Fundamentos de Segurança Informática (FSI)

2024/2025 - LEIC

Systems Security (Part 3)

Hugo Pacheco

hpacheco@fc.up.pt

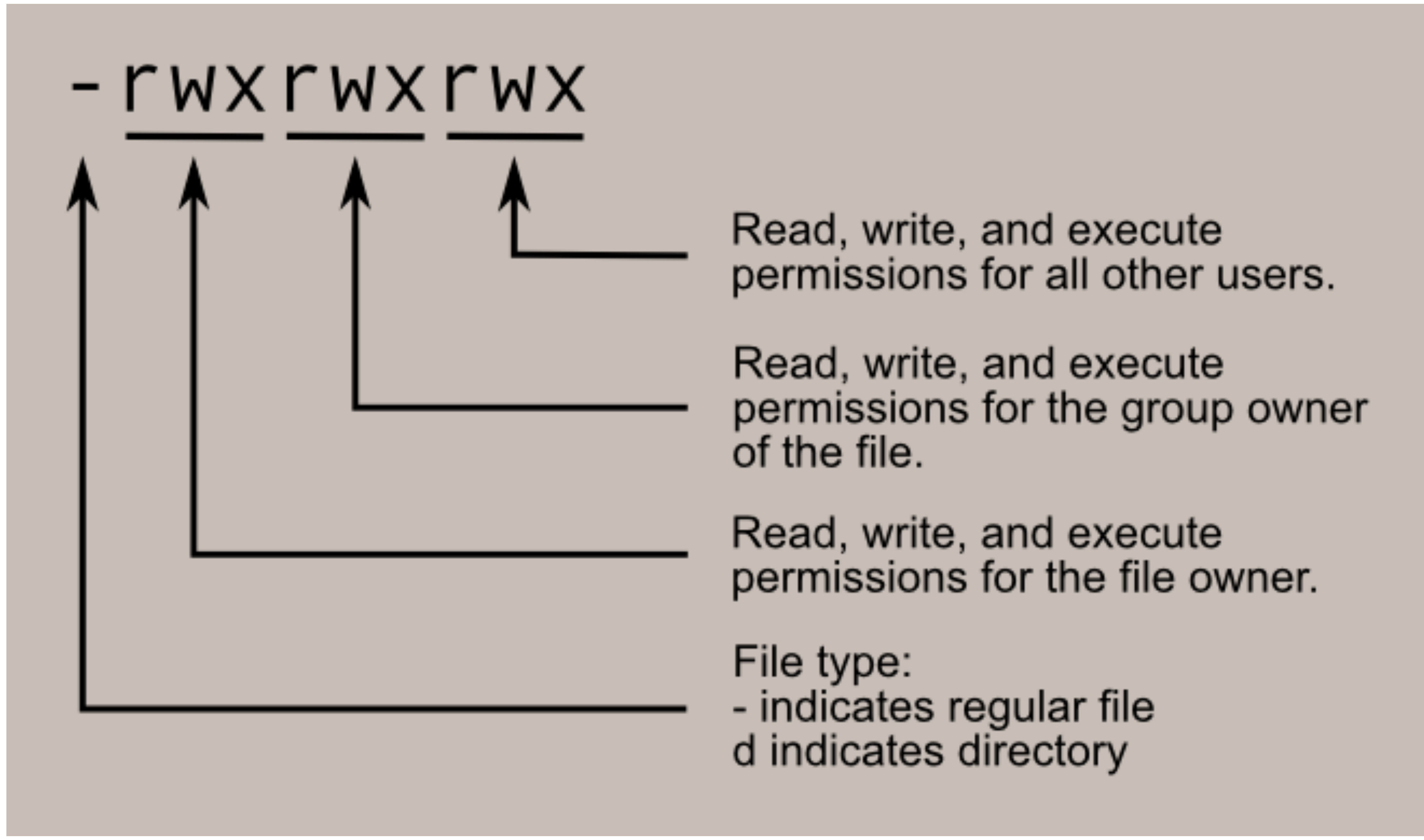
File System

File System

- We will use *n?x systems as an example:
  - **actors**: users, processes
  - **resources**: files and folders
  - **actions**/accesses:
    - r/w/x for files: clear meaning
    - r/w/x for folders: listing content, creating new content, “entering” folder
    - changing permissions?

