SUPSI

Operating System Security (2)

Operating Systems

Amos Brocco, Lecturer & Researcher

14 settembre 2018

Objectives

- Understand basic security mechanisms used in current OS
- Study how DAC and MAC work
- Study the concept of application sandboxing and isolation
- Study how we can control system resources
- Study how secure boot works

Browsing
- Get a rapid overview.

Reading
- Read it and try to understand the concepts.

Studying
- Read in depth, understand the concepts as well as the principles behind the concepts.

You are also encouraged to try out (compile and run) code examples!

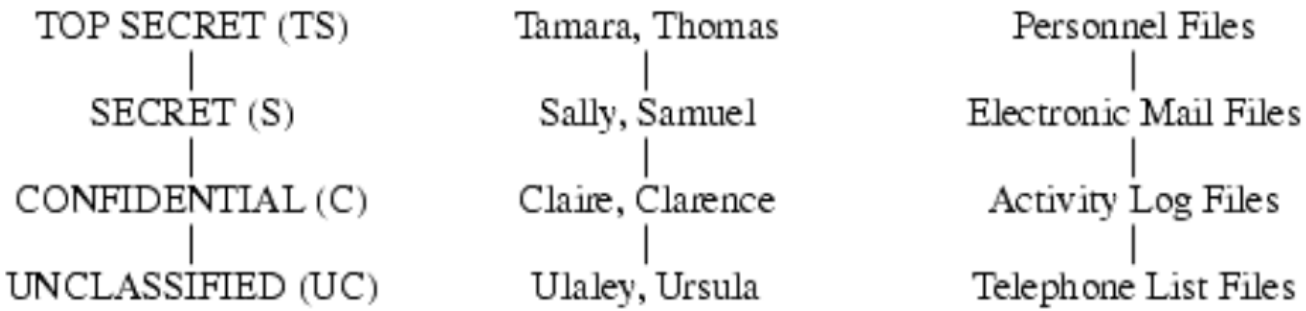
Mandatory Access Control (MAC)

- Whereas DAC focuses on fine-grained access control of objects, **MAC (Mandatory Access Control)** focuses on controlling disclosure of information by **assigning security levels** to objects and subjects *
  - security policies are independent of user operations
  - rules imposed by system administrators cannot be circumvented by users

* Ryan Ausanka-Crues, “Methods for Access Control: Advances and Limitations”

Bell-LaPadula model: multi-level security

- Developed by David Elliot Bell and Leonard J. La Padula
  - developed primarily to provide **confidentiality** → military/governmental security policy: confidentiality policies or information flow policy aim at preventing unauthorized disclosure of information.
- **Multi-level Security** (MLS): Assigns security levels (or *compartments*) to users/ processes* (**clearance levels**) and objects (**sensitivity levels**), with the goal of ***preventing read access to objects at a security classification higher than the subject's clearance***.



security level **clearance level** **sensitive level**

* commonly referred to as "subjects"

(source Matt Bishop, "Computer Security: Art and Science", 2015)

Bell-LaPadula model: multi-level security

- The Bell-Lapadula model uses two security properties based on both discretionary access control (DAC)a and multi-level security:
  - **simple security condition**
    - user/processes cannot access information labeled with a higher classification level
      - user/processes also need to have dicretionary read access to that information
  - *** (star) property**
    - user/processes cannot write to a lower classification level (prevents copying data to a lower security level)
      - user/processes also need to ave discretionary write access to the target

Bell-LaPadula model: multi-level security

- In addition to the simple and * security properties the exist also two tranquility properties:
  - **strong tranquility**: user/processes and objects may not change their security levels once they have been instantiated
  - **weak tranquility**: security levels may change, but not in a way that violates the rules of a given security policy → allows for moving to a higher security level during operation

Linux Security Modules (LSM)

- **LSM (Linux Security Modules)** is a framework which provides "hooks" in the kernel to intercept system calls that access internal kernel objects (for example, i-nodes)
  - modules can connect to such hooks and provide mechanisms for controlling access

- LSM was proposed as a generic solution for implementing security extensions in the Linux kernel, and as an alternative to “embedding” a specific solution (SELinux) into the kernel

MAC example: AppArmor

- **AppArmor** is a MAC framework built upon LSM
- Security policies are called **profiles**
  - each profile is related to a filesystem path (for example pointing to a program)
  - profiles consist of a set of rules based on Linux/POSIX capabilities and file permissions
  - Tracing can be used to determine permissions required and to create a new profile
- http://wiki.apparmor.net/index.php/Documentation
  - Used by: OpenSUSE, SLED, Ubuntu

MAC example: AppArmor (sample profile)

```
#include <tunables/global>
/usr/sbin/tcpdump {
  #include <abstractions/base>
  #include <abstractions/nameservice>
  #include <abstractions/user-tmp>
  capability net_raw,
  capability setuid,
  capability setgid,
  capability dac_override,
  network raw,
  network packet,
  # for -D
  capability sys_module,
  @{PROC}/bus/usb/ r,
  @{PROC}/bus/usb/** r,
  # for -F and -w
  audit deny @{HOME}/.* mrwkl,
  audit deny @{HOME}/.*/ rw,
  audit deny @{HOME}/.*/** mrwkl,
  audit deny @{HOME}/bin/ rw,
  audit deny @{HOME}/bin/** mrwkl,
  @{HOME}/ r,
  @{HOME}/** rw,
  /usr/sbin/tcpdump r,
}
```

```
r - read
w - write
m - memory map as executable
k - lock
l - create hard-link
```

MAC example: SELinux

- **SELinux** was originally created by NSA and Red Hat
  - is built upon LSM
- It provides different forms of access control:
  - Type Enforcement (TE)
  - Role Based Access Control (RBAC)
  - MLS (Bell-Lapadula Model) (not covered here, refer to https://selinuxproject.org/page/NB_MLS)
- Rules are configured centrally and enforced by the kernel
- Subjects (users/processes) and objects have a **security context**
  - *note: SELinux users may differ from system users*

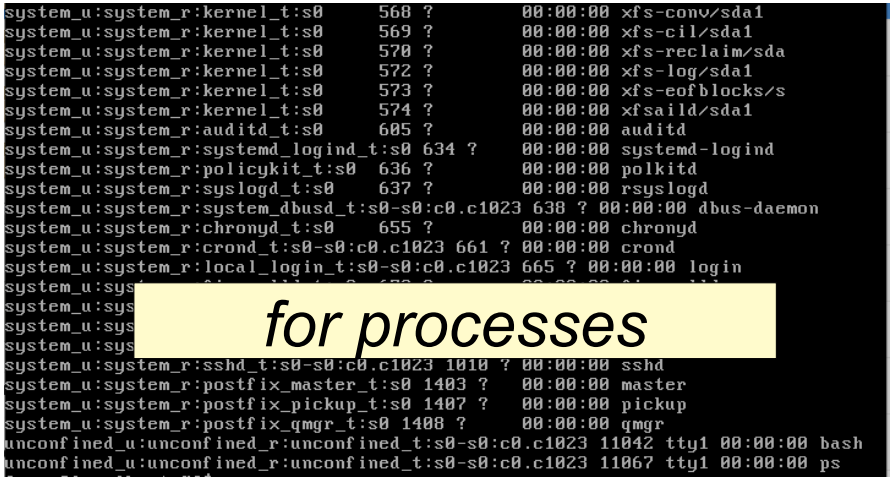
MAC example: SELinux security context

mostly used

username:role:type*:mls

* a.k.a *domain* for processes

`ls -Z`



`ps -eaZ`

for users

`id -Z`

MAC example: SELinux Type Enforcement

- **Type Enforcement**
  - processes and resources are assigned to a **type**
    - for running processes this is known as **domain**
  - resources are divided into different **classes**
    - for example: file, dir, socket, process
      - see https://selinuxproject.org/page/ObjectClassesPerms
  - each class has some specific **permissions**
    - for example: read, write, execute

MAC example: SELinux Classes and Permissions (example)

common file

Permission	Description
getattr	Get file attributes for file, such as access mode. (e.g. sta
relabelto	Relabel to new security context.
unlink	Remove hard link (delete).
ioctl	IO control system call requests not addressed by other p
execute	Execute
append	Write to a file opened with O_APPEND.
read	Read file contents.
setattr	Change file attributes for file such as access mode. (e.g.
swapon	Allows file to be used for paging/swapping space.
write	Write to a file.
lock	Set and unset file locks.
create	Create new file.
rename	Rename a file.
mounton	Use as mount point; only useful for directories and files in
quotaon	Use as a quota file.
relabelfrom	Relabel from old security context.
link	Create another hard link to file

