generation).

3.2 Docker

Like LXC, Docker[2] is an operating system-level virtualization solution built on top of cgroups and namespaces. It also uses copy-on-write to quickly deploy containers and facilitates devops operations[3]. First versions of Docker were using the LXC engine but since version 0.9, it provides its own engine called **libcontainer**.

Unlike LXC, Docker was at first intended to be a single-application container: it focuses on shipping software through building container images. That is a reason why Docker is able to apply strong security defaults to containers. By the way, many components of Docker are developed in Golang, which avoids many traditional vulnerabilities related to memory corruption.

Furthermore, MAC is strongly supported: AppArmor is activated by default with many available profiles[4], many of them inspired by the ones from LXC. The Docker Engine itself uses an AppArmor profile[5]. SELinux is also supported.

Since Docker 1.10, Seccomp filtering is enabled with a default profile (a whitelist of 310 syscalls[6] and a blacklist of 100 others[7]).

As for namespaces, Docker does not run the User namespace by default although it was recently implemented in 1.10, because it disables some other Docker features. One consequence is that the Docker daemon still needs to

[2] `https://www.docker.com/`
[3] `https://en.wikipedia.org/wiki/Docker_(software)`
[4] `https://docs.docker.com/engine/security/apparmor/`
[5] `https://github.com/docker/docker/commit/39dae54a3f40035b1b7e5ca86c53d05dec832ed2`
[6] `https://raw.githubusercontent.com/docker/docker/master/profiles/seccomp/default.json`
[7] `https://github.com/docker/docker/blob/master/docs/security/seccomp.md`

run as root (for instance to create the correct namespaces), which obviously represents a serious threat to the host system.

Compared to other solutions like LXC or CoreOS Rocket, Docker retains fewer capabilities. However, as it should be able to ship lots of different applications, the bounding set is still huge (14 capabilities), although significantly reduced with last versions.

3.3 Rkt

Rkt (pronounced Rocket) is a lightweight and modular container runtime. It is developed by the same team that maintains CoreOS, a minimalistic operating system that was designed for hosting Linux containers. CoreOS has been used for a long time in association with Docker for shipping containers, but Docker has become much complex and diverged from CoreOS's philosophy based on simplicity and reusability. Rkt comes as an alternative to Docker with a set of major improvements.

Rkt is meant to be simple, and consists of a simple executable rather than a background daemon. It has no real dependencies other than the Linux kernel and CPU architecture. Rkt implements open standards such as the Application Container specification (**appC**[8]) and the Container Network Interfaces specification (**CNI**[9]). It ships with many major Linux distributions and was designed to easily integrate with init systems such as **systemd**, and with cluster management tools such as **Nomad** or **Kubernetes**.

The Rkt execution is somehow similar to a real rocket, as it proceeds in different stages. Simply put, Rkt execution consists of running the **rkt** binary which creates the container's filesystem and then `/stage1/rootfs/init` executes, with the **rootfs** directory being the root of the new filesystem. A user trusted binary will then take care of setting up cgroups, namespaces and running processes and will run as root on the host. `systemd-nspawn` starts then and sets up external volumes, launches systemd as PID 1 which will in turn launch the remaining applications.

Rkt claims to be designed with security in mind. It introduces many security mechanisms that are missing in other technologies like Docker and LXC to name a few. Rkt uses granularity when it comes to privilege separation, making it possible to run routines as non-privileged user. Alongside with privilege separation, Rkt includes automatic SELinux configuration, making it impossible through MAC for a compromised container to get access to the host or to other containers. Moreover, Rkt integrates TPM (Trusted Platform Module)[10] to enhance cryptographic functions and hence system security.

But the Rkt project is still in its early stage and many security flaws still persist (277 open issues on GitHub Tracker at the time of writing[11]). Also, Rkt still does not integrate many isolation mechanisms like user namespaces (which remain experimental for the time being) or Seccomp filters.

