enabling the digital home

Boxes in Boxes

Virtualisation and Containerisation in the Context of Embedded routers

Wouter Cloetens & Wojtek Makowski

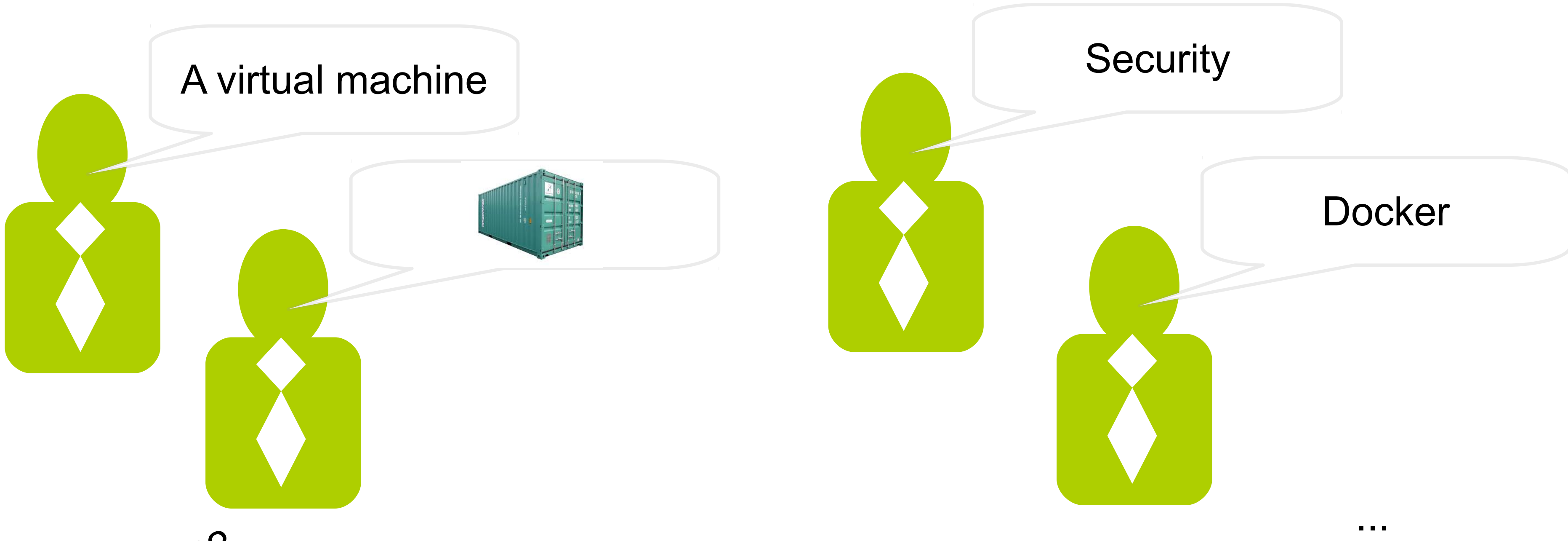
OpenWrt Summit
Berlin, 2016-10-13

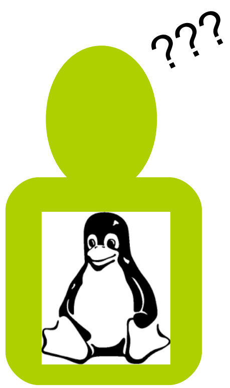
rev. 2

Part 1

Introduction to Containers

Everyone talks about containers…
… but what exactly is a Linux 'container' ?





This chapter aims to clarify the topic “containers” to an audience unfamiliar with the underlying technology.

Common misconceptions

- Linux containers are virtual machines

  No, they are not.

- Linux containers can run on any platform

  No, they can only run on a platform running the Linux kernel

- Running something in a container makes it completely secure

  No

- Containers will solve all my (security) problems

  No, although they are really powerful, they have limitations

Prerequisites for running Linux containers ?

- A (recent) Linux kernel with various security features enabled

  [preferably > Linux kernel 4.1]

- A framework to run containers : Docker, LXC, …

Container frameworks:

There are various technical frameworks that can be used for creating, managing and running « containers »:

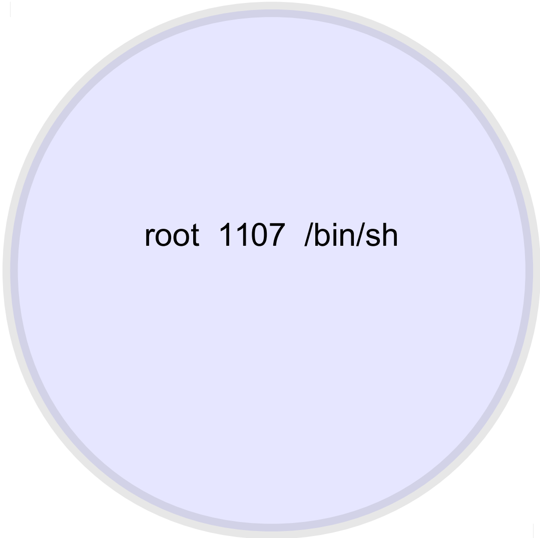
Docker, LXC, …

Each of these frameworks has its own features, but they all use the same underlying « security » mechanisms implemented in the Linux kernel :

Namespaces, Cgoups, Seccomp, Capabilities, user/group, chroot, apparmor, SELinux

⇒ *This presentation will describe the common security mechanisms used by containers and illustrate this with examples.*

Linux user space process running as root:



- Has access to complete file system
- Has access to all device nodes
- Can perform any kernel call
- Can reconfigure the network
- Can potentially consume all resources
- Can send signals to other processes (SIGKILL)
- …

Linux user space process running as root:

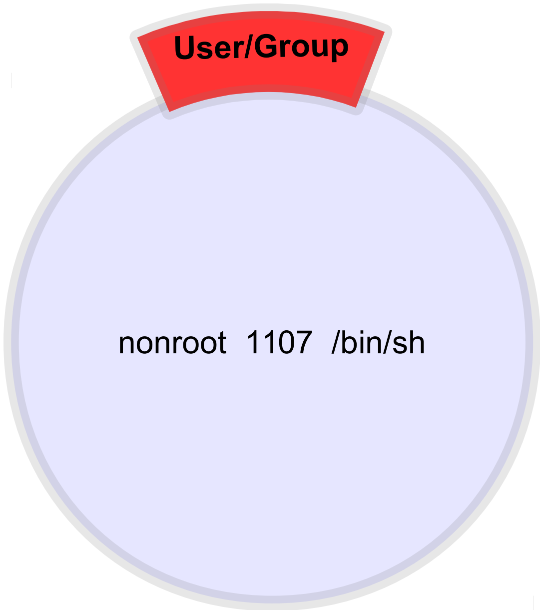
root 1107 /bin/sh

⇒ *If something goes wrong either due to a bug or due to a hacking attempt, nothing shields the rest of the system!*

```
rm -rf $(MY_UNDEFINED_VAR)/*
```

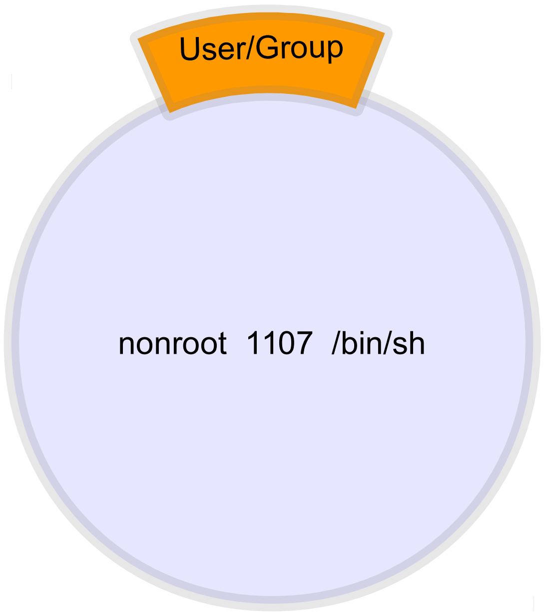
```
while (true) {
        switch(case){
            Up: break
            Default: continue
}
```

Step 1: Run the process as unprivileged user



- Discretionary Access Control: standard Linux (file) permissions and limitations are applied
- More limited access and control of devices
- More limited access and control of network devices
- Permissions of all kernel calls are checked prior to executing them
- Limited ability to send signals to processes owned by different users (e.g. !SIGKILL)

Can it still be made more secure?



setsockopt() is OK
...
Using raw socket and UDP protocol
...

YES!

⇒ *A process might still have potentially 'dangerous' capabilities!*

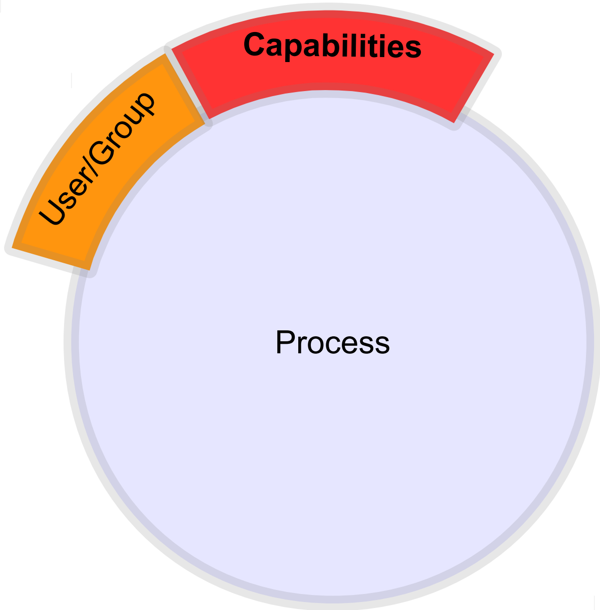
→ *set UID's on files*

→ *Open RAW sockets*

→ *Perform network configuration*

→ *...*

Step 2: Limit capabilities of the process

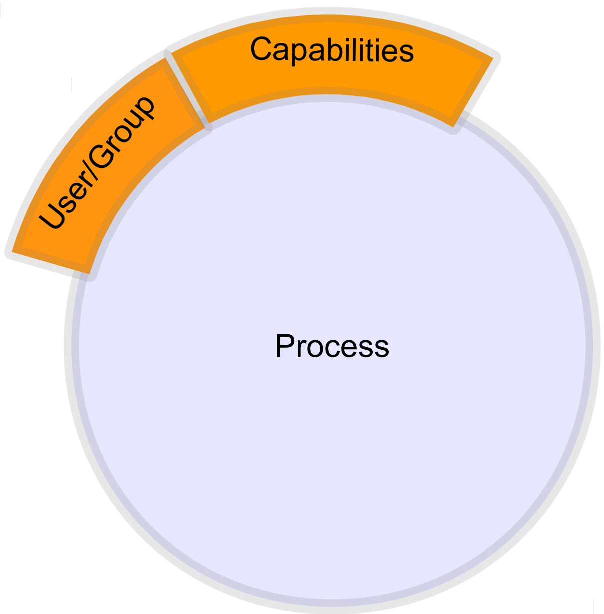


- Defined per thread
- Examples of available capabilities :

CAP_CHOWN	CAP_NET_ADMIN	CAP_SYS_ADMIN
CAP_FOWNER	CAP_NET_RAW	CAP_SYS_BOOT
CAP_FSETID	CAP_IPC_LOCK	CAP_SYS_NICE
CAP_KILL	CAP_IPC_OWNER	CAP_SYS_RESOURCE
CAP_SETGID	CAP_SYS_MODULE	CAP_SYS_TIME
CAP_SETUID	CAP_SYS_RAWIO	CAP_MKNOD
CAP_MAC_ADMIN	CAP_SYS_CHROOT	CAP_MAC_OVERRIDE

- Fewer capabilities = Better security

Can it still be made more secure?



YES !