file

Inherits from: common file

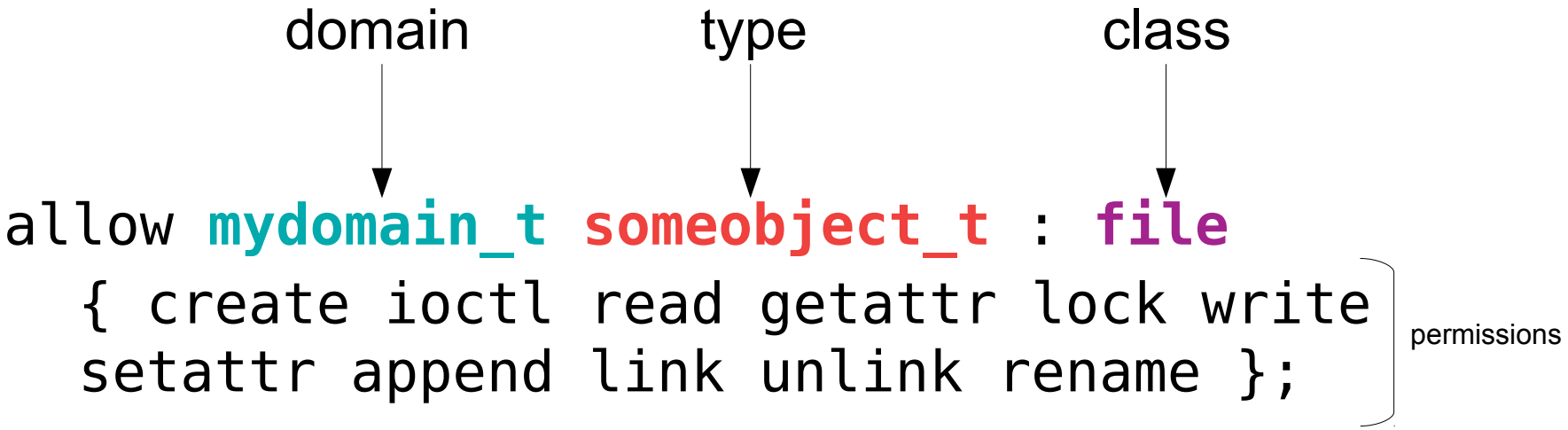
Permission	Description	Kernel Version/Capability
getattr	see common file:getattr	
relabelto	see common file:relabelto	
unlink	see common file:unlink	
ioctl	see common file:ioctl	
execute	see common file:execute	
append	see common file:append	
read	see common file:read	
setattr	see common file:setattr	
swapon	see common file:swapon	
write	see common file:write	
lock	see common file:lock	
create	see common file:create	
rename	see common file:rename	
mounton	see common file:mounton	
quotaon	see common file:quotaon	
relabelfrom	see common file:relabelfrom	
link	see common file:link	
execute_no_trans	Execute a file in the callers domain.	
entrypoint	Can be executed as the entry point of the new domain in a transition.	
execmod	Make executable a file mapping that has been modified by copy-on-write. (Text relocation)	2.6.11+
open	Open a file.	2.6.26+ / open_perms

ref. https://selinuxproject.org/page/ObjectClassesPerms#file

MAC example: SELinux Type Enforcement Rules

- Rules can specify what each domain can do which each resource types using
  - the source **domain** *(type of running process)*
  - the target **type** of the objects to which access is allowed
  - the object **classes** to which the rule applies
  - the access **permission** allowed
- Policies can be **strict** (denies access unless otherwise specified) or **targeted** (apply to system processes)
- Rules can also declare types and **transitions**
  - A type **transition** results in a new process running in a new domain different from the executing process, or a new object being labeled with a type different from the source doing the labeling.

MAC example: SELinux Type Enforcement Rule (example)



"Allow processes in domain (*with type*) **mydomain_t** to perform the specified actions on objects of class **file** of type **someobject_t** "

MAC example: SELinux RBAC

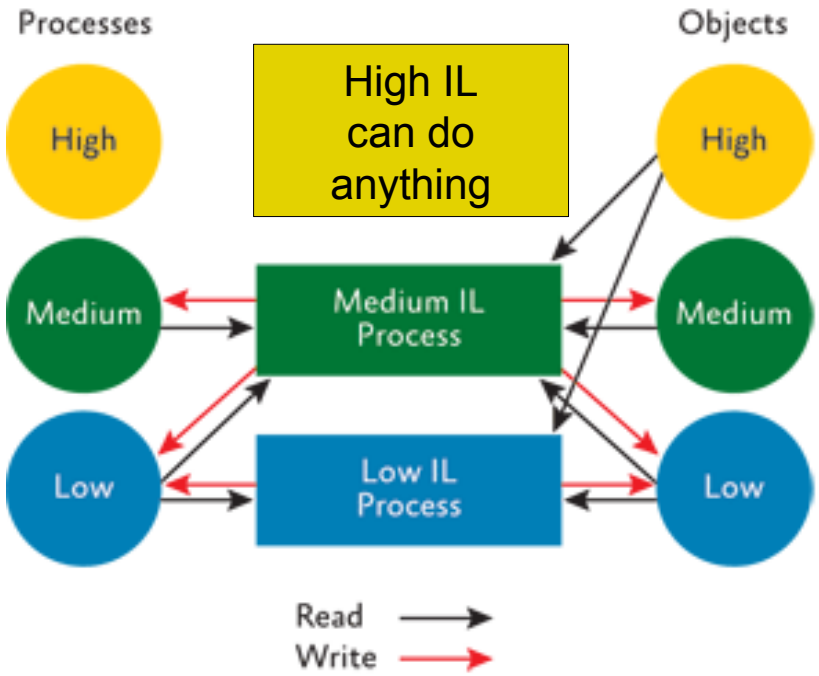
- SELinux also provides a form of **role-based access control** (RBAC)
  - Roles define which SELinux user identities can have access to what domains
    - Rules can limit the types to which a process may transition based on the role identifier
      - **`role administrator_r types mydomain_t;`**
        declares the *administrator_r* role with access to the *mydomain_t* domain
      - **`allow user_r administrator_r;`**
        allow transition from *user_r* to *administrator_r* (note: this alone is not enought to transition: there must be a transition rule allowing this operation, see https://selinuxproject.org/page/TypeRules#type_transition_Rule)

MAC on Linux (some remarks)

- AppArmor/SELinux are not the only security frameworks on Linux
  - other examples: TOMOYO, grsecurity
- In Linux MAC and DAC are orthogonal
  - **the kernel checks DAC permissions before MAC**
- SELinux policies are considered to be more complex than AppArmor
  - SELinux Alert Browser can help trubleshoot issues
- SELinux identifies file system objects by i-node whereas AppArmor uses the filesystem's path
  - creating an hard link to a file can circumvent AppArmor rules:
    - This link problem can be prevented by setting **`/proc/sys/fs/protected_hardlinks`** to 1 (so that users can no longer create soft or hard links to files which they do not own)

MAC on Windows: Windows Integrity Levels

- Since Windows Vista the operating system implements a **Mandatory Integrity Control (MIC)** mechanism
  - each subject (user/process) is associated with an **Integrity Level(IL)**
  - for each object with an Access Control List (typically files, but also registry keys, processes and threads) it is possible to specify the minimum Integrity Level required for write access
    - a process can write or delete an object only when its IL is greater or equal to the minimum required (as specified for the object)
- Windows defines four integrity levels: low, medium, high, and system
  - standard users receive medium, elevated users receive high.



(source: https://technet.microsoft.com/en-us/library/2007.06.uac.aspx)

Windows Integrity Levels vs Biba integrity model

- The Windows integrity mechanism resembles the **Biba security model**
  - developed by Kenneth J. Biba, it focuses on **integrity**
    - Different integrity levels are defined (low, medium high)
    - Prevents unauthorized user from making modifications to data with high integrity requirements
      - Subjects cannot read objects of lesser integrity ("no-read-down"), nor write objects or higher integrity ("no-write-up")
  - In contrast to Biba, Windows ILs **allow for reading objects at a lower integrity level**, and **prevent reading the address-space of process objects belonging to a higher level** ("no-read-up")

MAC on BSD/OSX: TrustedBSD / Seatbelt

- **TrustedBSD** is a mandatory access control framework implemented by the FreeBSD kernel and also used on MacOS
  - can be used to isolate applications (sandbox)
  - policies can be defined by means of kernel modules
    - different implementations can coexist on the same system
- TrustedBSD enables filtering of different events such as
  - access to the filesystem
  - access to the network
  - starting new processes
  - signals
- **Seatbelt** is an OSX Application Sandbox which enables user to apply sandboxing profiles written in a Lisp-like language when running applications

MAC on OSX: Seatbelt policy example

```
(version 1)

(deny default)

(allow signal (target self))

(allow file*
       (literal "/dev/dtracehelper")
       (literal "/dev/urandom")
       (literal "/dev/null")
       (regex #"^/Users/[a-zA-Z0-9_]+/Library/Application Support/Firefox")
       (subpath "/tmp")
       (subpath "/private/tmp")
       )

(allow file-read*
       (literal "/Library/Application Support/Macromedia/FlashPlayerTrust")
       (subpath "/usr")
       (subpath "/System/Library/Frameworks")
       (regex #"^/Users/[a-zA-Z0-9_]+")
       (subpath "/Library/Preferences")
       (subpath "/Applications/Firefox.app")
       (subpath "/var")
       (subpath "/private/var")
       (subpath "/private/etc")
```

(source: https://github.com/pansen/macos-sandbox-profiles/blob/master/firefox.sb)

The root user problem

- In a Unix/Linux system the root user (UID 0) is not bound to normal DAC/MAC rules
  - Working as privileged user root is therefore "dangerous"
  - … however always working as unprivileged user is impossible → there are some tasks that still need higher privileges
  - We need some granularity

Going further: POSIX/Linux Capabilities

- The general distinction between “normal users” and root clashes with the principle of least privilege
  - either you have few privileges (normal user)
  - or you have full privileges (root)
- **POSIX capabilities** can be used to provide a more granular privilege control:
  - capabilities refer to tokens used by processes to prove that they are allowed to access an object
  - capability based security uses lists to map objects to rights: it can effectively *switch off* certain facilities (system calls) for a process at the kernel level.

Going further: POSIX/Linux Capabilities (excerpt)

Capabilities list

The following list shows the capabilities implemented on Linux, and the operations or behaviors that each capability permits:

CAP_AUDIT_CONTROL (since Linux 2.6.11)
Enable and disable kernel auditing; change auditing filter rules; retrieve auditing status and filtering rules.

CAP_AUDIT_READ (since Linux 3.16)
Allow reading the audit log via a multicast netlink socket.

CAP_AUDIT_WRITE (since Linux 2.6.11)
Write records to kernel auditing log.