[8]https://github.com/appc/spec/
[9]https://github.com/containernetworking/cni
[10]https://en.wikipedia.org/wiki/Trusted_Platform_Module
[11]https://github.com/coreos/rkt/issues

Chapter 4

Security threats

In this part, we will take a look at the main and most serious security flaws that have impacted LXC, Docker and CoreOS Rocket during last years.

4.1 Escaping containers

4.1.1 Privileged operations

Escaping a container means being able to do things directly on the host system, and this is kind of the worst case scenario. The first and most obvious way to do that is by taking advantage of privileged operations. For instance, some capabilities like `CAP_SYS_ADMIN` are catch-all capabilities: they allow many things like remounting filesystems, including cgroups, procfs and sysfs which are crucial to security enforcement as explained earlier in this paper. And some other capabilities permit use of local network or raw disks, which both offer a huge attack surface for container escaping.

4.1.2 Namespaces implementation

Generally speaking, implementation of Linux Namespaces is relatively immature. They were **added** to the Linux kernel and thus it is not *security by design*, which is often considered hazardous in security engineering.

The main issue comes from procfs (`/proc`), since this pseudo-filesystem is not namespaced. This brings many risks of container escaping or host modification (which could be helpful for leveraging other vulnerabilities), especially through `/proc/sys` that gives write access to kernel variables. The most dangerous files in `/proc` are listed below:

- **`/proc/sys/kernel/core_pattern`** allows arbitrary code execution as root[1].
- **`/proc/sys/kernel/modprobe`** contains the path to the kernel module loader (modprobe). Modifying it allows trivial privilege escalation or container escape by loading arbitrary kernel modules.
- **`/proc/sys/vm/panic_on_oom`** triggers kernel panic when OOM exception, thus if manually edited. It can cause Denial of Service attacks, which are

[1] http://seclists.org/oss-sec/2011/q4/158

not truly containers escape but still actions that should only be available to the host system.

- **/proc/config.gz** (when explicitly set to appear this way) is a compressed version of the running kernel settings. It is a huge information leak for a container, making it easier for an assailant to focus on vulnerable parts of the kernel.

- **/proc/sysrq-trigger**: the *magic SysRq key* is a key combination understood by the Linux kernel, which allows the user to perform low-level actions regardless of the system's state (useful to recover from freezes, or to reboot a computer without corrupting the filesystem). Should a guest be able to write into this file, it could trigger a reboot of the host, invoke kernel debuggers, or remount filesystems as read-only.

- **/proc/kmsg** and **/proc/kallsyms** : these files expose kernel ring buffer messages (output of **dmesg**) and kernel exported symbols (as well as their address locations). These informations can be used to easily bypass ASLR protection or simply obtain other sensitive kernel information.

- **/proc/<pid>/mem** is responsible for many vulnerabilities like local privilege escalation.

- **/proc/(kcore|kmem|mem)** are pseudo-files directly linked to memory and thus can possibly lead to information leak.

- **/proc/sched_debug** is a special file gathering information about process scheduling for the whole system. It fully bypasses the PID namespace as well as cgroups, and is readable by every user. This was exploited in Docker and fixed[2].

One can also locate containers rootfs, list running processes across all namespaces and exfiltrate lots of other sensitive datas.

Similar vulnerabilities come with sysfs (especially `/sys/kernel`) because it contains many special files which are readable by all users. They provide knowledge of network interfaces, hardware devices, exposure of kernel keyrings[3], etc.

The above files are almost all only root readable/writable so privileged Docker containers use read-only bind mounts to protect procfs for instance. LXC, Docker and Rkt also take advantage of MAC to enforce security of these files and directories (also preventing users from remounting procfs). Finally, the User namespace is definitely very precious to tackle this kind of issue as it allows having a root user inside a container, which has no privileges outside of it. However, this namespace is precisely an important source of vulnerabilities (see this one related to OverlayFS for instance).