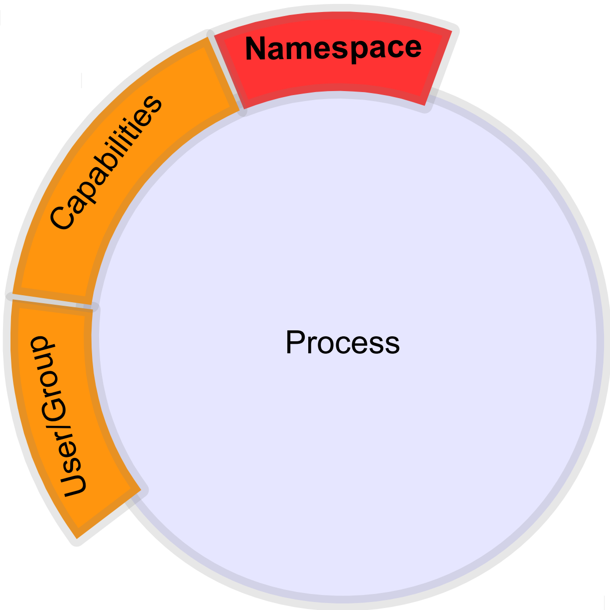
⇒ The process can still see all processes, network interfaces, mount points, IPC sockets, cgroups, users and UTS configuration!

```
>ps
PID USER        COMMAND
0    root          /sbin/init
```

```
>ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
```

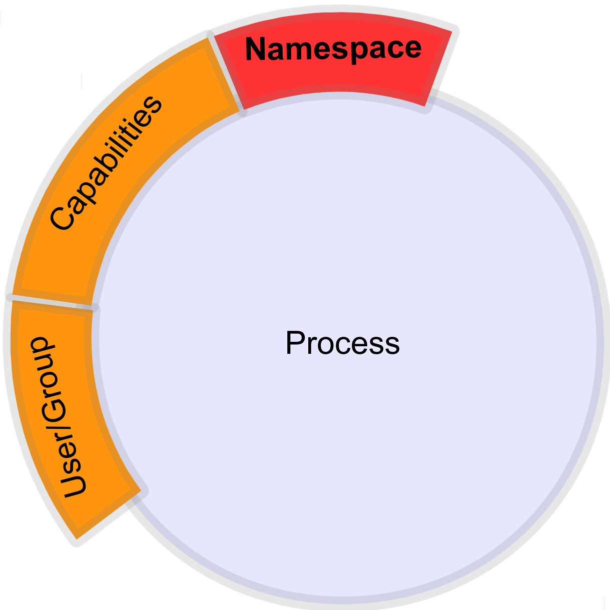
```
socket.error: [Errno 98] Address already in use
```

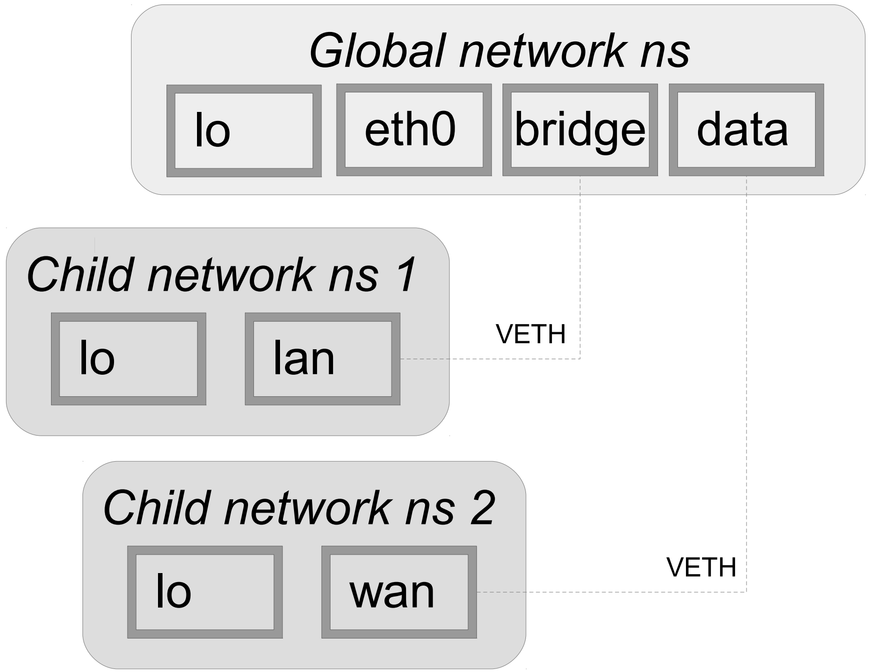
Step 3: Apply namespaces



- The kernel currently supports 6 different namespaces :
  - Network
  - PID/Process
  - Mount
  - User
  - IPC
  - UTS
- Provides isolation for each namespace type
- More namespace types are under development

Step 3: Apply namespaces: Network namespace



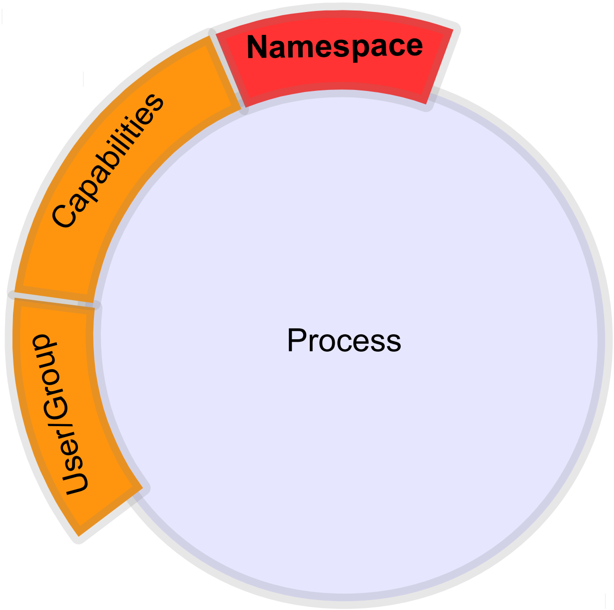


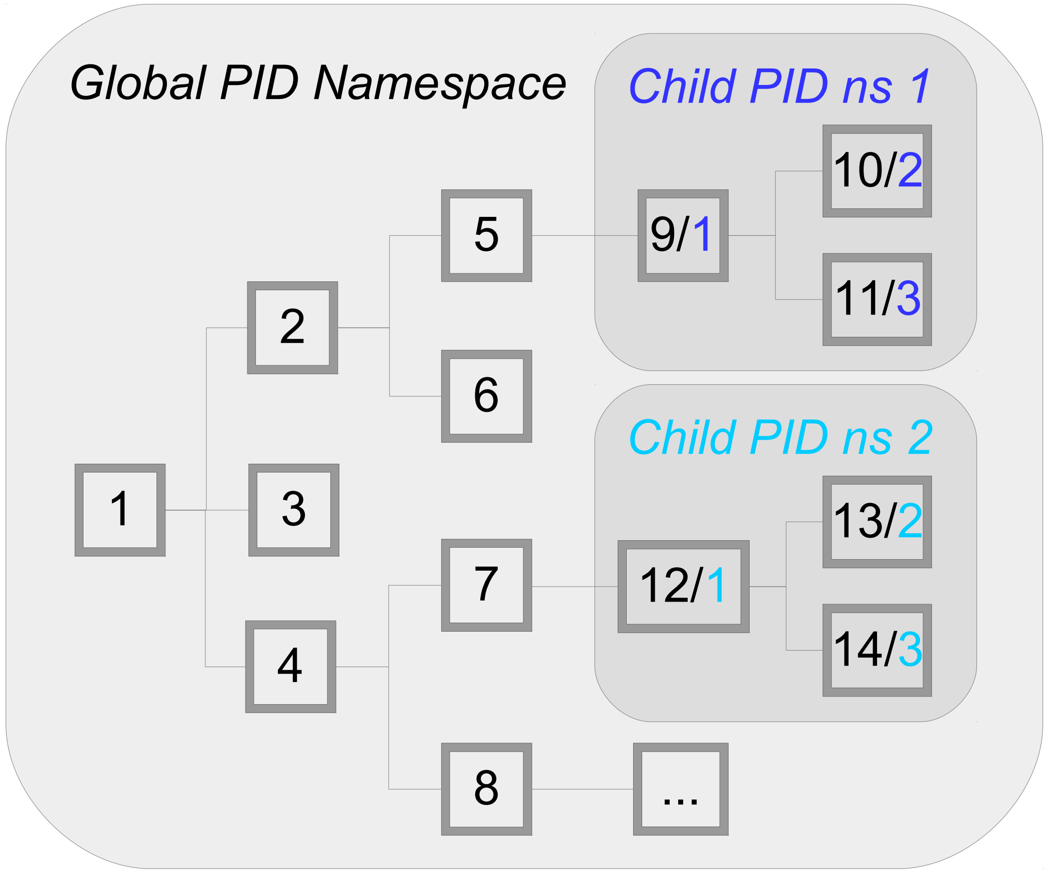
In global network namespace:
```
> ifconfig
lo      Link encap:Local Loopback
eth0    Link encap:Ethernet  HWaddr 20:47:47:73:e1:52
bridge  Link encap:Ethernet  HWaddr 10:38:1e:1a:ea:33
data    Link encap:Ethernet  HWaddr 22:50:e5:a0:f0:65
```

In child network namespace 1:
```
> ifconfig
lo      Link encap:Local Loopback
lan     Link encap:Ethernet  HWaddr 56:87:47:73:45:11
```

In child network namespace 2:
```
> ifconfig
lo      Link encap:Local Loopback
wan     Link encap:Ethernet  HWaddr 02:14:a5:e4:4f:15
```

Step 3: Apply namespaces: Process namespace





In global mount namespace:
> ps -A

PID	CMD	PID	CMD
1	init	8	rcu_sched
2	kthreadd	9	httpd
3	rcuos/0	10	cgi-bin
4	softirqd/0	11	php
5	kworker/0:0H	12	node.js
6	kworker/0:1H	13	/bin/sh
7	rcuob/0	14	mongodb