CAP_BLOCK_SUSPEND (since Linux 3.5)
Employ features that can block system suspend (**epoll**(7) **EPOLL-WAKEUP**, /proc/sys/wake_lock).

```
man 7 capabilities
```

Going further: POSIX/Linux Capabilities

- Each process has four* sets (or masks) of capabilities:
  - **effective**: when a privileged operation is requested, the kernel checks if the capability is within this set
  - **permitted**: contains the capabilities which are permitted (a process can temporarily disable a capability, which will get removed from the effective set)
  - **inheritable**: determines which capabilities will get inherited when an **`exec`** system call starts a new program inside the process (this is needed because **`clone`** and **`fork`** will just copy the existing capabilities)
  - **bounding set (bset)**: maximum capabilities of a process

* in Linux actually five, as there is also an *ambient* set, see https://lwn.net/Articles/632520/

Going further: Unix/Linux sudo

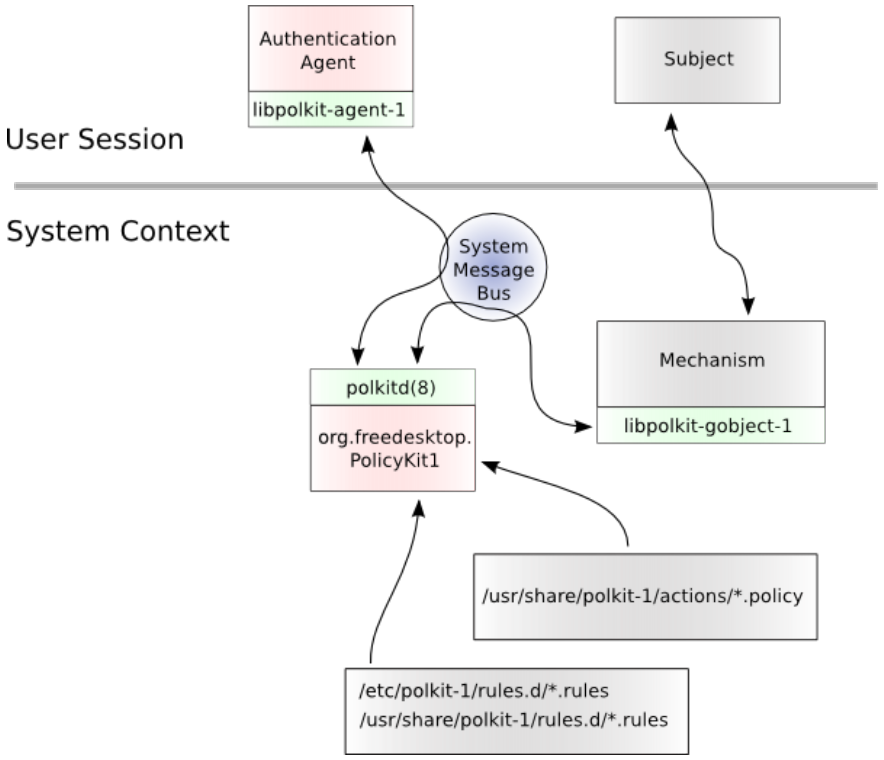
- **sudo** can be used to execute a command with root privileges without logging in as root
- sudo has the **setUID** bit set, which means that it's executed with the access right of the file owner (root) instead of those of the user calling the program:

```
attila@localhost:~> which sudo
/usr/bin/sudo
attila@localhost:~> ls -l /usr/bin/sudo
-rwsr-xr-x 1 root root 150944 12 ott 03.16 /usr/bin/sudo
```

- When sudo is invoked it checsk the rules defined in **`/etc/sudoers`**, asks for the user password, and spaws a child process with root privileges to execute the specified command
  - rules can limit which commands can be executed and which users can execute them

Going further: PolKit (a.k.a. PolicyKit)

- **polkit** "is an application-level toolkit for defining and handling the policy that **allows unprivileged processes to speak to privileged processes**"
  - processes are not granted root privileges, but can ask a privileged process to perform some action which would normally require root privileges



(source: https://www.freedesktop.org/software/polkit/docs/latest/polkit-architecture.png)

* https://www.freedesktop.org/wiki/Software/polkit/

Going further: Windows User Access Control (UAC)

- **Windows User Access Control (UAC)** was introduced with Windows Vista, and is designed to address the problem of users and applications having by default too much power
- Users are initially given minimal administrative rights
  - when needed, similarly to **sudo**, the user is presented a consent and/or credential prompt dialog to confirm his/her actions or to elevate his/her privileges

Consent prompt

Credential prompt

Sandboxing

- **Sandboxing** can be used to separate running (untrusted) programs in a **confined** and **controlled** environment, typically with restricted access to system resources, in order to prevent spreading of security problems due to vulnerabilities or bugs
  - sandboxing can be based on operating system features such as MAC, system call filtering, capabilities,...
  - virtualization also provides sandboxing

Simple sandboxing: system call filtering with seccomp/seccomp-bpf

- **seccomp** / **seccomp-bpf** (secure computing mode, berkley packet filter) is a security mechanism implemented by the Linux kernel
  - version 1 (non-bpf) allowed for only **read**, **write**, **_exit** and **sigreturn**
  - **seccomp-bfp** allows filtering of system calls using customizable policies described using Berkeley Packet Filter rules
    - filters can be installed at runtime, but cannot be removed
    - filters are inherited by child processes
    - seccomp-bfp filters are used by Firejail (asandboxing tool) and Chromium web browser on Linux
  - Limitations: seccomp cannot setup filters based on filenames or port/ip (filter are based on arguments, cannot dereference pointers, like strings)

Simple sandboxing: seccomp-bpf (example)

```c
#include <stdio.h>
#include <stdlib.h>
#include <libseccomp/seccomp.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <assert.h>
#include <fcntl.h>
// needs: libseccomp-dev or libseccomp-devel
// compile with: gcc -o seccomp_example seccomp_example.c -lseccomp
int main() {
	scmp_filter_ctx ctx;
	FILE *exfile;
	int fd;
	exfile = fopen("/tmp/myfile", "w");
	// We want that filtered calls to fail with an error
	assert(ctx = seccomp_init(SCMP_ACT_ERRNO(13)));
	// Default is to deny: define rules to allow some system calls
	assert(seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0) >= 0);
	assert(seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(close), 0) >= 0);
	assert(seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0) >= 0);
	// Load rules
	assert(seccomp_load(ctx) >= 0);
	printf("This will work...\n");
	fprintf(exfile, "hello world\n");
	printf("This will fail...\n");
	fd = open("/tmp/myfile2", O_RDWR);
	if (fd < 0) { printf("Got an error: %d errno=%d\n", fd, errno);}
	fclose(exfile);
	exit(0);
}
```

Simple sandboxing NOT: chroot

```
man 2 chroot
```

chroot() changes the root directory of the calling process to that specified in path. This directory will be used for pathnames beginning with /. The root directory is inherited by all children of the calling process.

Only a privileged process (Linux: one with the **CAP_SYS_CHROOT** capability in its user namespace) may call **chroot**().

This call changes an ingredient in the pathname resolution process and does nothing else. In particular, it is not intended to be used for any kind of security purpose, neither to fully sandbox a process nor to restrict filesystem system calls. In the past, **chroot**() has been used

Simple sandboxing NOT: exiting a chroot (example)

```
#!/bin/bash
cat <<-EOF > escapechroot.c
#include <sys/stat.h>
#include <unistd.h>
int main() {
       mkdir("escapedoor", 0755);        // Current working directory is X
       chroot("escapedoor");        // chroot does not change the working directory
       // here you will normally chdir("/"); so that cwd -> /
       // and "../../../" will get translated to "/../../../" -> /
       // But for now cwd -> ../
       // and "../../../" will get translated to "../../../../"
       // hence you escape the jail (if the absolute path is /home/user/mychroot/escapedoor)
    chroot("../../../");
    // Note: if you don't know exactly where you are you can repeat ../
    // many times (at some point you will reach the root)
    return execl("/bin/bash", "-i", NULL);
}
EOF
mkdir -p mychroot/{bin,lib,usr,var,etc}
gcc -o mychroot/bin/escapechroot escapechroot.c
CHROOT_CMDS="bash"
for cmd in $CHROOT_CMDS ; do
      cp -v $(which $cmd) mychroot/bin
      for l in $(ldd $(which $cmd) | awk '{print $1}'); do
            find /lib* /usr/lib* -name "$(basename $l)" 2>/dev/null -exec cp -v {} mychroot/lib \;
      done
done
cd mychroot
ln -s lib usr/lib
ln -s lib usr/lib64
ln -s lib lib64
sudo chroot . /bin/bash
# another fix, start chroot with unprivileged user:
# sudo chroot --userspec=1000:1000 $(pwd) /bin/bash
```

Sandboxing with Linux Namespaces

- The problem: In a typical Linux setup each userspace process is able to see global information about the system, such as:
  - mounted filesystems
  - process hierarchy
  - network interfaces

Sandboxing with Linux Namespaces

- The solution: provide different “views” on the system by means of separate namespaces
- A **namespace** wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource
  - Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes.