4.1.3 Misconfiguration

Another big threat is (as often in security engineering) weak configuration, especially for cgroups and MAC policies. As we saw above, cgroups and MAC

[2] https://github.com/docker/docker/pull/21263
[3] https://github.com/docker/docker/issues/10939

are among the best options to fill namespaces security holes: they can easily be used to prevent users from accessing kernel ring buffers, procfs, sysfs, and from mounting devices or writing on them. Obviously, misconfigurations of these tools directly create serious vulnerabilities and facilitate container escaping. The problem here is that this is far from being easy. Setting up efficient MAC policies, whether it is with SELinux or AppArmor, is a true challenge. Same thing for cleanly creating cgroups. In both cases, it requires a complete understanding of the targeted infrastructure to establish whitelists, blacklists and other rules. Every single case is a new one.

Lastly, networking should be handled with care as some services can sometimes be bound by default to all interfaces, including for instance a bridge interface connected to containers. This would of course be a good way of increasing risks of containers escaping.

4.2 Cross-container attacks

Escaping techniques discussed above can have chaotic consequences, but they are difficult to conduct since they exploit design weaknesses in some Linux kernel features that get patched quickly by the community. A pretty interesting alternative would be attacking other containers on the same host. This has the merit of being service dependant, and vulnerabilities that affect services like web servers or database management systems are countless. Of course, this type of attacks cannot be achieved unless processes inside containers can communicate between them, and this is done in most cases through networking[4].

And again, default configuration of operating system-level virtualization solutions have many weaknesses, consequently providing malicious users with a large panel of attacks. Default network behavior in Docker, LXC and rkt allows the host to act as a network bridge to aggregate container-to-host or cross-container traffic. Such configuration weakness clears the field for many exploitations that are inherent to broadcast domains such as ARP spoofing[5]. This attack can be done when capabilities like `CAP_NET_RAW` are retained by containers or assigned by the container to certain processes within its own namespace.

We mentionned earlier that containers management and clustering could be done using orchestration systems like Kubernetes and Nomad. Network access can expose such tools to compromised containers, which may indirectly increase the possibility of enumeration and fingerprinting of other containers.

However, these examples only represent a tiny slice of the bigger set of cross-container attacks, and enumerating them all is not the purpose of this paper. Nevertheless, it is important to keep in mind that giving containers access to network is pretty equivalent to giving them direct access to the resources connected to it.

4.3 Denial of service

Denial of service is quite an often neglected attack vector because it does not necessarily lead to direct exploitation. It represents, however, a major security

[4]http://events.linuxfoundation.org/sites/events/files/slides/secure-lxc-networking.pdf

[5]https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1548497

threat due to its extreme effectiveness and easy apprehension. A clever attacker uses DoS alongside other attacks to cover tracks, mislead system administrators or gain more access into the system (e.g. retrieving network traffic at boot time which may contain credentials).

DoS attacks' huge impact on businesses and accessibility make them worthy of discussion, regardless of their final purpose. We will be discussing some DoS techniques in the context of operating system-level virtualization.

- **Infinite process replication** aka forkbombs. This technique consists of infinite duplication of a process in the sole purpose of monopolizing CPU and memory, leaving the system in an unusable state. This technique is probably the most popular and easiest among DoS attacks since it can be done via a single-line bash command such as `(:(){ :|:& };:)`

  If processes inside a container are not properly isolated, this attack could paralyze the entire host system.

- **Direct access to hardware** such as CPU, memory, devices or network can open the doors to infinite attacks. It is wise to limit what processes inside a container can do by associating them to cgroups and capabilities in accordance with the principle of least privilege.

  This kind of attack is simple to conduct. CPU monopolization, for example, can be done by running `dd if=/dev/zero of=/dev/null` inside a container, since these two devices are accesible by all users on the system.

- **Indirect access to hardware**, mainly via network. Such resources can be NFS mounted drives, central configuration management services or any other system on the same broadcast domain. Harmless and legitimate requests can, if generated in huge amounts, magnify log files (and thus use more disk space) or disrupt network activity due to excessive bandwith usage.