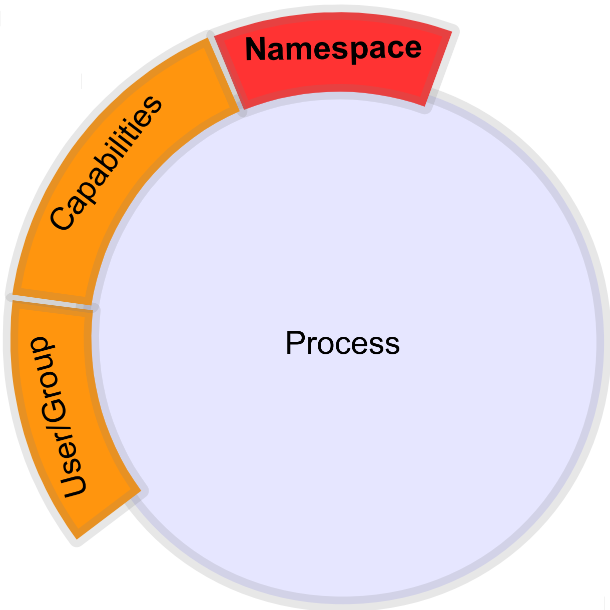
In PID ns 1:
> ps -A

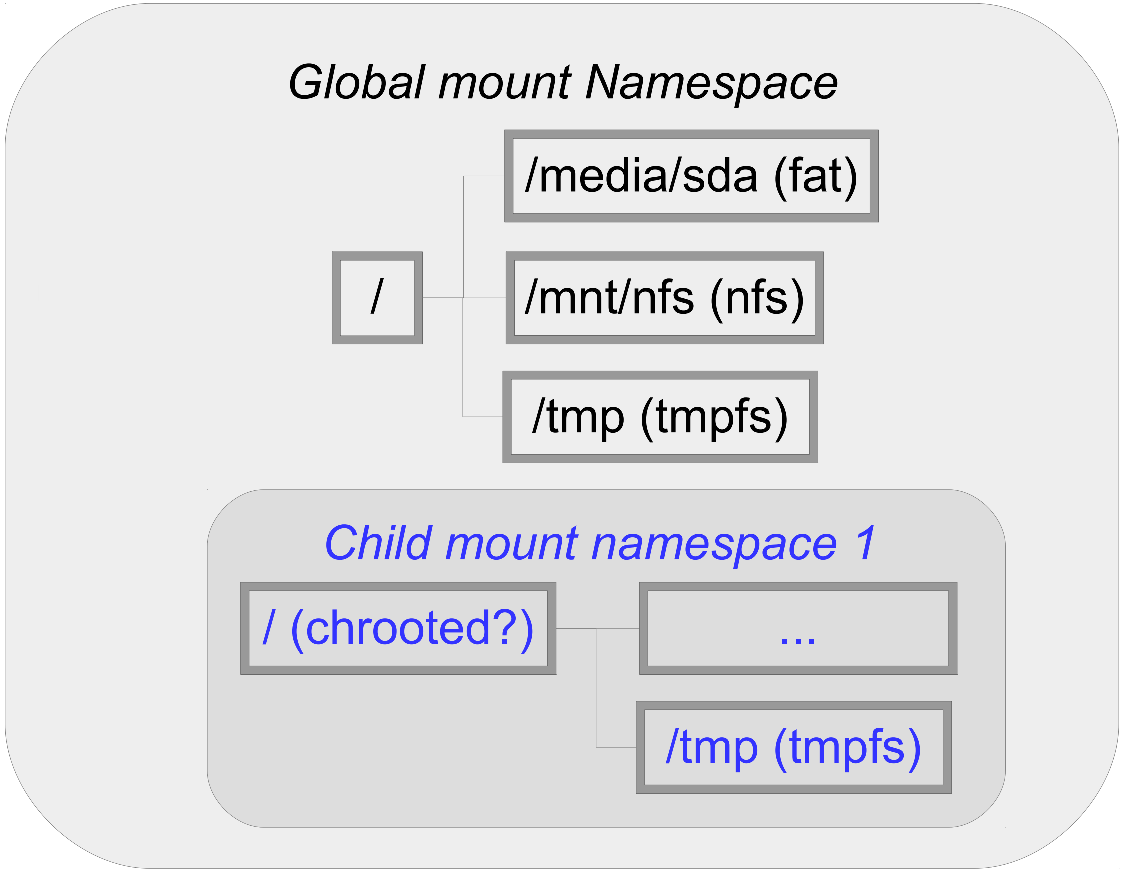
PID	CMD
1	httpd
2	cgi-bin
3	php

In PID ns 2:
> ps -A

PID	CMD
1	node.js
2	/bin/sh
3	mongodb

Step 3: Apply namespaces: Mount namespace

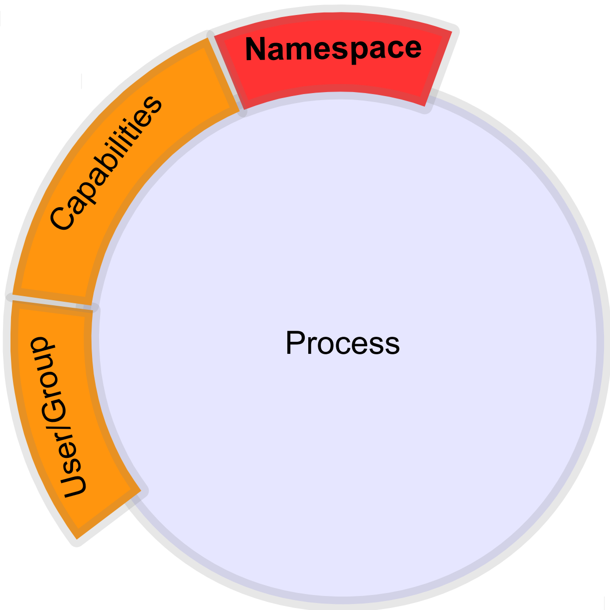




In global mount namespace:
> mount
/dev/sda3 on / type ext4 (rw,errors)
10.0.0.2:/home on /mnt/nfs type nfs(rw)
none on /tmp type tmpfs(rw)

In mount ns 1:
> mount
none on / type ext4 (rw,errors)
none on /tmp type tmpfs(rw)

Step 3: Apply namespaces: User namespace

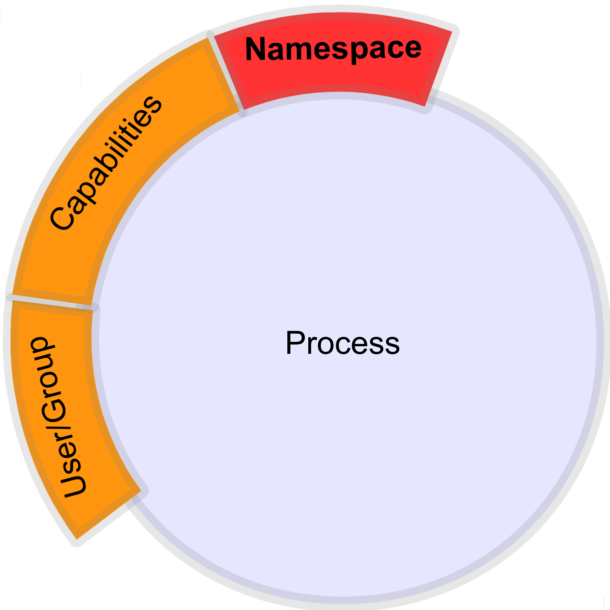


- User ID's (UID) and group ID's (GID) are specific per namespace
- Each UID/GID in another namespace maps onto a real UID/GID in the host
- Process can be privileged user within its own namespace (e.g. run with root-like privileges, without impacting the host)

In global user namespace:
>cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
user1:x:1:1:user1:/home/user1:/bin/bash
user2:x:2:2:user2:/home/user2:/bin/bash
virtuser1:x:3:3:virtuser1:/home/virtuser1:/bin/bash
virtuser2:x:4:4:virtuser2:/home/virtuser2:/bin/bash

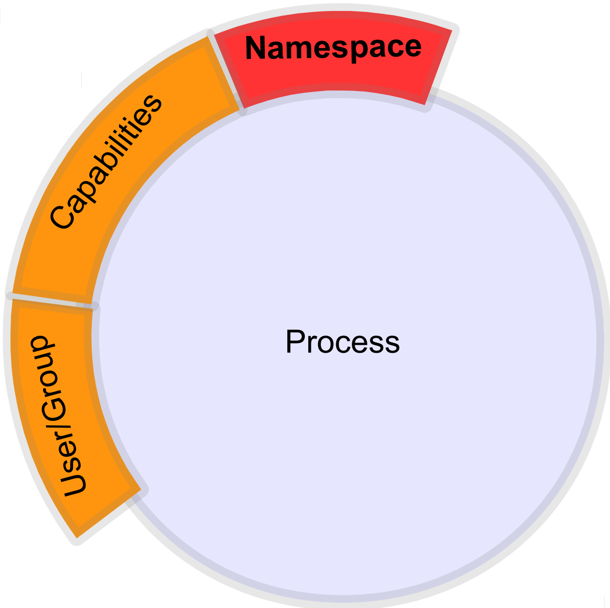
In different user ns 1:
> cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
service:x:1:1:service:/service:/bin/bash

Step 3: Apply namespaces: IPC namespace



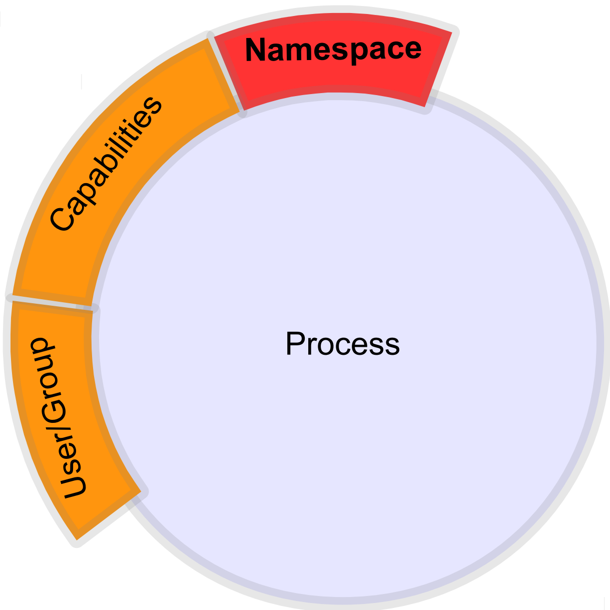
- Provides namespace separation for the following IPC mechanisms :
  - → System V IPC (svipc)
  - → POSIX message queues (mq_overview)
- Processes in different IPC namespace will not be able to communicate using the above IPC mechanisms.

Step 3: Apply namespaces: UTS namespace



- Provides namespace separation for some system identifiers:
  - → nodename
  - → domainname
- Processes in different UTS namespaces will be able to use the identifiers without impacting the rest of the system.

Can it still be made more secure?

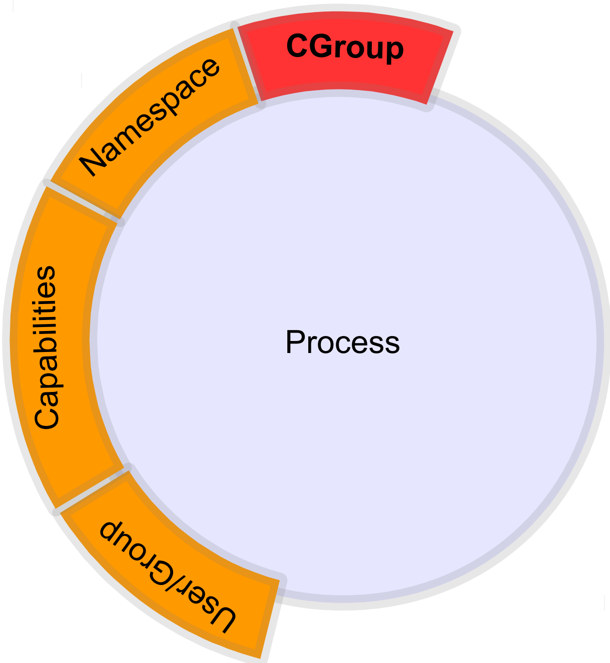


YES!

⇒ *How can you protect against abuse of system resources? (e.g. consuming all CPU/RAM)*

PID	USER	PR	NI	VIRT	RES	SHR S	%CPU	%MEM	TIME+ COMMAND
2064	user1	20	0	2765836	1.029g	115464 S	**97.0**	6.6	75:25.45 firefox
1337	root	20	0	488716	15.029g	31828 S	3.0	**93.4**	7:51.26 Xorg

Step 4: Apply CGroups

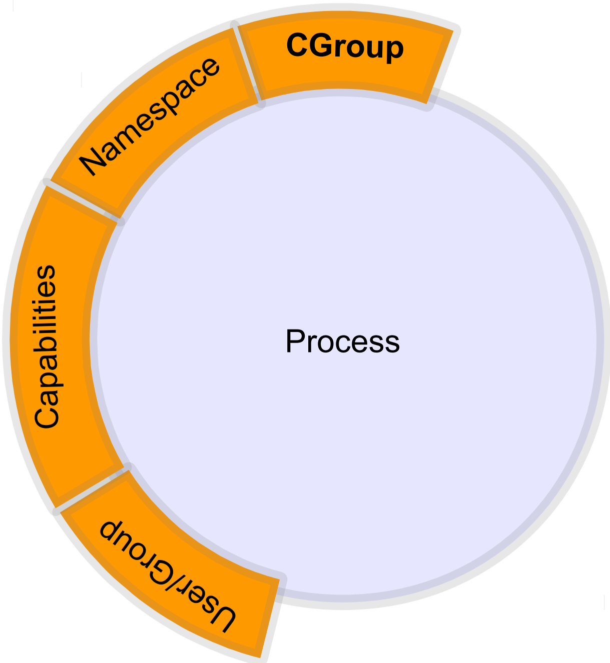


cgroup.memory.limit_in_bytes = 512M
(apply on a process that consumes all memory)

Out of memory:
Kill process 29957 (firefox)
score 366 or sacrifice child

- CGroup features :
  - Resource limiting – groups can be set to not exceed a configured memory limit, which also includes the file system cache
  - Prioritisation – some groups may get a larger share of CPU utilisation or disk I/O throughput
  - Accounting – measures a group's resource usage, which may be used, for example, for billing purposes
  - Control – freezing groups of processes, their checkpointing and restarting
- CGroup restrictions can be applied to both individual processes and groups of processes

Can it still be made more secure?



YES!

⇒ The process still has access to the complete file system :

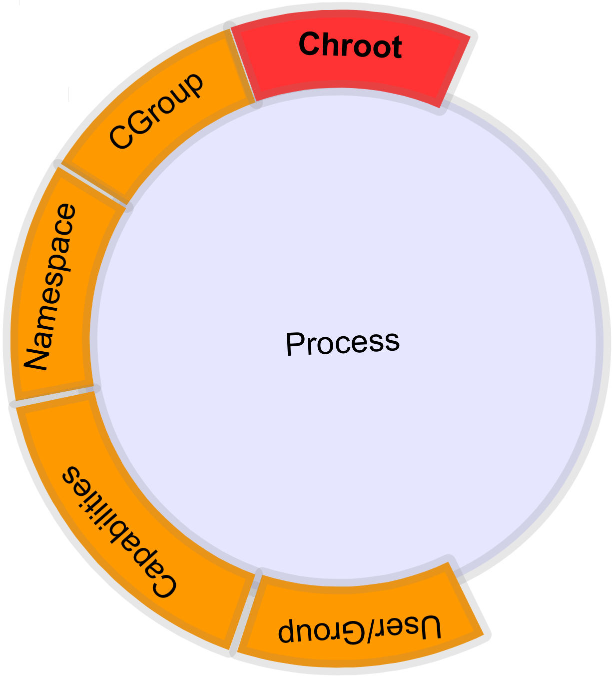
→ One misconfigured permission…

→ It can still launch most binaries

Drwxrwxrwx user group /etc/config

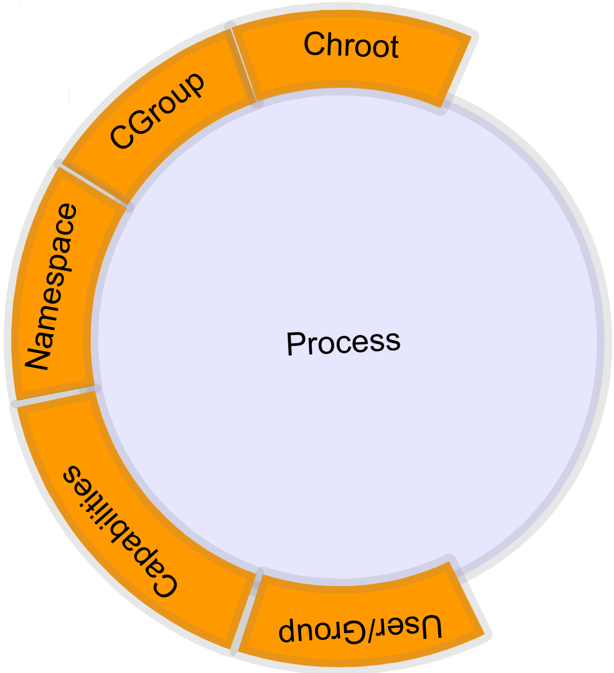
/bin/curl
/bin/sh
/bin/...