Sandboxing with Linux Namespaces

- There are currently 7 different namespaces:
  - Peer namespaces:
    - **mount**: provides an “isolated” view of mounted filesystems
    - **UTS**: provides custom host and domain information
    - **IPC**: provides isolated interprocess communication, for example to prevent processes from accessing shared memory areas
    - **network**: provides a custom view of a subset of network devices (with an independent network stack)

Sandboxing with Linux Namespaces

- Hierarchy namespaces (permissions are inherited)
  - **PID**: Processes hierarchy (each hierarchy starts with PID 1)
  - **user**: namespace of user identifiers (each user namespace has a UID 0, for the root user)
  - **cgroup**: provides a way to isolate system resource limits

Sandboxing with Namespace: namespace creation

- Namespace are created when spawning a new process through the clone system call
  - the child process and all descendants can be isolated inside a new namespace
- It is also possible to create a new namespace and attach the current process to it using the **unshare** system call, the **unshare** shell command, or use **setns** to join an existing namespace

Example: unshare from the shell

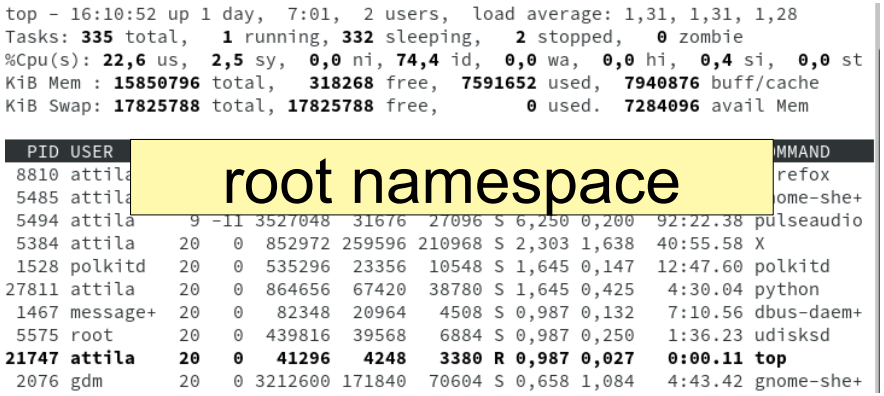
- To create a separate child namespace for mount and pid

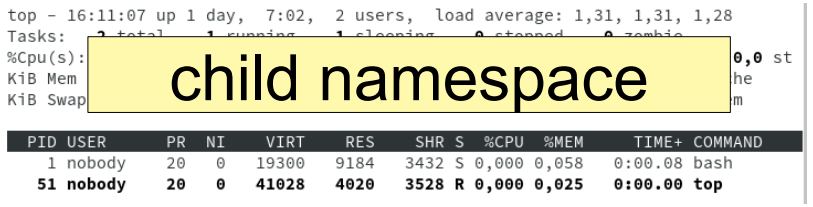
  `sudo unshare --mount --pid --fork --mount-proc`

  - by default this will spawn a new shell
  - depending on the operating system, it is also possible to create user namespaces (without requiring root privileges or sudo) with:

    `unshare --user --mount --pid --fork --mount-proc`

- You can try commands like **`top`** or **`mount`** to check that namespaces are different





- `mkdir alpha beta; sudo mount --bind alpha beta` in the root namespace
- `mount` see the differences between the root and the child namespace

Example: unshare from the shell

- We can go further and create a new filesystem root for our container:

```
#!/bin/bash
# save script as simplecontainer.sh and run with sudo unshare --mount --pid --fork ./simplecontainer.sh

# 1. Create the container filesystem
mkdir mycontainerroot 2>/dev/null
mkdir -p mycontainerroot/{bin,lib,usr,var,etc}
CONTAINER_CMDS="bash ls mkdir mount ps umount rmdir"
for cmd in $CONTAINER_CMDS ; do
       cp -v $(which $cmd) mycontainerroot/bin
       for l in $(ldd $(which $cmd) | awk '{print $1}'); do
               find /lib* /usr/lib* -name "$(basename $l)" >&/dev/null -exec cp {} mycontainerroot/lib \;
       done
done

# 2. Mount the container filesystem
mkdir container
mount --bind mycontainerroot container

# 3. Move into the container filesystem and create base directories
cd container
ln -s bin usr/bin
ln -s lib usr/lib
ln -s lib usr/lib64
ln -s lib lib64
mkdir prev-root
```

Example: unshare from the shell

- We can go further and create a new filesystem root for our container:

```
# 4. Change the root (like chroot)
pivot_root . prev-root
cd /
export PATH=/bin:/usr/bin

# 5. Mount other basic filesystems
mkdir /proc /dev
mkdir /dev/pts
mount -t proc proc /proc
mount -t devtmpfs -o size=50k,nr_inodes=2k none /dev
mount -t devpts -o newinstance,gid=5,mode=620,ptmxmode=666 none /dev/pts

# 6. Unmount old root
umount -l -R prev-root
rmdir prev-root

# 7. Exec a shell
exec /bin/bash -i

# 8. To enter into the namespace use nsenter --target {pid of bash} --mount --pid
```

* **pivot_root** is similar to **chroot**, but allows for accessing the old filesystem tree before (manually) unmounting it (in the new namespace). Additionally since it requires the root of a mounted filesystem cannot be easily escaped.

Windows Software Restriction Policies (SRP)

- "**Software Restriction Policies (SRP)** is Group Policy-based feature that identifies software programs running on computers in a domain, and controls the ability of those programs to run."
  - SRP have been introduced in Windows XP and Windows Server 2003, and enable system administrator to implement restriction policies and define what programs should be trusted and allowed to run

(source: https://technet.microsoft.com/en-us/library/hh831534(v=ws.11).aspx)

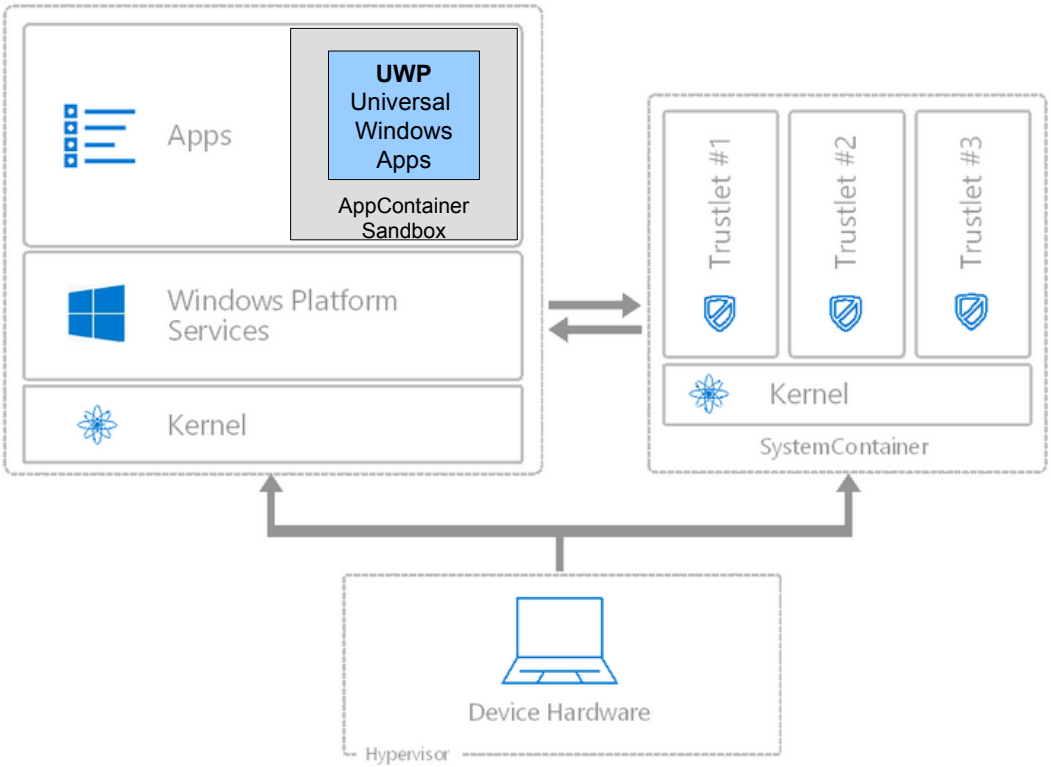
Windows AppLocker

- **AppLocker** was introduced with Windows 7 as an improvement over SRP (which has been since then deprecated)
- The main improvements over SRP are *:
  - By default SRP allows execution, and requires rules to deny; AppLocker denies by default and allows execution only if a corresponding rule is created.
  - SRP policies apply on the whole system, whereas AppLocker rules can be enforced on a specific user or a group of users.
  - SRP can only enforce a rule, whereas AppLocker provides an audit mode to check rules before they are enforced.

* https://technet.microsoft.com/library/hh994614

Windows 10 Virtualization-Based Security

- **Virtualization-Based Security (VBS)** makes use of hardware virtualization features to create and isolate a memory area from the normal operating system



AppContainer: protects system, apps and data (used for UWP)

SystemContainer (aka. Virtual Secure Mode VSM) protects sensitive parts of the system

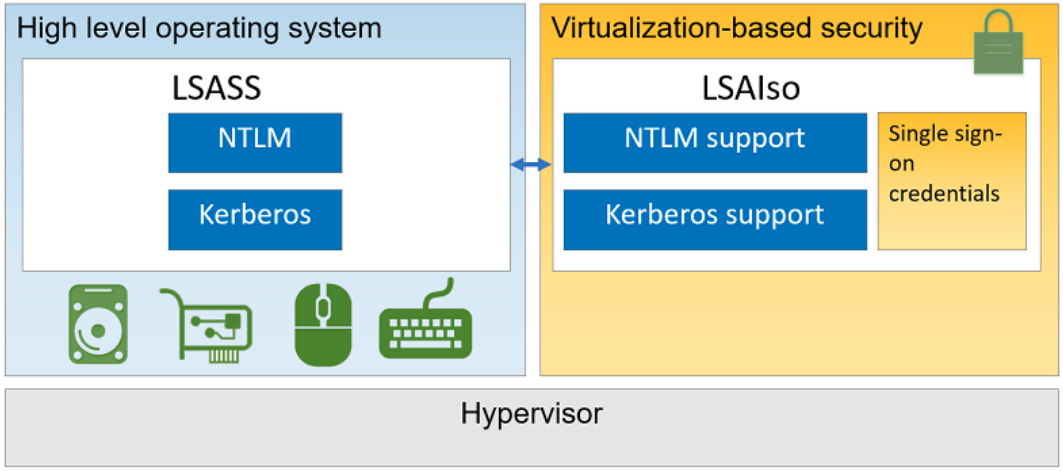
(source: https://blogs.technet.microsoft.com/windowsitpro/2016/06/29/step-change-in-security-with-modern-devices-and-architecture/)