- **Attacks on randomness** to disrupt cryptographic functions on the host system. By default, devices `/dev/random` and `/dev/urandom` are accessible to processes inside a container. A conceivable DoS attack would consist in decreasing the host's entropy by issuing a huge amount of read calls to these devices. This can result in the host system being unable to generate cryptographic keys for example.

  The command `dd if=/dev/random of=/dev/null` is a simple implementation of entropy-related DoS attacks. We can notice an immediate decrease of available entropy:
  `watch -n 1 cat /proc/sys/kernel/random/entropy_avail`

  Note that GRsecurity patch fixes this problem by continuously increasing the size of the system entropy pool.

- **Per-user limits exhaustion** which is more of an implementation problem related to how Docker and LXC handle user namespaces: by default, `uids` in containers are obtained from the host `uid` by adding an offset to it, which happens to be the same offset for all containers. This means than processes running as `root` in a given container can affect the performances, if not deny service, of other `root` running processes in other containers.

Per-user exhaustion can be done when resources limiting techniques like `ulimits`[6] are used by the host to limit a certain number of parameters for a certain `uid`. DoS can result when hitting the limits for maximum open files, pending signals, user processes, file locks, POSIX message queues and so forth.

A proof of concept for blocking all signals sent by the system is available here, the next step being sending the maximum allowed signals to the queue.

4.4 Attacking the host container engine itself

As with hardware virtualization where an attacker could try to exploit the hypervisor, the software responsible for containers management is an interesting target which can be directly focused to escape a container or obtain higher access level.

For instance, CVE-2015-1335 allowed a LXC container to bypass AppArmor using a symlink attack. A similar vulnerability, CVE-2015-3627, focused the Docker Engine and libcontainer.

One more time, procfs created many vulnerabilities especially when unmounted, overwrote or overmounted in LXC and Docker. One example was CVE-2015-1334 which allowed for a fake procfs to be mounted with crafted SELinux or AppArmor profiles, thus breaking the MAC confinement.

Chroot was not spared, as shown by CVE-2014-9357 which introduced a privilege escalation vulnerability in Docker when dealing with archive extraction.

Last, LXC was impacted by CVE-2015-1331, resulting in a directory transversal flaw and thus allowing arbitrary file creation as the root user.

[6] `http://www.linuxhowtos.org/Tips%20and%20Tricks/ulimit.htm`

Chapter 5

What about other systems?

Linux is not the only operating system providing mechanisms of isolation. FreeBSD and Solaris also provide operating system-level virtualization which are discussed below. We will also take a quick look at Windows.

5.1 FreeBSD Jails

The Jail mechanism was introduced in FreeBSD 4.0. It is an evolution of *chroot jails*[1], because processes are sandboxed.

Jails are similar to Linux containers since they run off the same kernel but they are known to be more secured[2]: it is impossible for a process to break out of a jail on its own. Still, some vulnerabilities exist. For instance, an unprivileged user outside of a jail can take advantage of a privileged user inside of a jail to obtain root access on the host.

Jails rely on system calls like `jail_attach(2)` for instance, to attach a new process to a jail. They use copy-on-write and allow for updating multiple jails at once[3].

5.2 Solaris Zones

Zones[4][5] are a virtualization solution introduced in Solaris 10. They are midway between `chroot` and virtual machines. The operative mode consists of having a host operating system (called global zone) on which run `chroot`*ed* processes. These processes are special in the sense that they can run full operating systems. They cannot, however, have access to processes belonging to different zones. Zones are similar to Jails, as a number of system libraries are shared between the global zone and the other zones. Two types of zones are available:

- **Big zones**: This type of zones have their own filesystem.

[1]http://linux.die.net/man/2/chroot

[2]https://www.freebsd.org/doc/handbook/jails.html

[3]https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails-application.html

[4]http://docs.oracle.com/cd/E19253-01/820-2318/zones.intro-1/index.html

[5]http://www.unixmaniax.fr/wiki/index.php?title=Zones_-_Solaris

- **Sparse zones**: They share the `/usr`, `/lib`, `/sbin` and `/platform` directories with the global zone. They have the advantage of consuming less disk space.