Step 5: Chroot: Limit access to the file system



- Use chroot to only allow access to files that are really needed (libraries, executables, configuration files, data files)
- From a process point of view: only a very small file system is visible.
- Only root can escape a chroot environment
- Different possibilities to implement the alternative file system: hard links, overlayfs, etc.
- Considered by most people as a 'hardening' feature, not a 'security' feature as you can still potentially escape the chrooted file system.

Can it still be made more secure?



YES!

Chroot shields the file system, but is this enough ?

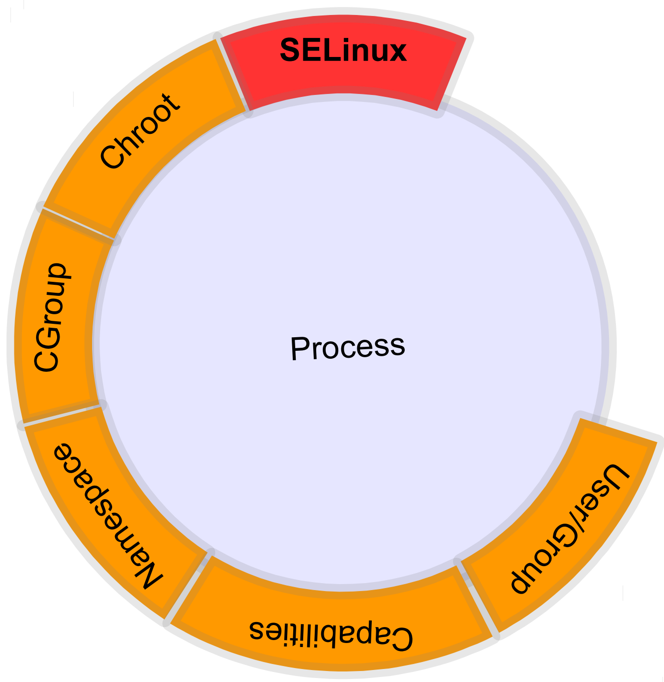
→ one could still escape a chroot

→ difficult to maintain

⇒ The kernel supports several alternative ways to configure fine-grained access control per process, using Mandatory Access Control:

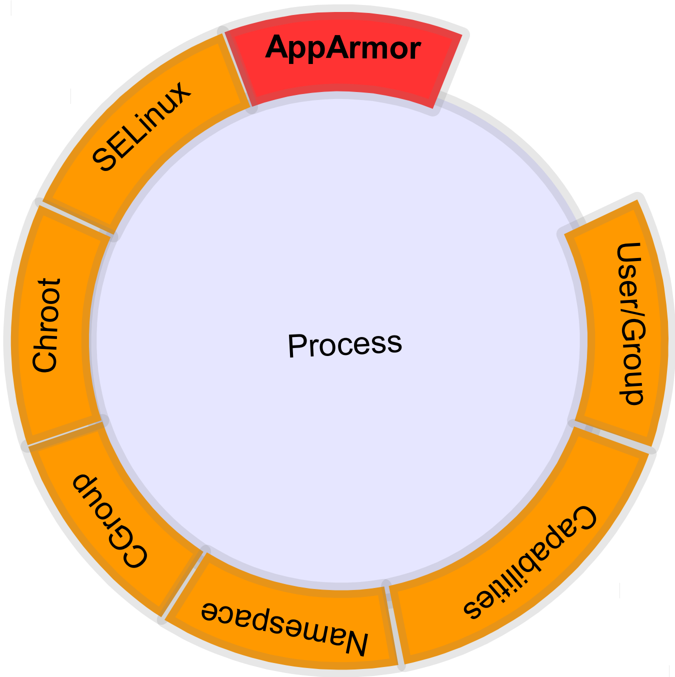
- SELinux
- AppArmor
- Smack
- Tomoyo

Step 6A: SELinux



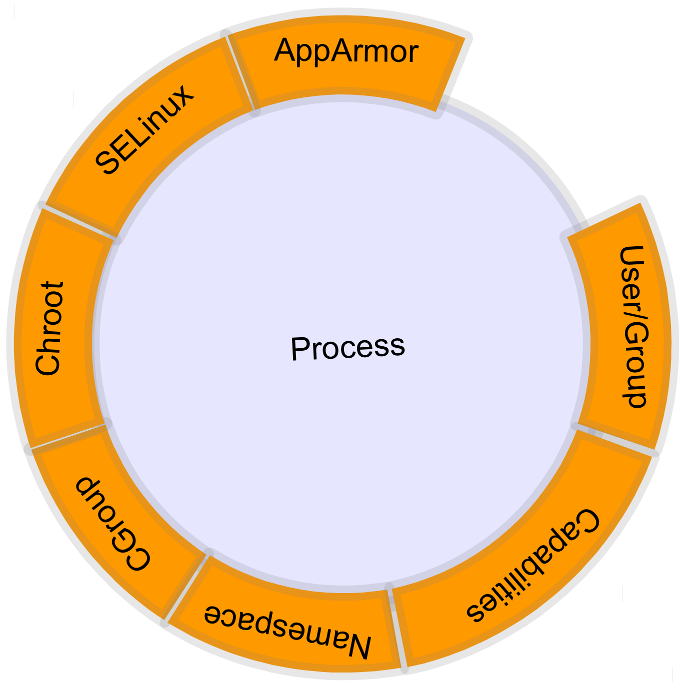
- Access control per process
- Extends traditional Linux security mechanisms
- SE Rules allows (ultra) fine grained control
- Applies labels to files
- Perceived by many people as complex to learn and maintain
- Can be used in combination with chroot

Step 6B: AppArmor



- Alternative to SELinux
- Access control per process
- Extends traditional Linux security mechanisms
- Per-program profile
- Profiles allows fine grained control per file path
- Automatic learning mode
- Can be used in combination with chroot

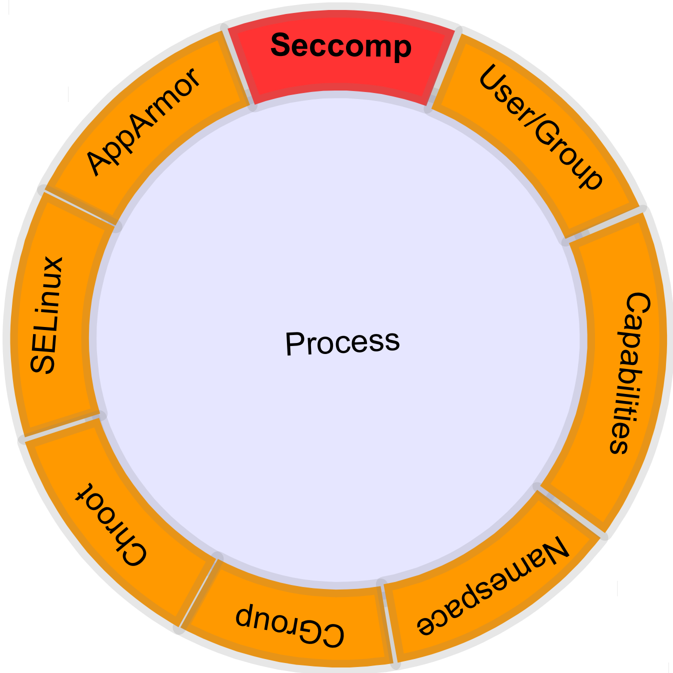
Can it still be made more secure?



YES!

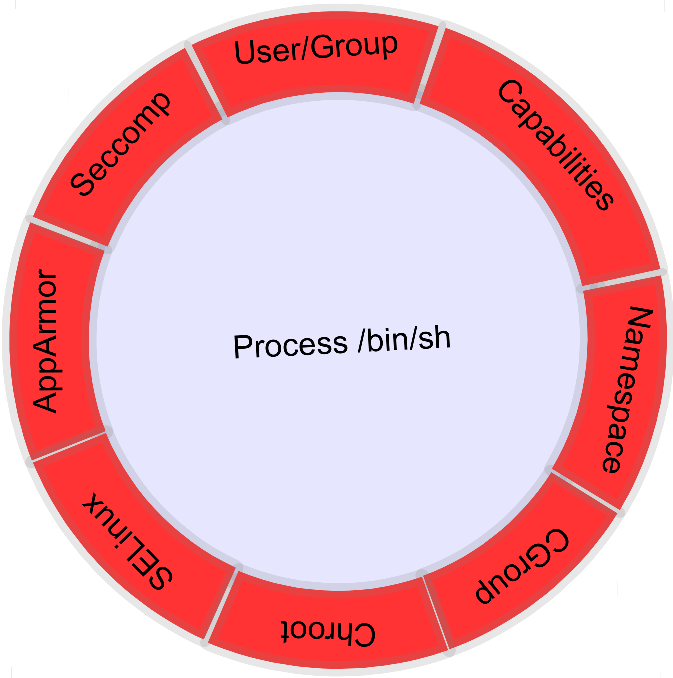
⇒ Depending on the application you want to run, you might want to consider putting it into a 'sandbox'.

Step 7: Seccomp (secure computing mode)



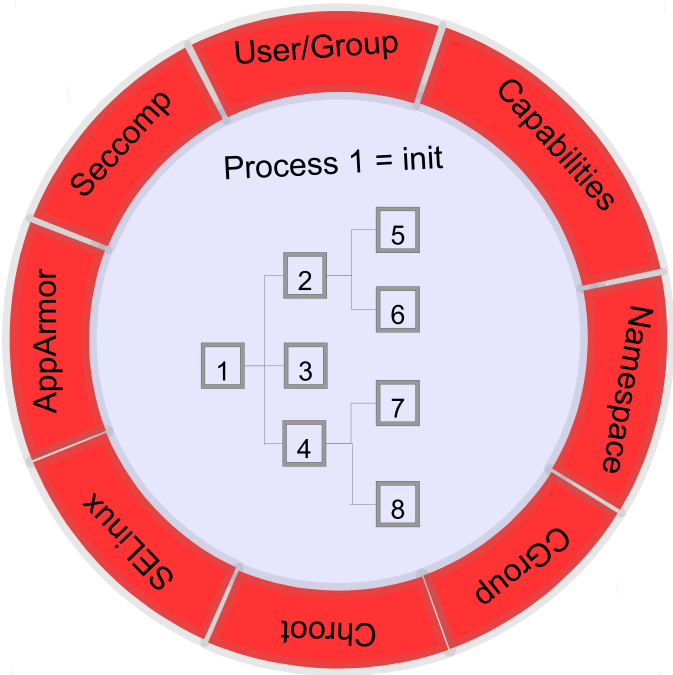
- Not applicable to all processes
- One way transition to 'secure' mode
- Allowed system calls :
  - exit
  - sigreturn
  - read/write on open file descriptor
- Any other system call results in SIGKILL
- This imposes a 'limitation', not 'virtualisation'

Imagine Process = /bin/sh, what do you see ?



- A kind of 'virtualised' shell:
  - Virtual users (user Namespace)
  - Only one active process (PID Namespace)
  - Virtual network devices (network Namespace)
  - Limited availability of system calls (Capabilities, Seccomp)
  - Limited set of files visible (chroot)
  - Limited set of files accessible (User/Group, SELinux/APPArmor)
  - Limited use of system resources (CGroup)

Imagine Process = init, what do you have?