Windows 10 Device Guard

- **Device Guard** is a technology which makes use of software and hardware security mechanisms to define integrity policies enabling execution of only trusted applications
  - supports locking down devices
  - check code signatures: allows only apps considered **trusted** to be executed on the system (system administrators can decide what they deem trusted)
    - similar to Software Restriction Policies (SRP, Vista) and Applocker (Windows 7+), but the process which verifies code is run in secure mode (trustlet)

Windows 10 Credential Guard

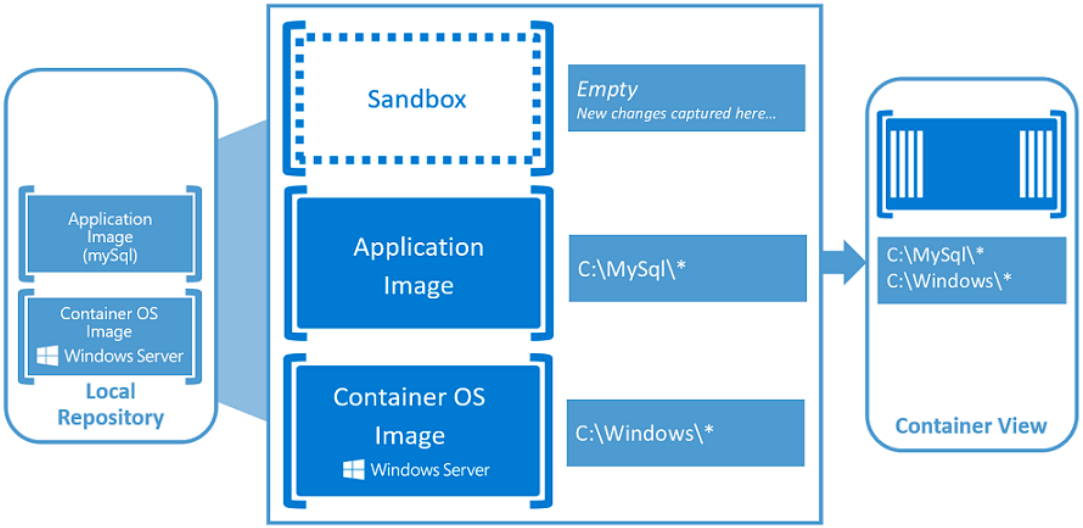
- **Credential Guard** is a security tool which makes use of virtualization to protect NTLMv2 and Kerberos credentials in a secure virtualized environment
  - prevents attacks which try to access the operating system memory to steal credentials: instead of storing credentials in the LSA (Local Security Authority), an isolated LSA is created in a virtualized container.
  - local account/Microsoft credentials are not protected



(source: https://docs.microsoft.com/it-it/windows/access-protection/credential-guard/credential-guard-how-it-works)

Windows Containers

- **Windows Containers** allow for the creation of isolated, resource controlled runtime environments
  - Two types of containers are supported:
    - **Windows Server Containers**:provide application isolation through process and namespace isolation (the kernel is shared with the host and all other containers) → can be managed with Docker*
    - **Hyper-V Isolation**: container are run in a virtual machine.



(source: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/)

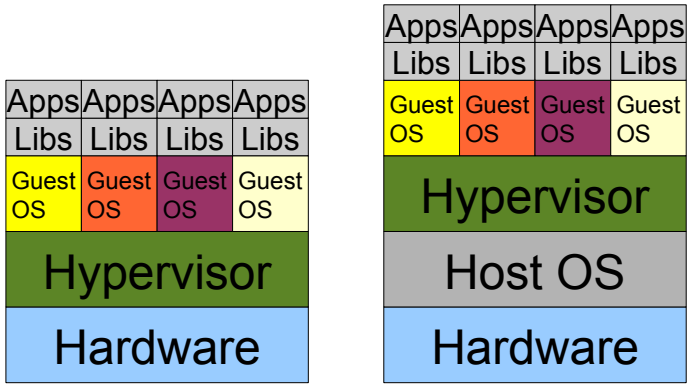
* Linux Docker containers are run using Hyper-V and a minimal Linux distribution called LinuxKit

Lightweight virtualization: Linux Containers

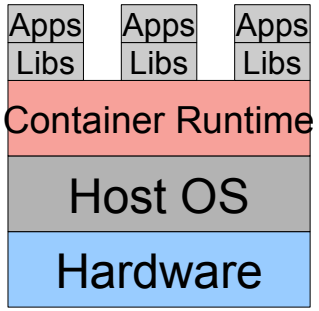
- **Linux containers** are “constructed” using technologies from the kernel:
  - namespaces
  - cgroups
  - seccomp
  - MAC (selinux, apparmor)
- The concept of “containers” does not exist in the kernel
  - different userspace managers exist to provide “containers” :
    - docker
    - lxc
    - flatpak
    - ...

Lightweight virtualization: Linux Containers

- Containers can be viewed as a “lightweight” virtual machine
  - can install and run its own software packages
  - can have its own users (even root!)
  - can configure its own network
- … but each container shares the same host kernel (i.e. you cannot use a different operating system)



Virtualization



Containers

Why containers?

IBM Research Report

An Updated Performance Comparison of Virtual Machines and Linux Containers

Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758
USA

V. Conclusions and Future Work

Both VMs and containers are mature technology that have benefited from a decade of incremental hardware and software optimizations. In general, Docker equals or exceeds KVM performance in every case we tested. Our results show that both KVM and Docker introduce negligible overhead for CPU and memory performance (except in extreme cases). For I/O-intensive workloads, both forms of virtualization should be used carefully.

We find that KVM performance has improved considerably since its creation. Workloads that used to be considered very challenging, like line-rate 10 Gbps networking, are now possible using only a single core using 2013-era hardware and software. Even using the fastest available forms of paravirtualization, KVM still adds some overhead to every I/O operation; this overhead ranges from significant when performing small I/Os to negligible when it is amortized over large I/Os. Thus, KVM is less suitable for workloads that are latency-sensitive or have high I/O rates. These overheads significantly impact the server applications we tested.

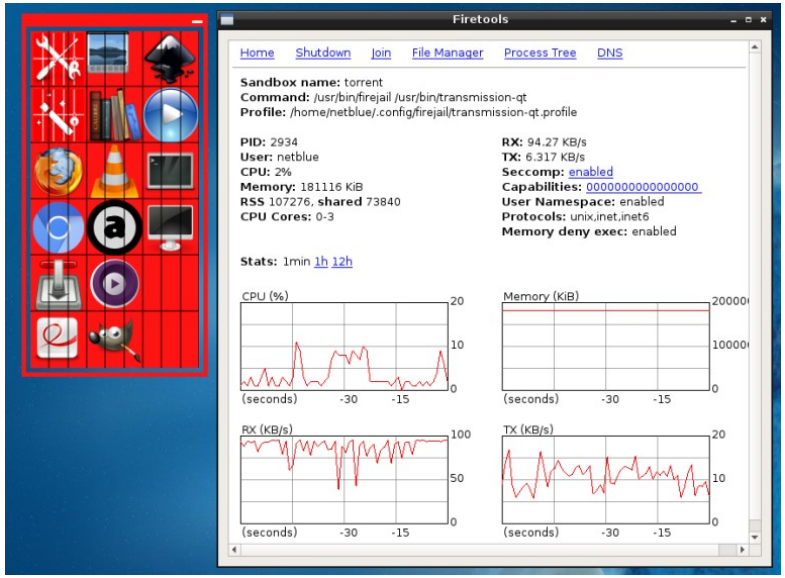
Although containers themselves have almost no overhead, Docker is not without performance gotchas. Docker volumes have noticeably better performance than files stored in AUFS. Docker's NAT also introduces overhead for workloads with high packet rates. These features represent a tradeoff between ease of management and performance and should be considered on a case-by-case basis.

How are containers implemented

- The concept of “container” is based on different technologies:
  - **namespaces** (to create “private” views on the underlying system)
  - **cgroups** (to control resources)
  - **seccomp-bpf**
  - “chroot” (using `pivot_root`)
- There exist different container runtimes:
  - Docker
  - LXC
  - LXD

Lightweight virtualization: Firejail

- Instead of creating full containers **Firejail** enables user to confine applications into an isolated and controlled sandbox
- Firejail makes use of different technologies:
  - Linux namespaces
  - seccomp-bpf
  - chroot
  - capabilities
  - overlayfs (mount a Linux overlayfs filesystem on top of regular filesystem. All modifications stay in overlay layer)
  - bind mounts (to render files/directories inaccessible or read-only)



(source: https://firejail.wordpress.com/screenshots/)