Many mechanisms are implemented by Solaris Zones to ensure security such as credentials handling, privileges separation, privilege escalation prevention, role-based access control and so forth. But the fact that Zones are kernel-shared containers raises the same security issues discussed earlier with Docker, LXC and Rkt, and we will not discuss them any further.

5.3 Windows

As containerization is starting to become a standard for shipping applications and making DevOps life easier, Microsoft is trying to catch the tide and recently announced the release of Windows Server 2016[6] to be in the second semester of this year. This version introduces two types of container runtimes[7] alongside with Docker containers support. The two container runtimes are:

- **Windows Server Containers**: This type of containers achieves isolation using namespaces and process isolation.

- **Hyper-V containers**: These containers are not *real* containers in the sense that they have their own copy of the Windows kernel instead of sharing the host's. They provide more security by running containers inside virtual machines to prevent security breaches from impacting the host kernel. Hyper-V containers management can be done either using PowerShell or the Docker client. The optimized underlying structure (see Nano Server below) makes this type of containers combine flexibility with isolation.

  However, performance is not the key strength of Hyper-V containers as they are slower to start and larger in size than Windows Server containers.

Nano Server[8] is yet another feature announced by Microsoft to be shipped with Windows Server 2016. It is a minimized version of Windows Server built for the purpose of serving as a platform for containers, which was attained by removing the graphical user interface and 32-bits support.

[6] https://www.microsoft.com/en-us/server-cloud/products/windows-server-2016/
[7] https://msdn.microsoft.com/en-us/virtualization/windowscontainers/about/about_overview
[8] https://tinyurl.com/h6ppl8t

Chapter 6

Conclusion

Within a few years, operating system-level virtualization evolved from being an interesting substitute to full virtualization to being an essential tool: it is definitely useful for many companies and in many cases. It provides flexibility, efficiency, simplicity, compatibility and security.

However, containers are clearly not as secure as virtual machines. They rely on elegant but still immature features of the Linux kernel. All those mechanisms need to improve, as well as the many virtualization solutions implementing them, like LXC, Docker and Rkt. But still, there is absolutely no doubt that they represent an important part of the future of virtualization, even if they should be hardened and have to be thoroughly configured. Besides, those kernel mechanisms are pretty useful outside of containers. They provide isolation to a system for running potentially harmful applications.

On reflection, the best might lie in the middle of the two concepts. That is for instance why Intel is trying to do with its Clear Containers, which are designed as a hybrid of hardware and operating system-level virtualization in order to combine the best of both technologies.

Appendix A

Bibliography

The following references were used several times when writing this paper, which is why they did not appear as footnotes:

A.1 Videos

- A talk by Steven Ellis on resource allocation using cgroups.
- A talk by Jérôme Petazzoni on Linux containers.
- A video tutorial about cgroups by Justin Weissig (Sysadmincasts).

A.2 Web pages

- Wikipedia page for Cgroups
- Wikipedia page for Linux namespaces
- Wikipedia page for Seccomp
- Wikipedia page for SELinux
- Statistics about CVE for Docker

A.3 Articles and papers

- *Securing Linux Containers*, by Major Hayden from SANS Institute.
- *Understanding and Hardening Linux Containers*, by Aaron Grattafiori from NCC Group.
- A post by Brad Spengler from Grsecurity on capabilities.
- A blog article about Docker's security, by Daniel Walsh.
- *Abusing Privileged and Unprivileged Linux Containers*, by Aaron Grattafiori.
- *Virtualization and Namespace Isolation in the Solaris Operating System*, by John Beck, David Comay, Ozgur L., Daniel Price, and Andy T.

A.4 Presentations

- A presentation by Michael Kerrisk about Seccomp.