- A lightweight 'virtual machine':
  - Using a shared kernel
  - Having bare-metal speed
  - Having its own file system
  - Having limited access to system calls
  - Having limited access to devices
  - Having limited access to system resources

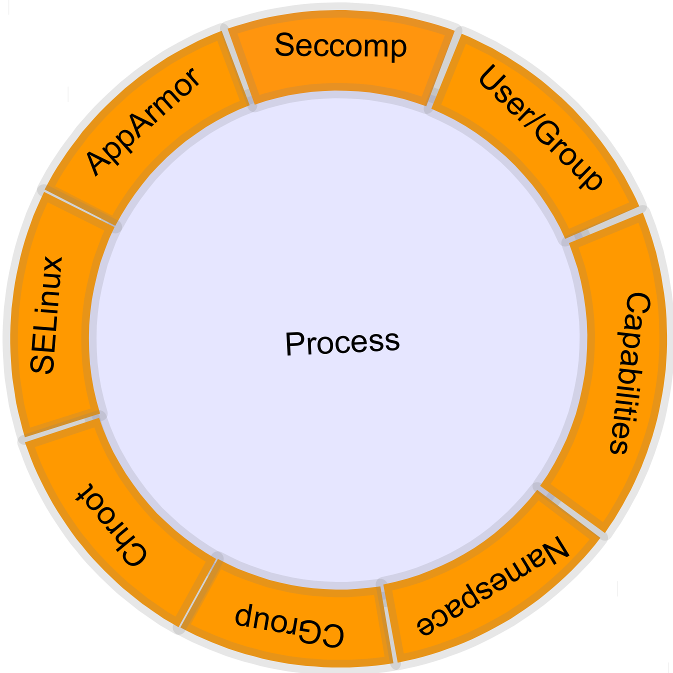
What added value do container frameworks like Docker/LXC offer?

- One configuration file per process or set of processes
- User space tools to manage the configuration
- User space tools to start/stop/freeze/load 'containers'
- Utilities to (remotely) manage/deploy containers
- ...

Part 2

Containers vs. Virtual Machines

Can it still be made more secure?

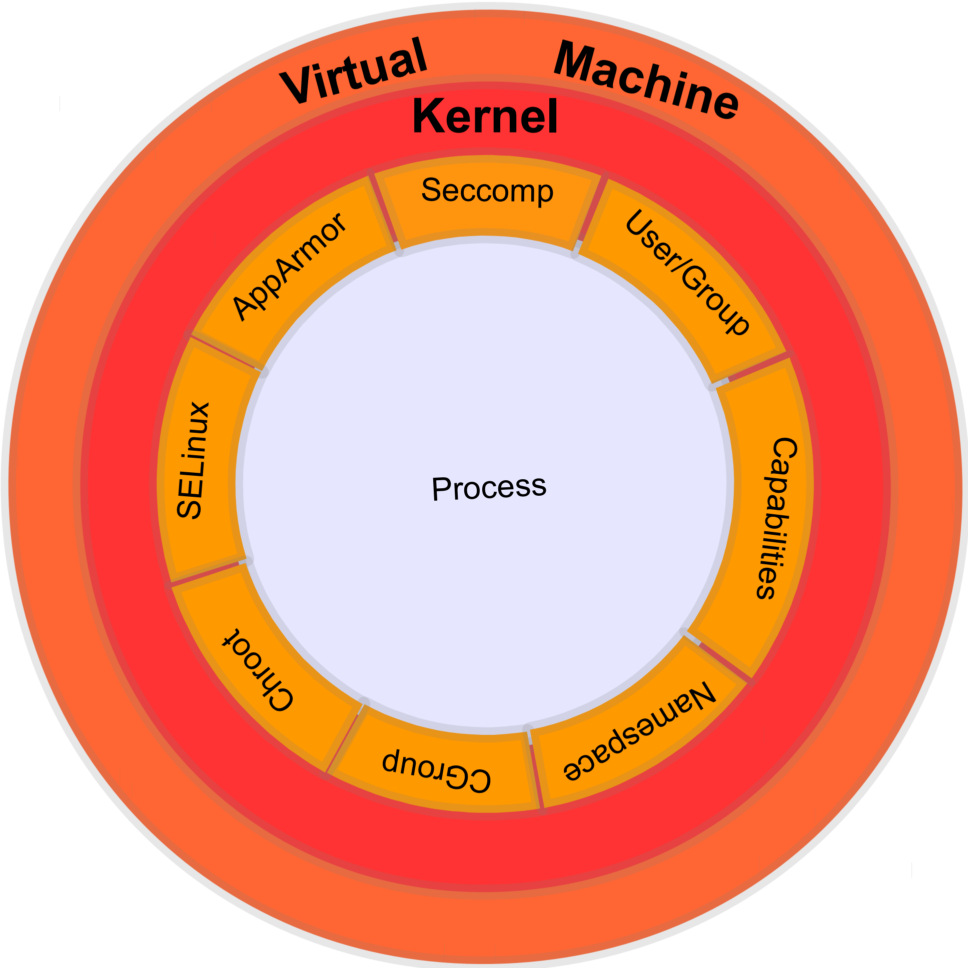


YES!

⇒ We're still sharing the same kernel...

- the same device drivers
- risk of currently unknown security holes allowing privilege escalation
- CGroups accounting may not be completely accurate
- all hardware resources *must* be supported by the same kernel

Step 8: Hardware Virtualisation



Containers or Hardware Virtualisation?

	Benefit	Containers	HW Virtualisation
1	Split firmware into several independent entities	**	**
2	Ensure runtime independence	*	**
3	Ensure life cycle independence	*	**
4	Secure HGW's baseline services entity	*	**
5	Open entities for 3rd party and "unfriendly open source"	*	**
6	Allow different kernel versions to run in parallel		**
7	Allow different OS to run in parallel		**
8	Leverage Cloud-based on-demand HW resources (CPU, memory, storage)	*	*
	Essential Characteristics		
1	Memory Overhead	**	
2	Performance	**	
3	Boot time	**	*
4	No specific HW requirements	**	
5	Ease of implementation	*	
6	Ease of SW migration	*	**

Containers or HW Virtualisation? → **Both**

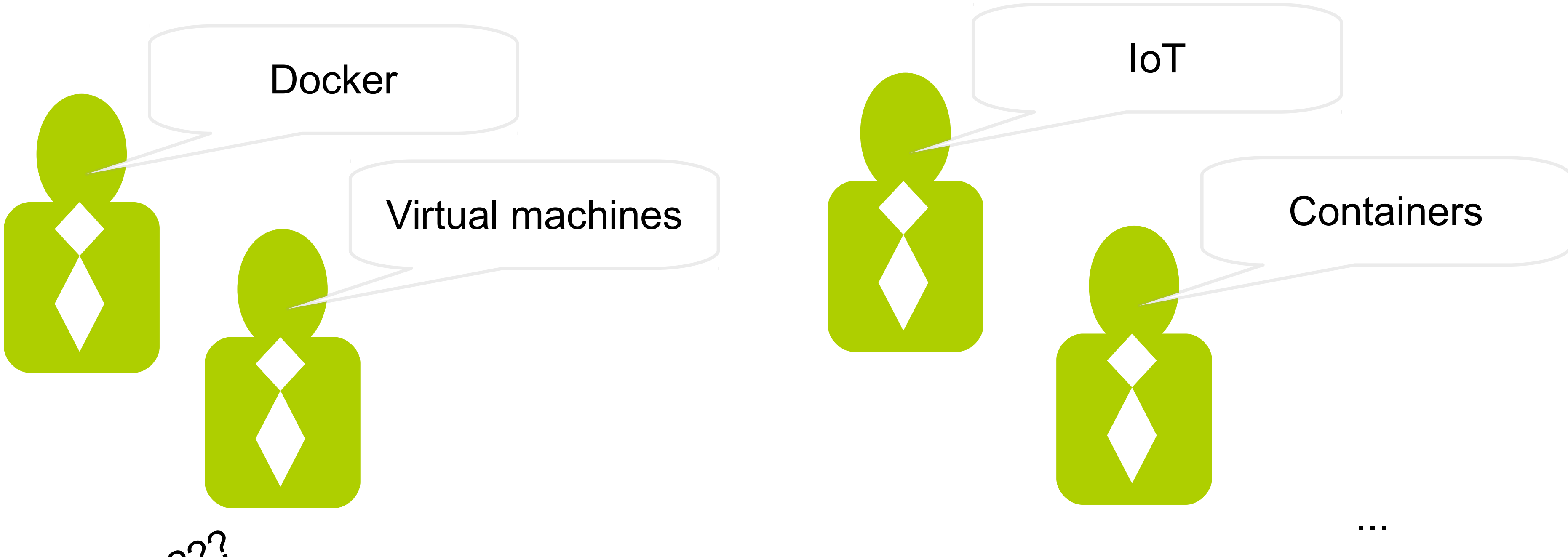
- Containers
  - address HGW challenges partially... but in a **very efficient** way
  - “lightweight virtualisation” - **available now**
- HW Virtualisation
  - addresses more challenges... but with additional HW/SW burden/cost
  - **not yet available**... but on some router SoC roadmaps

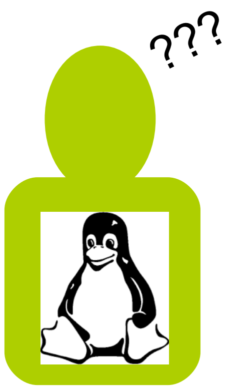
- Conclusions
  - Containers are an obvious step forward – available now
  - **Mix of containers and HW virtualisation** looks like a **future winning solution** addressing the challenges in the most efficient way
    - Example: an open application environment in a dedicated “open” Virtual Machine with each app packaged in its own container
  - To become reality, HW Virtualisation requires effort from chipset manufacturers

Part 3

Containers on Embedded Platforms

Which 'hype' are we going to implement today?



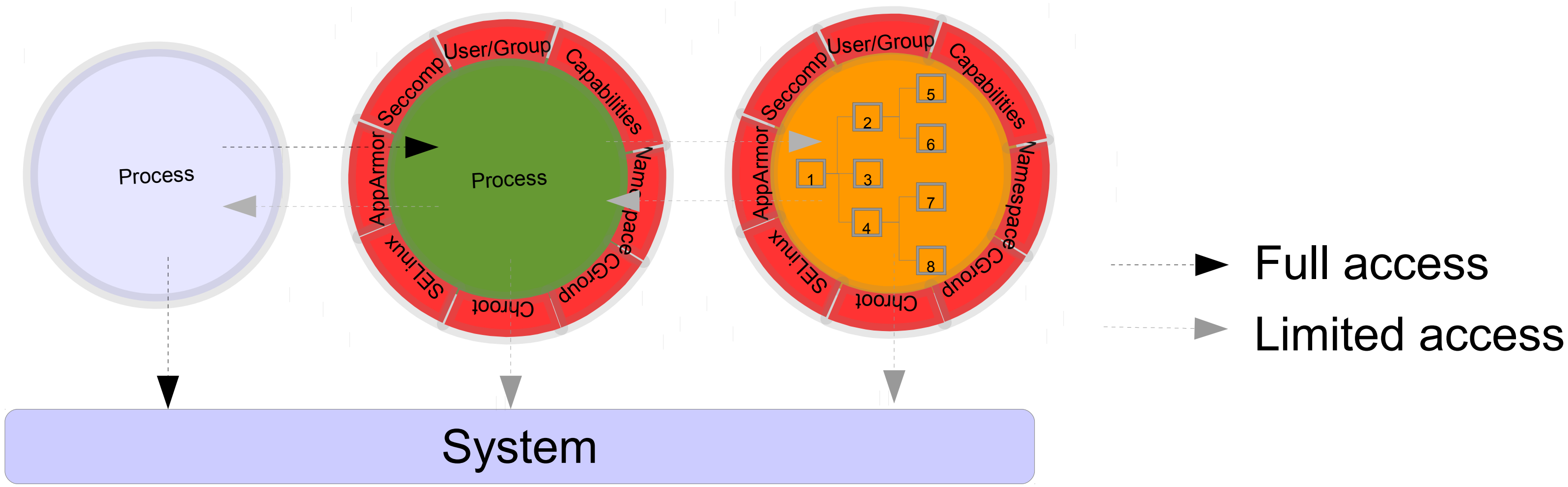


This chapter aims to clarify the use cases of “container” technology in embedded products.