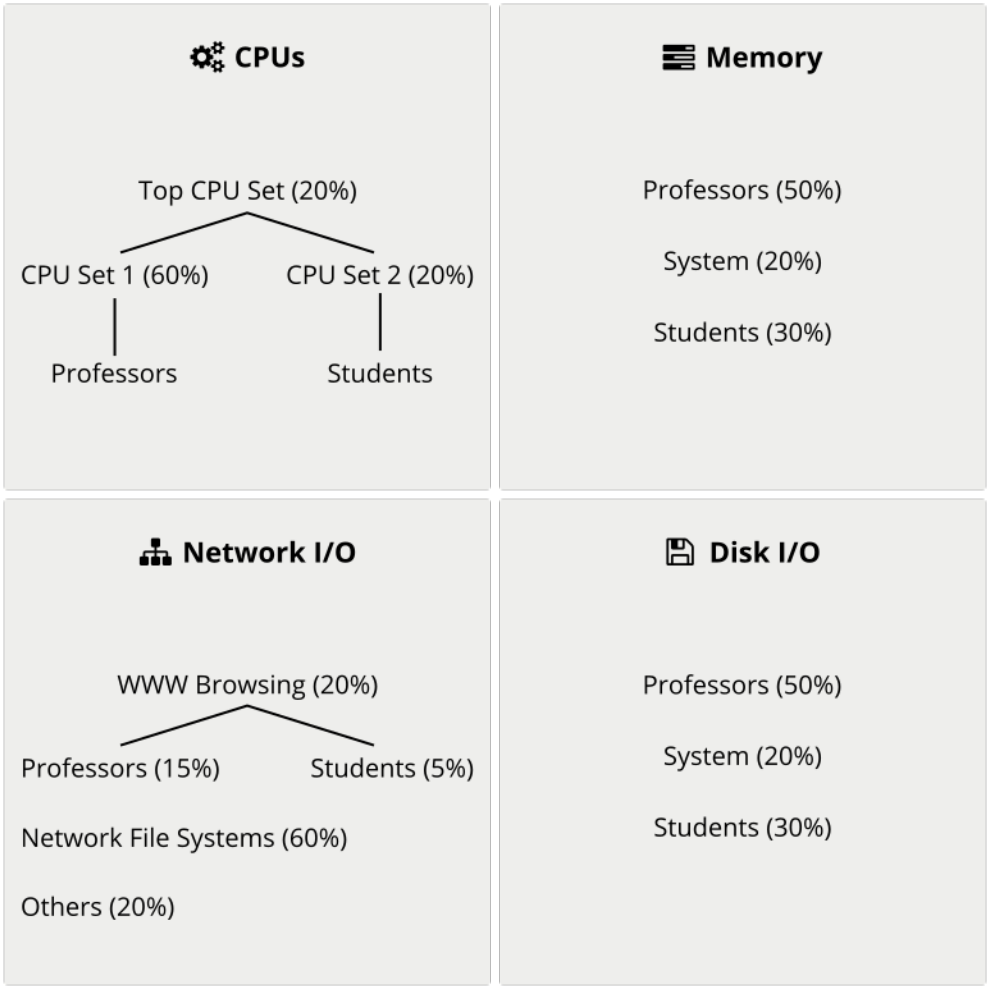
Controlling resources: prlimit and ulimit

- The tasks of an operating system also include resource control
- In Linux it is possible to control limits for some resources using the **prlimit** and **ulimit** commands (or by setting limits in **`/etc/security/limits.conf`**)
  - **`prlimit`** allows for setting process resource limits (either soft or hard)
    - limits are inherited by child processes
    - Example: see the default limits for a process: **`prlimit --pid pid`**
    - Example: per-user processes/threads*:
      `prlimit --nproc=1024 /bin/bash`

      at most 1024 threads for this user
      → limits effects of a fork-bomb
  - **`ulimit`** works in a similar way but applies system-wide limits
    - Example: see current limits: ulimit -a
    - Apply soft process limit: **`ulimit -S -u 128`**
    - Apply hard process limit: **`ulimit -H -u 128`**

* to see how many threads are running use `ps m -ef | grep $(whoami) | wc -l`

A more precise way of controlling resources



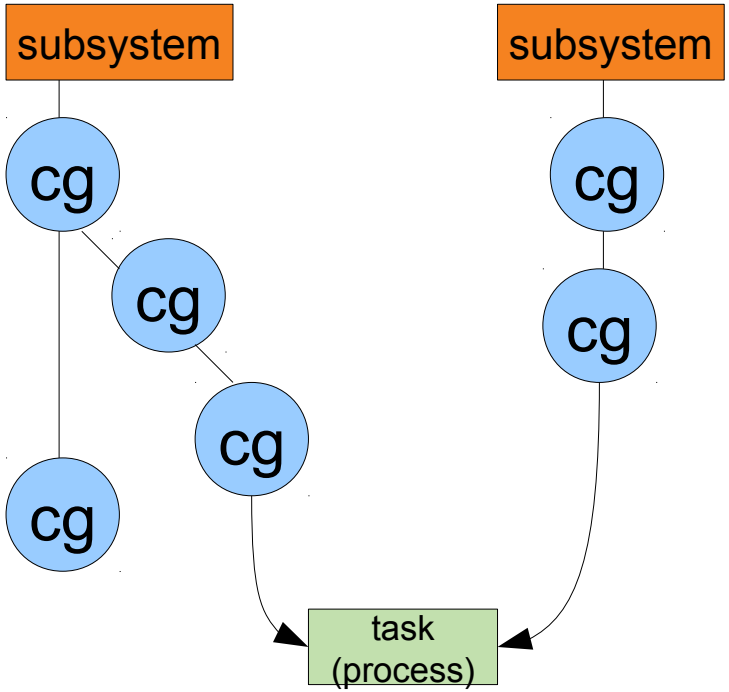
(Source: https://doc.opensuse.org/documentation/leap/tuning/html/book.sle.tuning/cha.tuning.cgroups.html)

Resource control: Linux Control Groups (cgroups)

- We saw that Linux namespaces can limit the resources seen by a process
- With **Control Groups (cgroups)** we can limit how many (*how much*) resources a process can use
  - we can **allocate resources** such as CPU time, memory, network bandwidth among user-defined groups of processes (tasks)
  - **fine-grained control** over allocating, prioritizing, denying, managing, and monitoring system resources
  - we can **monitor** resource usage

Resource control: cgroups hierarchies

- Control groups are organized into **hierachies**
  - multiple hierarchies can co-exist on a system
- Hierarchies are attached to one or more subsystems
  - a **subsystem** represents a single resource (for example CPU or memory)
- Control groups define some parameters (“limits”) that apply to the related subsystem
  - control groups inherit parameters from their parent in the hierarchy
  - tasks (processes) can be assigned to at most a cgroup per-hierarchy



Resource control: cgroups subsystems

- The available subsystems are exposed through the **/proc** filesystem

```
$ cat /proc/cgroups
#subsys_name    hierarchy       num_cgroups     enabled
cpuset  11      1       1
cpu     6       1       1
cpuacct 6       1       1
blkio   8       1       1
memory  5       1       1
devices 9       93      1
freezer 12      1       1
net_cls 7       1       1
perf_event      4       1       1
net_prio        7       1       1
hugetlb 3       1       1
pids    2       102     1
rdma    10      1       1
```

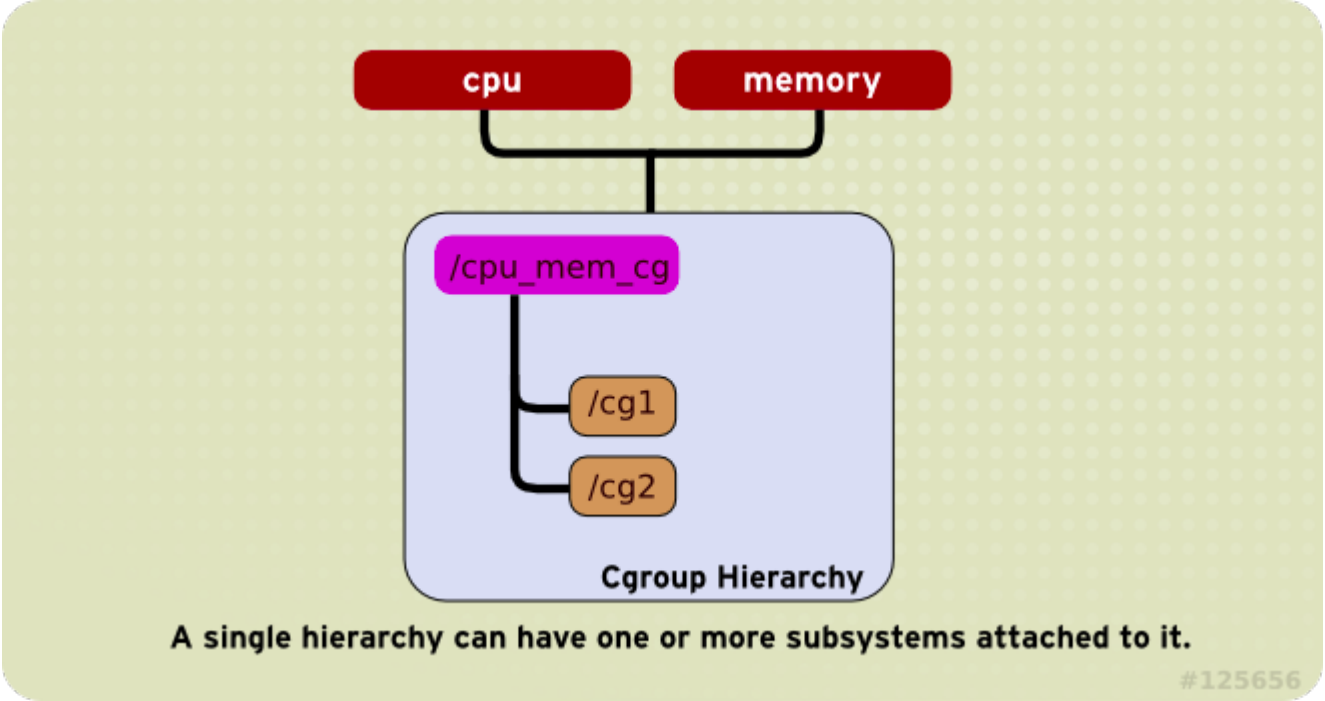
Resource control: cgroups subsystems

- **blkio**: manages limits on block devices I/O
- **cpu**: manages scheduling limits
- **cpuacct**: generates reports on CPU time used by tasks in the cgroup
- **cpuset**: can be used to assign CPU cores to tasks
- **devices**: allows or denies access to devices
- **freezer**: this subsystem suspends or resumes tasks
- **memory**: this subsystem sets limits on memory use
- **net_cls**: tags network packets and allows the kernel to identify traffic from tasks from a cgroup
- **net_prio**: allows for prioritizing network traffic per network interface