Implementing containers on embedded platforms

There are 2 possible **ways** of **implementing** container technology:

- **Single process** per container
- Containers as **virtual machines**



Technical goals of container technology

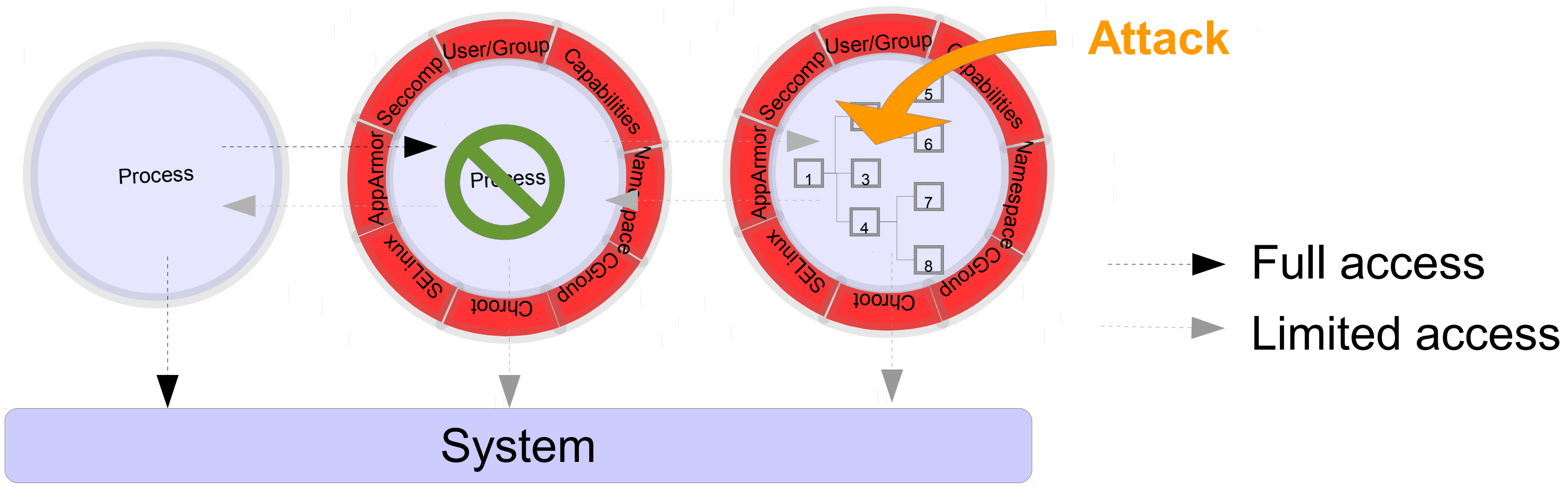
These can be used to achieve 2 main **goals**:

- Increase **robustness** of a software stack

  → Running processes in a contained environment **shields the system** from bugs within those processes.

- Improve **security** of a software stack

  → Running a process in a contained environment isolates processes from the other system components, **reducing the attack surface**.



Embedded use cases of containers

This means you can use containers for different **use cases**:

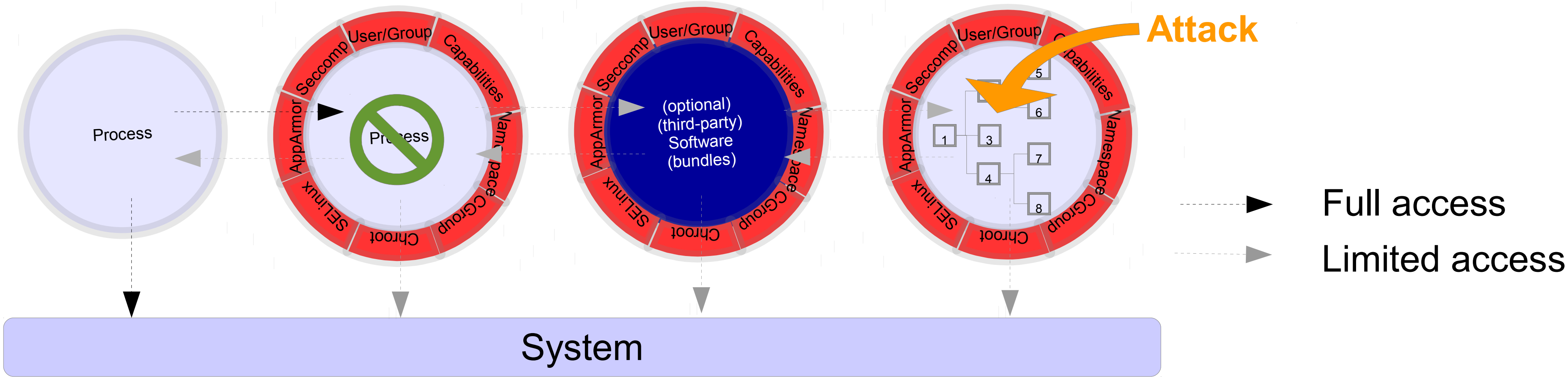
- Improve robustness of existing software stack.

  → e.g. Isolate specific processes that might impact system stability
- Improve security of existing software stack.

  → e.g. Secure processes that can be reached externally
- Integrate (third-party) (optional) software (bundles).

  → Put binary third-party deliveries into sandbox environment

  → Enables app-store like concepts (of complex software stacks)



Single process vs virtual machine approach

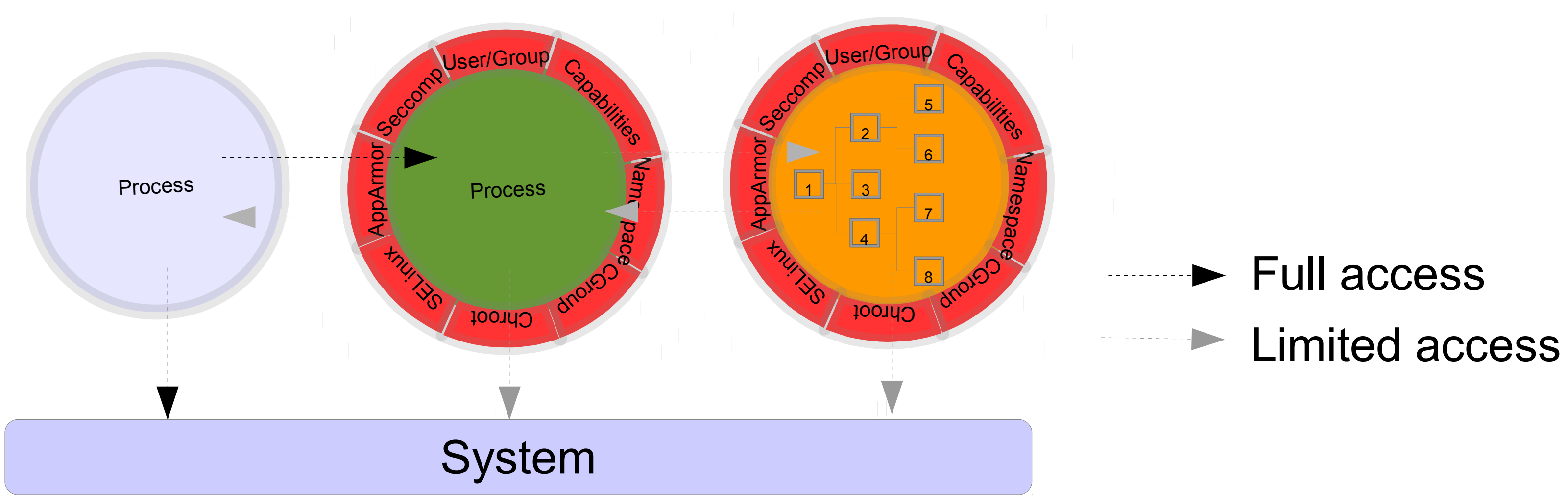
Single process per container approach:

+ Configuration can be really tailored for that executable.

+ Optimal security and shielding

- Runtime dependencies are an issue

- No 'logical' packaging (think app-store)



Single process vs virtual machine approach

Virtual machines per container approach:

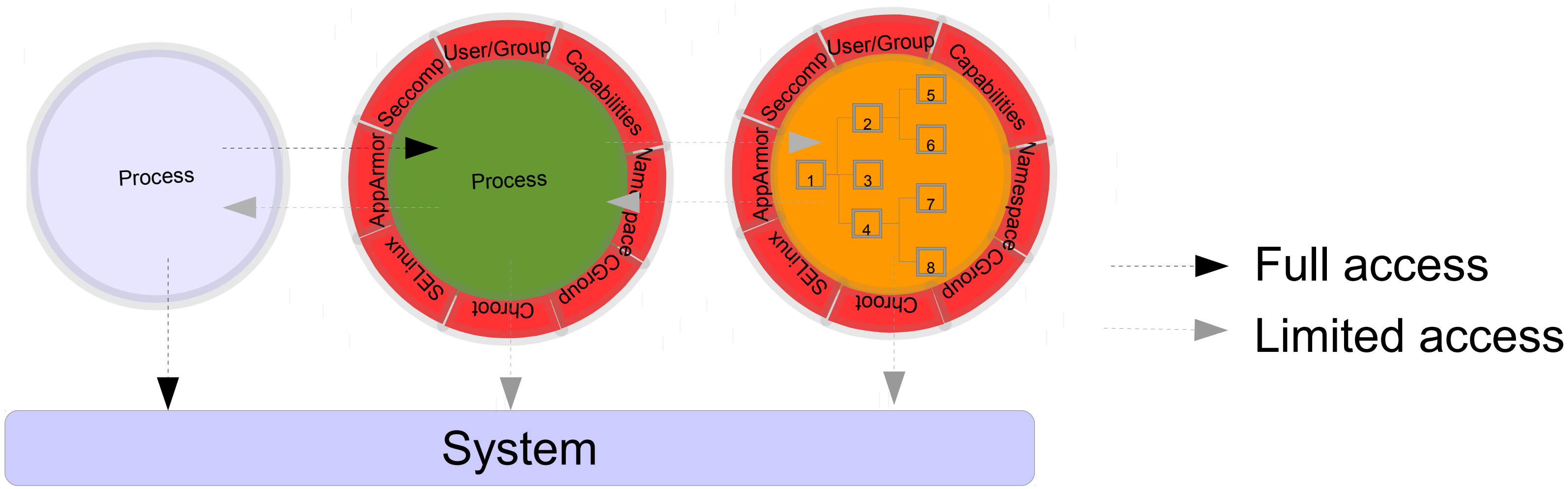
+ Possibility to create 'logical' packages

+ Potentially resolves a lot of run-time dependencies

- Applications within container are not protected against each other

- Shielding depends on the weakest link

→ e.g. if one process requires privileges to mount a USB disk, all the other elements in the container have the same ability



Side note

Why use container technology (Docker/LXC) and not use the kernel API's directly in the source of the different services ?

⇒ Adapting each service is time consuming and error prone.

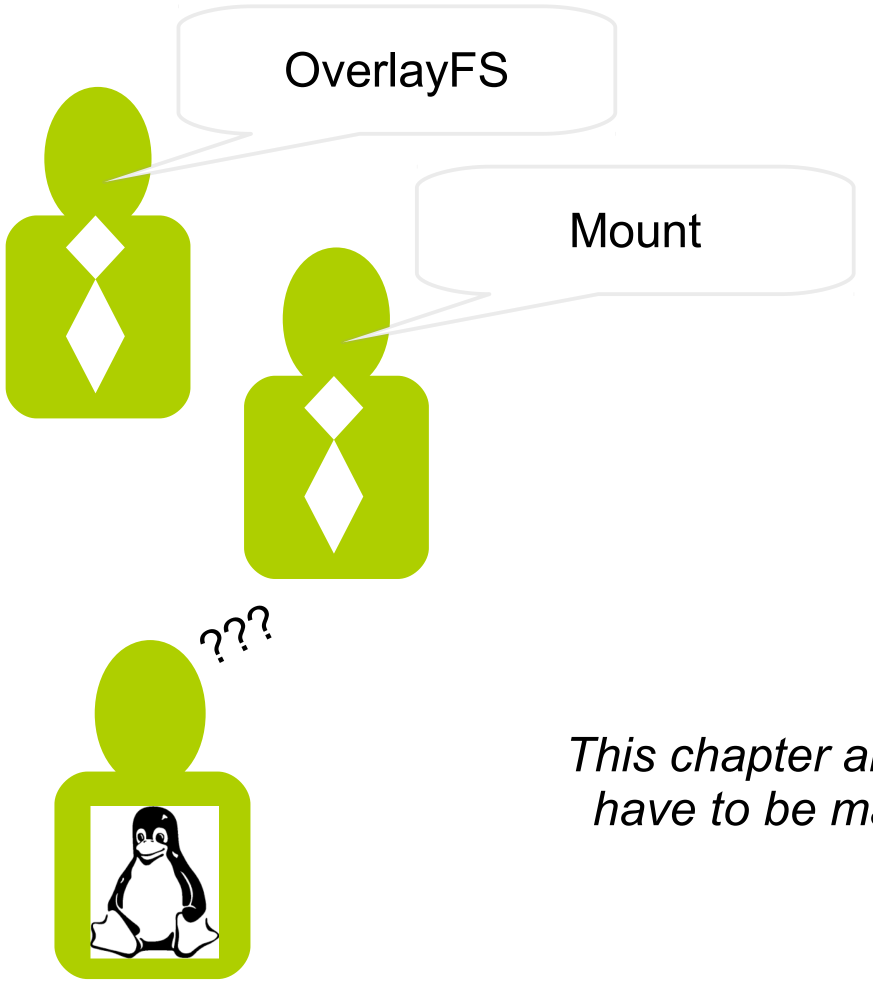
⇒ Container frameworks offer a uniform interface to configure and manage a large variety of different settings.

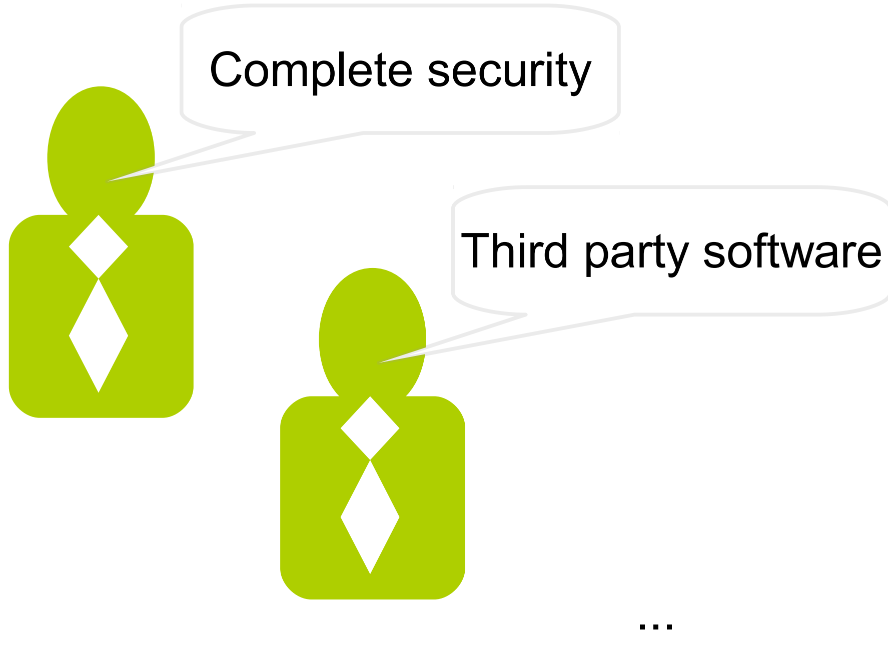
⇒ The uniform interface makes it easier to integrate into an existing build system.

Part 4

Container Design Choices

Just run it inside a container, right?





This chapter aims to give an overview of design choices that will have to be made when integrating "containers" in embedded products.

Analyse your setup

- Which Linux kernel version?
- Which container facilities apply?
- What are the device limitations? (CPU/RAM/flash)

Which Linux kernel version?

Kernel	Mount NS	UTS NS	IPC NS	PID NS	Network NS	User NS	CGroup v1	CGroup v2	APP Armor	SE Linux	Seccomp
2.4.19	X										
2.6.0	X									X	
2.6.12	X									X	X
2.6.19	X	X	X							X	X
2.6.24	X	X	X	X	X		X			X	X
2.6.36	X	X	X	X	X		X		X	X	X
3.8	X	X	X	X	X	X	X		X	X	X
4.5	X	X	X	X	X	X		X	X	X	X

Which container facilities apply?

- The user/group policy
- Capabilities
- Namespaces: Mount, UTS, IPC, PID, Network, User
- CGroups
- Chroot
- SELinux/APPArmor
- Seccomp

Some configuration options are easy to configure:

- The user/group policy
- Capabilities
- Namespaces: Mount, UTS, IPC, PID, Network, User
- CGroups
- Chroot
- SELinux/APPArmor
- Seccomp

Others are more difficult to use and configure: Network - File system - Users

- The user/group policy
- Capabilities
- Namespaces: Mount, UTS, IPC, PID, Network, User
- CGroups
- Chroot
- SELinux/APPArmor
- Seccomp

Configuring your network: types

• empty	Only loopback interface in container
• phys	Moves the host interface to the container namespace
• vlan	VLAN on top of interface
• none	Shared network namespace between host and container
• veth	Virtual Ethernet interface
• macvlan	Multiple MAC/IP on the same physical interface • 3 modes : VEPA, private, bridge

Configuring your network: empty/phys/vlan

- Empty: When container does not need networking
- Phys: Might be used for complex use case e.g. Ethernet over USB etc.
- VLAN: Use cases limited in consumer environment

Configuring your network: none

+ All host interfaces reachable in container

+ No performance penalty

+ Easy to listen on host IP address

- Less secure

    * container can access all interfaces

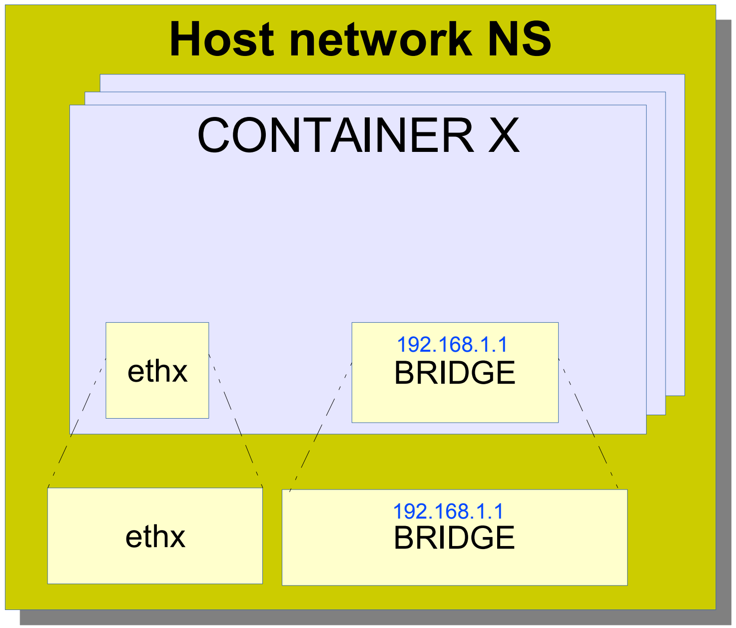
    * container can block ports

    * containers & firewall

    * ...

- Might be complex for third parties developing code in the container

- Network service configuration is shared; how to block port usage in container?



Configuring your network: veth

+ Typically connected to software bridge on host

+ Container can access host + other containers

+ Containers can share a network namespace

- Security: container can access host

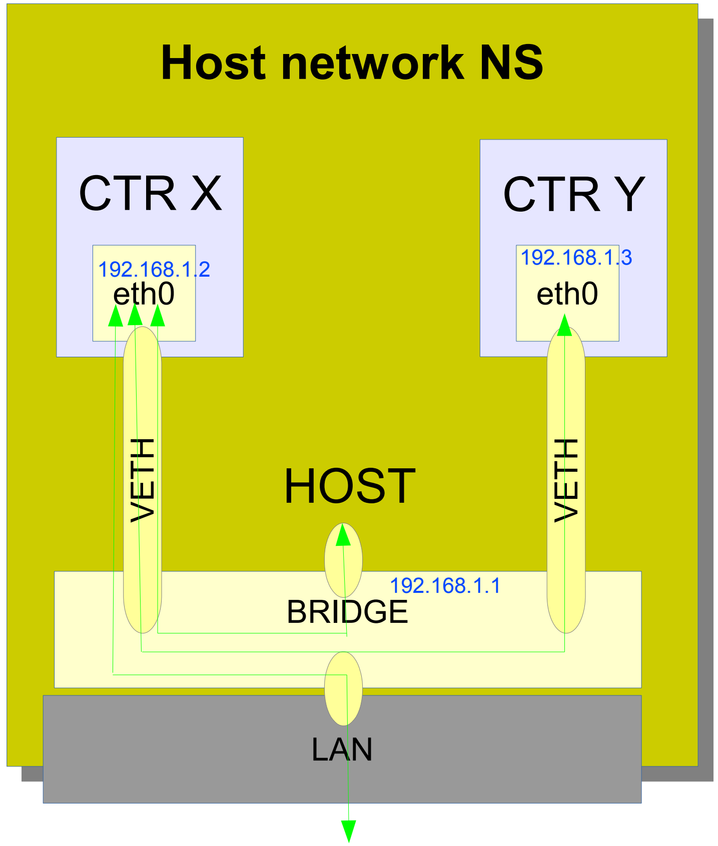
- Security: container can access other containers

- Security: firewall configuration needed

- Performance penalty caveats

- Port forwarding needed in case a service should be reachable on host IP

- Traffic from the container will have a different IP address.



Configuring your network: Macvlan VEPA

+ Packets directly written to interface, no performance penalty

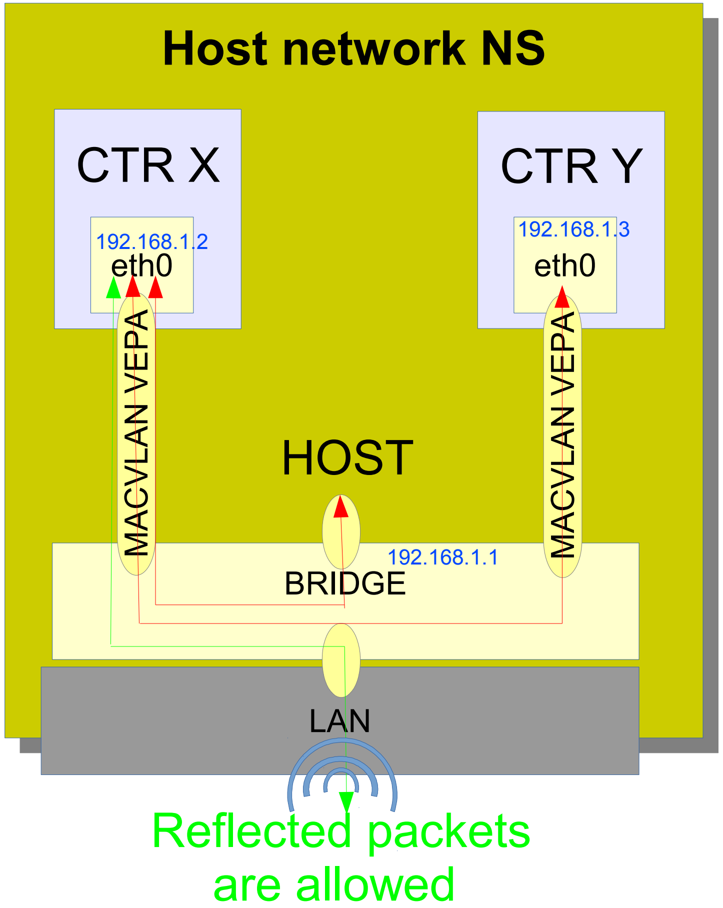
+ Container can not access host + other containers unless packets are reflected or hairpin mode is used

+ Security : everything is blocked by default

- Only on (local) interfaces where you can add extra MAC's and IP's dynamically

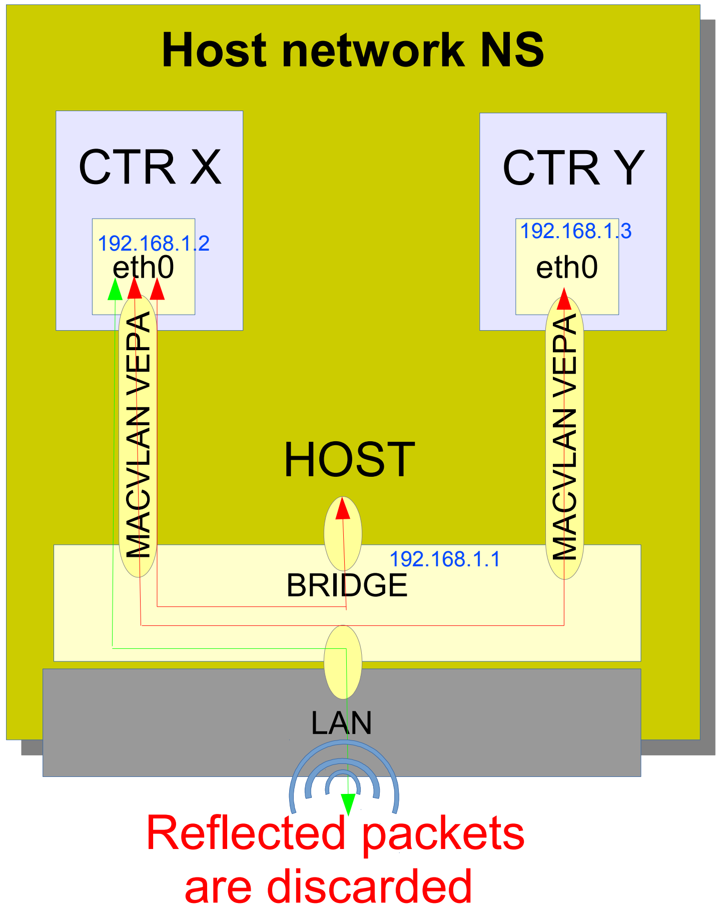
- Port forwarding needed in case a service should be reachable on host IP

- Traffic from the container will have a different IP address.