Each subsystem exposes some **parameters** which can be read or written

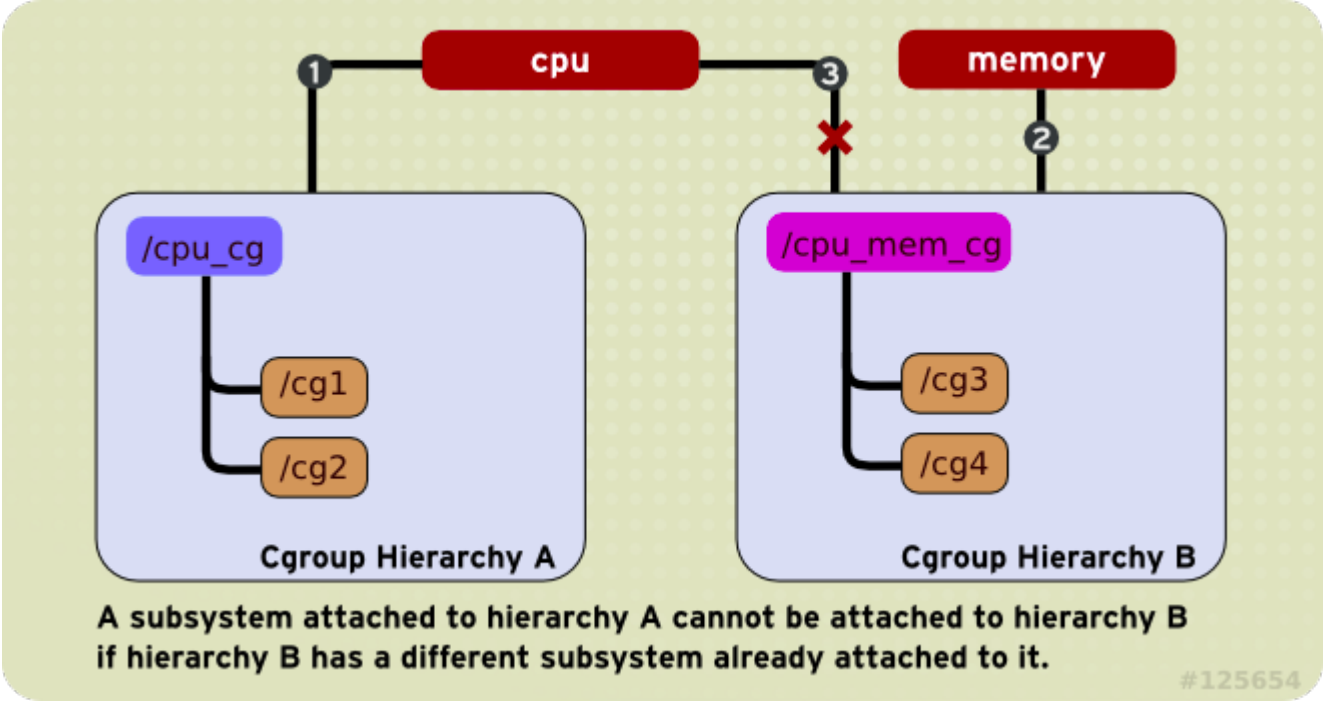
Examples:
`memory.limit_in_bytes`

Resource control: cgroups rules



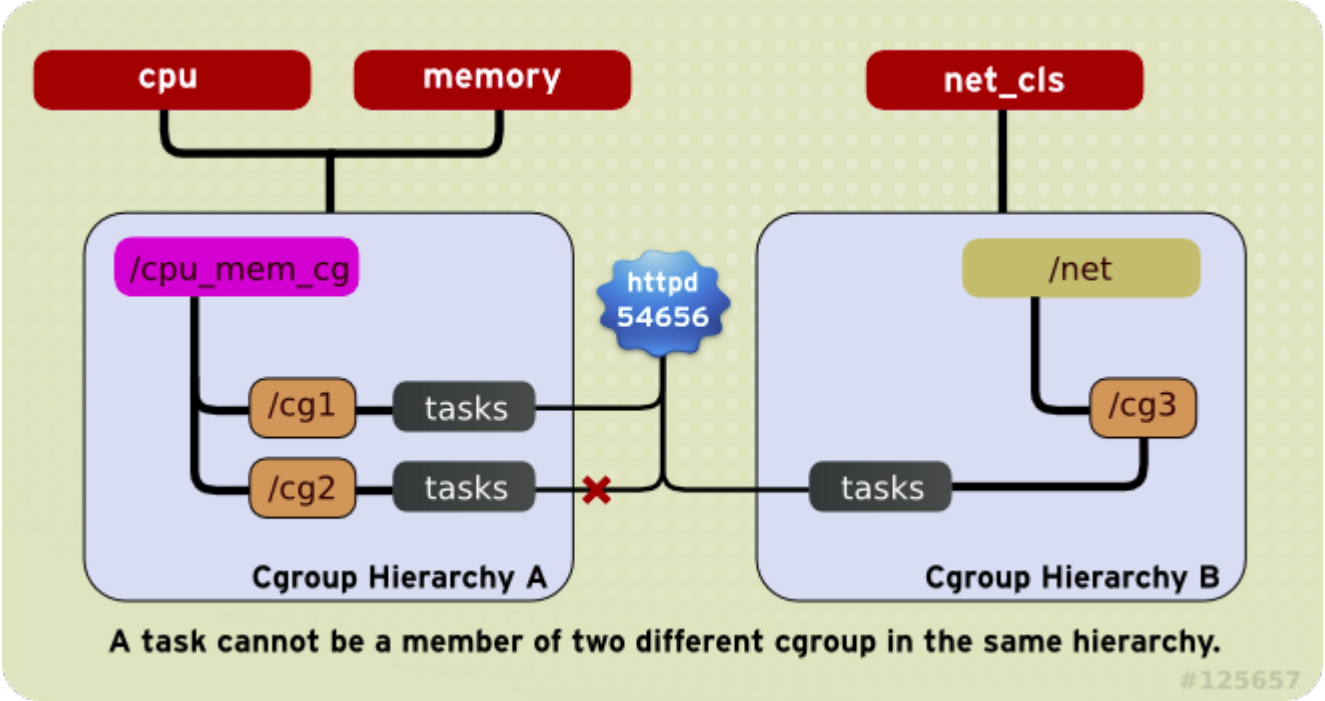
Source: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-relationships_between_subsystems_hierarchies_control_groups_and_tasks

Resource control: cgroups rules



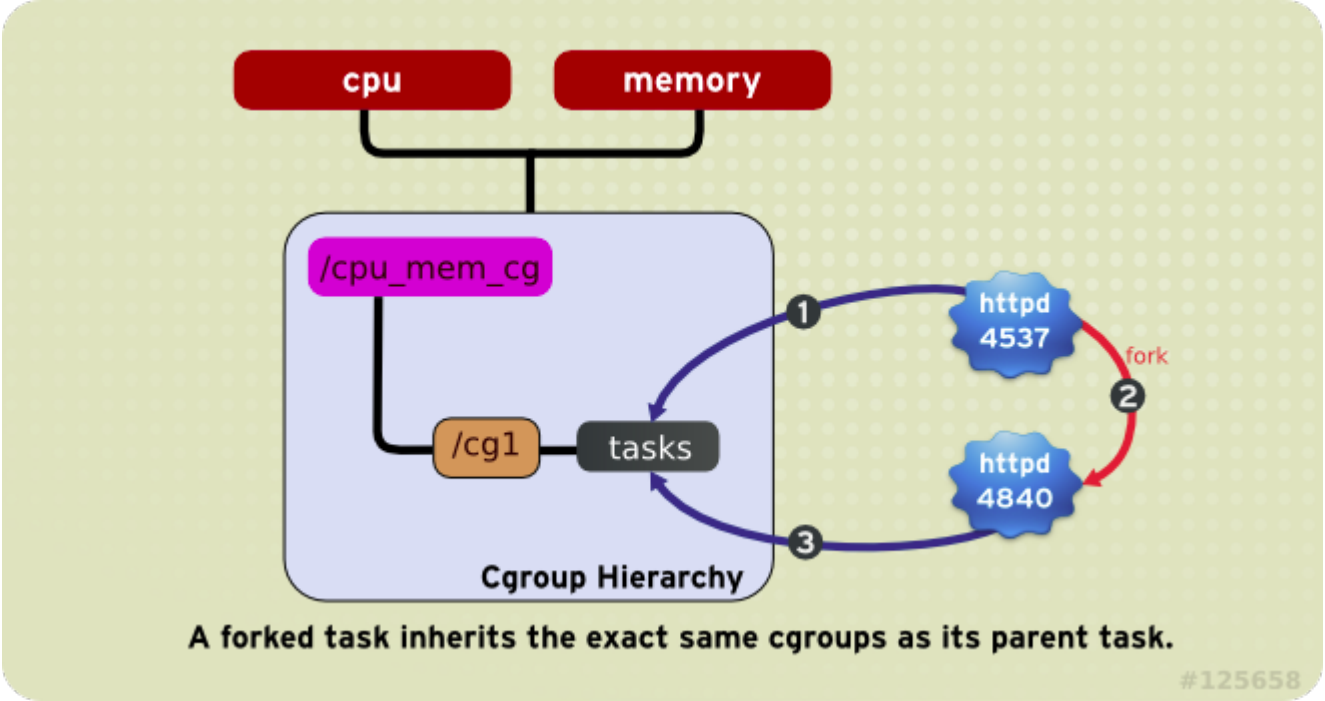
Source: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-relationships_between_subsystems_hierarchies_control_groups_and_tasks