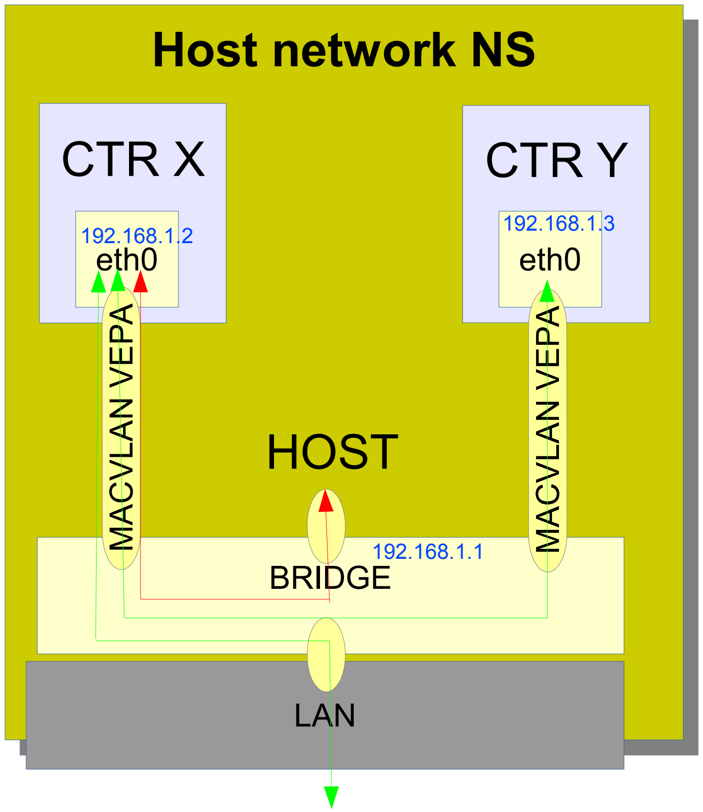
Configuring your network: Macvlan private

Exactly the same as VEPA, but reflected packets are discarded



Configuring your network: Macvlan bridge

+ Packets directly written to interface, no performance penalty
+ Container cannot access host unless hairpin mode is used
- Security: containers can talk to each other
- Only on (local) interfaces where you can add extra MAC's and IP's dynamically
- Port forwarding needed in case a service should be reachable on host IP
- Traffic from the container will have a different IP address.



Defining your file system: RootFS requirements

- Different (often conflicting) requirements pop up:
  - As small as possible (flash + memory)
  - As performant as possible
  - Stand-alone capabilities
  - Security:
    - Preferably read-only / executable
    - Signature check at load
    - Correct file permissions
  - …

Defining your file system: RootFS technologies

- Container technologies:
  - File permissions
  - Chroot
  - SELinux/APPArmor

- Closely linked technologies:
  - Squashfs
  - Hardlinks
  - OverlayFS (or alternatives)

Defining your file system:
RootFS **for securing** an existing software stack

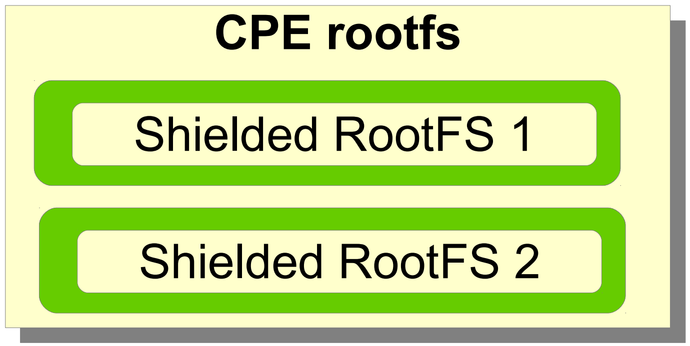
- Considerations for securing the rootfs
  - Rootfs is defined at build time of host
  - Hardlinks are possible, but watch out for file permissions!
  - Possible to **optimise** for **flash** size (no duplicate files)
  - Possible to **optimise** for **RAM** size (no duplicate files cached)
  - **Native performance** possible (no cache misses)
  - Fine tuning of SELinux/APPArmor possible (and really needed if you want to do it right)

Defining your file system:
RootFS **for software bundles**

- Considerations for creating the rootfs for software bundles
  - Rootfs is **not always** defined at build time of host
  - Hardlinks are possible, but more difficult to use!
  - **Stand alone capabilities** are more important, but have a cost on RAM/flash/performance.
  - Fine tuning of SELinux/APPArmor possible

Defining your file system:
RootFS implementation options

- Rootfs shared with host, shielded by SELinux/Apparmor
  - \+ For real security SELinux/APParmor mandatory anyway
  - \+ No performance/memory impact
  - \- File permissions in combination with user namespace?
  - \- Containers can see all files
  - \- Difficult to maintain



Defining your file system: RootFS implementation options

– Chroot to a hardlinked rootfs per container

  + No performance/memory impact

  + Cleaner view from container point of view

  - Hardlinks typically created at build time

  - Hardlink file permissions combination with user namespace?

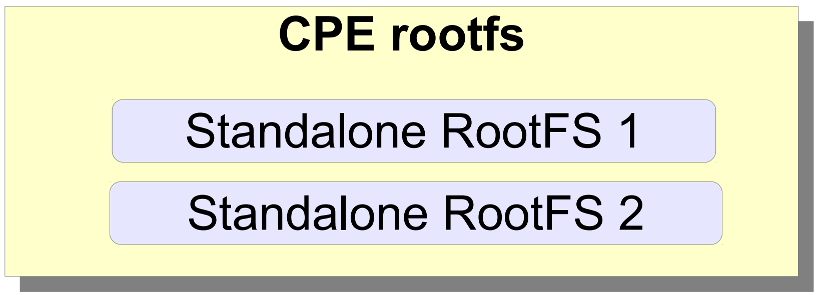
  - For real security SELinux/APPArmor needed anyway

CPE rootfs

hardlinks1	Standalone RootFS 1
hardlinks2	Standalone RootFS 2

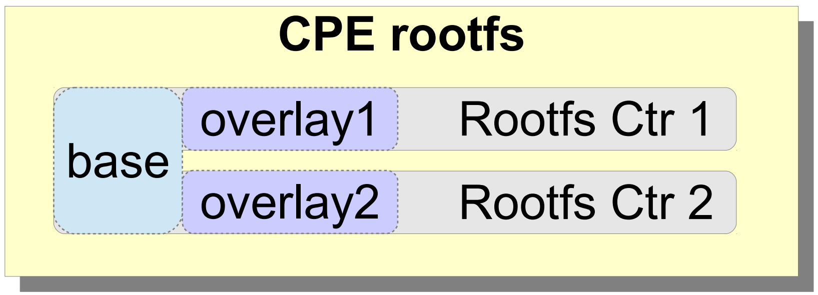
Defining your file system: RootFS implementation options

- Chroot to a duplicated rootfs per container
  - + No issues with file permissions in combination with user namespace
  - + Cleaner view from container point of view
  - + Minimal dependency on host rootfs
  - - Performance/memory impact
  - - Note: For real security SELinux/APPArmor needed anyway



Defining your file system: RootFS implementation options

- Share a base rootfs, use overlayFS as addon
  - + Solves memory issue of using separate rootfs partially
  - + Minimal dependency on host rootfs
  - + Standardisation of container rootfs possible
  - - OverlayFS needs to be managed
  - - Note: for real security SELinux/APPArmor needed anyway



Defining your file system: RootFS implementation options overview

	Memory usage	Performance	Standalone	Manageability
Stand alone package	--	-	++	+
Shared base package	+	-	+	+
Shared with host – hard links	++	++	--	-

→ Which approach to choose depends on the goal you want to achieve

Defining your file system: Tmpfs

- Requirements:
  - Read-write - preferably no execute
  - Limited in size

- Possible solutions:
  - Tmpfs mount in container configuration

    ⇒ Easier to manage
  - Tmpfs mount in container firmware

    ⇒ More freedom for container developers, tmpfs is counted into cgroup limitations

Defining your file system: Persistent storage

- Requirements:
  - Read-write - no execute
  - Limited in size

- Possible solutions:
  - Bind mount R/W directory from host
  - Bind mount R/W directory from file (containing a FS) from host
  - Mount native partition in flash
  - Provide key/value store service on host
  - OverlayFS

Defining your file system: Persistent storage

- Bind mount R/W directory from host:
  - + No runtime overhead
  - + Used by Docker, which allows you to 'name' a volume and share over different containers
  - - Migration of data is difficult
  - - No limitation possible on size:
    - * user quota works in Linux per owner/group, not per directory
    - * https://code.google.com/p/fusequota/ but this is GPLv3
- Bind mount R/W directory from file (containing a FS) from host:
  - + Easy to limit in size
  - - Possibly stability issues (how to control which data was really written to the flash)
  - - Migration of data is difficult
  - - Runtime overhead

Defining your file system: Persistent storage

- Mount native partition in flash:
  - \+ No runtime overhead
  - \+ Easy to limit in size
  - \- Dynamic management of flash partitions needed
  - \- Migration of data is difficult
- Provide key/value store service on host :
  - \+ Easy to limit in size
  - \+ Migration of data can be easily managed in centralised way
  - \+ Can be used to push default settings to container as well
  - \- Runtime overhead
  - \- Complicated
  - \- Not easy to use
- Overlayfs (can be used in combination with the previous techniques)

Defining the user namespace

- Map virtual user ID's with root-like privileges to real user ID's on the host (« unprivileged containers »)

  map real user 100000 → 165536 to virtual user 0 → 65536

  → possible to be 'root' in container without being root on host

  → root user in container gains extra capabilities within that container

Part 5

Case Studies

Case 1: Integration of a NAS software stack

- Main use cases:
  - isolation from the router stack, mostly CPU, RAM, flash, I/O
  - security is not the main goal; NAS stack is semi-trusted
  - independent upgrade cycle from the main router firmware
- « virtual machine » model
  - the container has its own rootfs
  - a list of processes is started at container init

Case 1: Integration of a NAS software stack: Implementation choices

- Single container, always present.
- (Latest version of the) rootfs is embedded in the software image of the router, as a subdirectory of the router's rootfs.
- Upgrade of the container via TR-069 or web UI is possible. The upgrade file is:
  - stored in a flash partition
  - digitally signed, and checked at installation and at every boot
  - contains a squashfs rootfs, mounted through loopback
- Two network interfaces in the NAS « vm »:
  - veth in a private bridge with the host, firewalled, for communication with the host and (NAT'ed) the WAN
  - veth in the LAN bridge, with its own MAC and LAN IP address
- USB storage devices are managed by the NAS « vm » with udev rules
- tmpfs for temporary storage, dedicated flash partition for persistent storage

Case 2: Untrusted third party application store

- Main use cases:
  - third party application deployment on the router
  - isolation from the router stack at every possible level
  - security is critical; the application is untrusted
- Lightweight « virtual machine » model
  - the container's rootfs overlays an (upgradable) « base » filesystem with a set of standard libraries and utilties
  - one or more processes are started at container init

Case 2: Untrusted third party application store: Implementation choices

- any number of containers can be installed
- unprivileged containers are mandatory
- the container image file:
  - is stored in a flash partition
  - is digitally signed, and checked at installation and at every boot
  - contains a squashfs rootfs, mounted through loopback
  - contains a large list of customised configuration settings:
    - network configuration, RAM, CPU, I/O, firewall settings
    - fine-grained access control on management IPC mechanism

... perhaps this use case is better addressed with HW virtualisation.

For more information about
SoftAtHome

Corporate Headquarter

9 rue du Débarcadère
92700 Colombes
France

United Arab Emirates office

Building 6EA office 207
P.O. Box 371438,
Dubai Airport Freezone
United Arab Emirates

Belgium office

Vaartdijk 3 701
3018 Wijgmaal
Belgium

www.softathome.com

contact@softathome.com

linkedin.com/company/softathome

@SoftAtHome