Resource control: cgroups rules



Source: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-relationships_between_subsystems_hierarchies_control_groups_and_tasks

Resource control: cgroups rules



Source: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-relationships_between_subsystems_hierarchies_control_groups_and_tasks

Example: CPU limit

- As an example, let's see how we can limit the number of CPUs (cores) available to a process:
  - For this example we suppose that subsystems are already mounted and available under **`/sys/fs/cgroup/`**, otherwise run (as root) **`mount -t cgroup none /sys/fs/cgroup`**
  - First we create a new control group under **`/sys/fs/cgroup/cpuset`**

    ```
    cd /sys/fs/cgroup/cpuset
    mkdir mycontrolgroup
    cd mycontrolgroup
    ```

    - Note: inside the mycontrolgroup directory there will be some files related to the cpuset subsystem parameters
    - Check the number of available cores using /proc/cpuinfo

    Set the **cpuset.cpus** parameter, for example to limit execution on the first core: **`echo 0 > cpuset.cpus`**

Example: CPU limit (continued)

- Set the **cpuset.mems** parameter, to define the accessibile memory: **echo 0 > cpuset.mems**
- Now we can move the shell process into the mycontrolgroup

  ```
  echo $$ > tasks
  ```

  - the $$ variable corresponds to the PID of the current shell process
- Now the shell (and all subprocesses run from it) can only access one core:
  - You can check that with **nproc** (should return 1)
- When all processes in the cgroup mycontrolgroup terminate it's possible to remove the control group with:
  - **rmdir /sys/fs/cgroup/cpuset/mycontrolgroup**

Example: fork bomb mitigation

- As an example, let's see how we can limit the processes (PIDs) in a hierarchy:
  - For this example we suppose that subsystems are already mounted and available under **`/sys/fs/cgroup/`**, otherwise run (as root) **`mount -t cgroup none /sys/fs/cgroup`**
  - First we create a new control group under **`/sys/fs/cgroup/pids`**

    ```
    cd /sys/fs/cgroup/pids
    mkdir mycontrolgroup
    cd mycontrolgroup
    ```

    - Note: inside the mycontrolgroup directory there will be some files related to the pids subsystem parameters

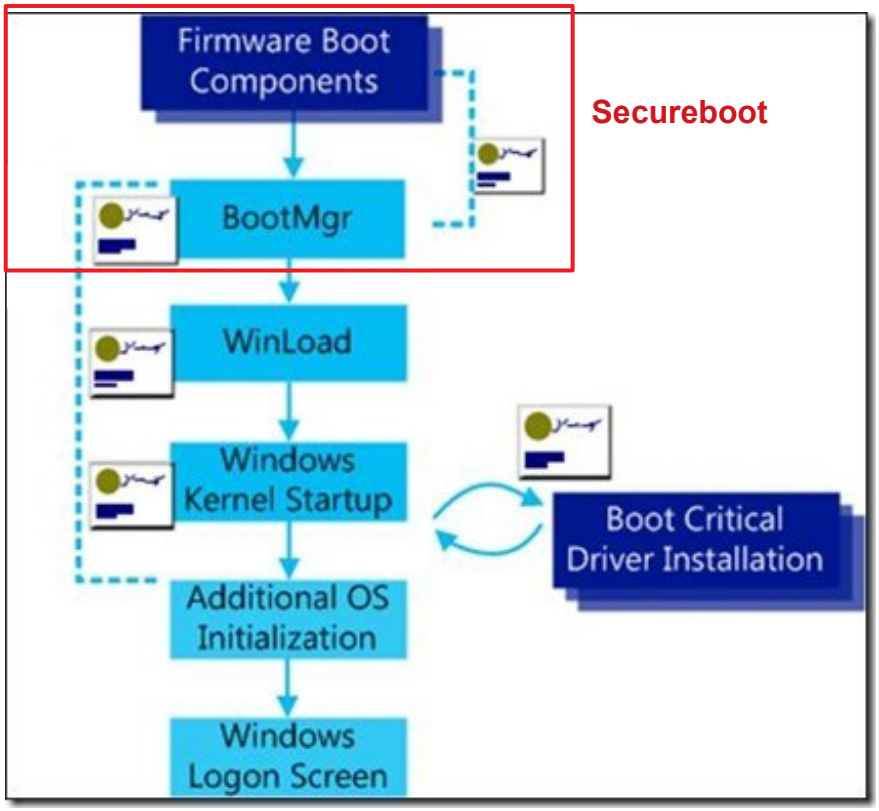
    Set the **pids.max** parameter, for example to limit it to 10 PIDs:

    ```
    echo 10 > pids.max
    ```
  - Now we can move the shell process into the mycontrolgroup

    ```
    echo $$ > tasks
    ```
  - … and now try with a fork bomb: **`:(){ :|: & };:`**

Chain of trust: UEFI Secureboot

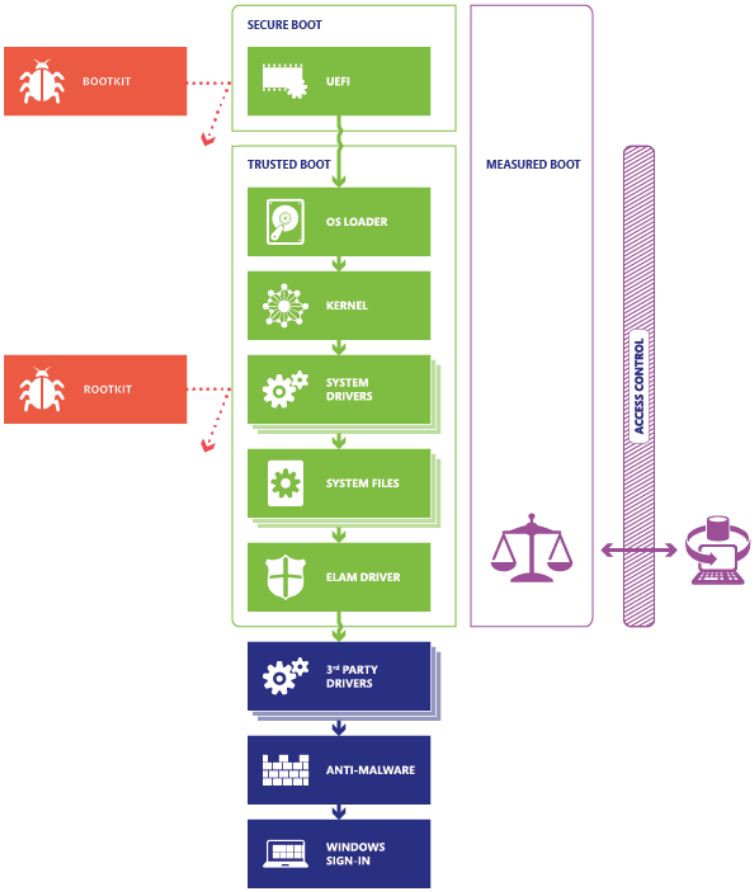
- The Unified Extensible Firmware Interface (UEFI) secure boot prevents booting of an untrusted (or tampered) operating system
  - this mechanism requires a bootloader signed with a key trusted by the secure boot implementation



(source: https://blogs.msdn.microsoft.com/olivnie/2013/01/09/windows-8-trusted-boot-secure-boot-measured-boot/)

Chain of trust: Windows Trusted boot

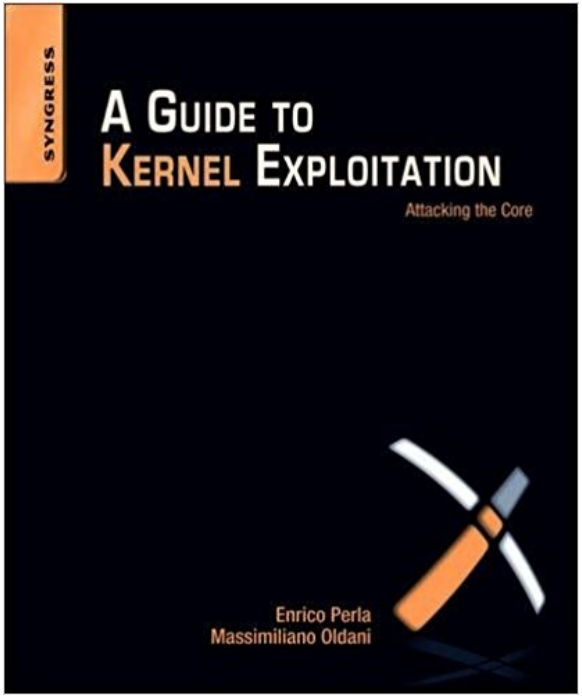
- If the bootloader has been verified, boot continues and trusted boot takes over:
  - it checks the digital signature of the Windows kernel, which in turn checks every other component
  - it starts the Early Launch Anti-Malware component (ELAM) which checks every boot driver
    - if available, a **TPM** (Trusted Platform Module) can be used for Measured Boot in order to create a digitally signed log of the machine's boot sequence for further verification



(source: https://docs.microsoft.com/en-us/windows/threat-protection/secure-the-windows-10-boot-process)

Attacks and countermeasures...

- Security in operating systems is still a very active topic
- Several other security issues need to be considered and prevented
  - Example: NULL dereference, stack smashing



a nice starting point…
“A guide to kernel exploitation”
Enrico Perla
Massimiliano Oldani

Syngress
ISBN-10: 1597494860
ISBN-13: 978-1597494861