File System

- Each **user** belongs to a **group**: enables a kind of **RBAC**
- Each **resource** has an **owner** and a **group**
  - **Permissions** are given independently to
    - **owner**
    - **group members**
    - **all other users**



File System

- Many *n?x filesystems support **Extended ACL**:
  - extend the base RBAC with ACLs beyond the simple user/group/other
    - named users / groups
    - mask denotes maximum permissions for applications not supporting Extended ACL

Means there are extended ACL permissions!

```
$ ls -l myfile
-rw-r-----+ ... tux linux ... myfile

$ getfacl myfile

# file: myfile
# owner: tux
# group: linux
user::rw-
group::r-x              # effective:r--
group:penguins:r-x   # effective:r--
mask::r--
other::---

$ setfacl -m u:nobody:rw myfile
```

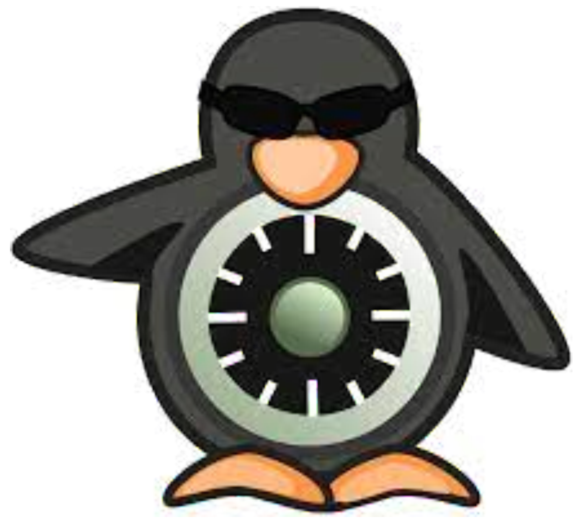
https://www.novell.com/documentation/suse91/suselinux-adminguide/html/apbs03.html

File System

- Superuser:
  - Used to be a special user (`root`)
  - nowadays a role: `sudo`
  - `uid = 0` used to identify such user/role
  - best practice: minimum superuser usage (least privilege 🕮)

File System

- Changing permissions:
  - Everything allowed to the **superuser** / administrator
  - The **owner** can be changed by **superuser** (`chown`)
  - The **group** can be changed by the **owner** or the **superuser** (`chgrp`)
- **Discretionary Access Control:** the **owner** can change permissions (`chmod`)
- **Mandatory Access Control:** only the **superuser** can
  - e.g., SELinux (Security-Enhanced Linux)



File System

- **Process** permissions:
  - **users** interact with the system via **processes**
  - each **process** has an associated **effective user id**
    - Determines the **permissions** of the **process**
    - Usually the **uid of the user** who launched the process
    - Some **exceptions:** e.g., `sudo` or changing a password with `passwd`

File System

- How does **user login** work?
  - The system executes a login process as root
  - That process authenticates the user (can read the credentials stored in the system)
  - Changes its own `uid` and `gid` to those of the user
  - Launches the `shell` process
- **Critical: the login process has to drop privileges**
  - If incorrectly implemented, vulnerable to capability leaking
- The reverse (**escalating privileges**) should be **impossible**… but what about `passwd`?

File System

- The `setuid` **bit** associated to a file (remember SEED LABS):
  - Allows binding the **user under which a process is run** to the **owner** of the executable (and not to the **user** who actually executes it)
  - May be activated by the superuser and by the file's owner
  - Implications:
    - If the owner has many privileges ⇒ **privilege escalation!**
  - For the case of `passwd`, the owner is **root**

Privilege Escalation

- There are many real-world examples of privilege escalation
  ⇒ **abusing setuid programs**

- CVE-2004-0360 (passwd)

```
wks111% passwd
passwd: Changing password for joeuser
Enter existing login password:
New Password:
Re-enter new Password:
passwd: password successfully changed for joeuser
wks111%
```

The vulnerability lies in the second set of entered characters, the "new password." The program does not check the length of the entered string to make sure that it will fit in the memory that has been allocated to it. This provides an opportunity to overwrite areas of memory that are being used by the program to store other values.

https://vulmon.com/vulnerabilitydetails?qid=CVE-2004-0360

https://www.giac.org/paper/gcih/700/local-privilege-escalation-solaris-8-solaris-9-buffer-overflow-passwd1/105309

- CVE-2023-22809 (sudoedit)

```
Example: EDITOR='nano -- /etc/passwd' sudoedit /etc/motd
```

Description

A vulnerability was found in sudo. Exposure in how sudoedit handles user-provided environment variables leads to arbitrary file writing with privileges of the RunAs user (usually root). The prerequisite for exploitation is that the current user must be authorized by the sudoers policy to edit a file using sudoedit.

https://access.redhat.com/security/cve/cve-2023-22809

https://hamzakhattak.medium.com/sudo-vulnerability-in-linux-lead-to-privilege-escalation-cve-2023-22809-fbb7f300ef49

Process Privileges

- **When executing a process, it typically executes with the UID of the user who launched it**
    - Can access the resources of such user
- Some processes are **executed with the UID of the owner** of the file (setuid bit = 1)
- Kernel processes are launched with UID = 0 (root)
    - access to all resources ⇒ maximum privilege!

Process Privileges

- The transition of privileges is more subtle
- A process has, in fact, three UIDs:
  - **Effective User ID (EUID)**: determines the permissions
  - **Real User ID (RUID)**: user who launched the process
  - **Saved User ID (SUID)**: previous user, to model privilege transitions

Process Privileges

- What can change during execution?
- **The root user** may use the `setuid(x)` system call to change these values to arbitrary UIDs:
    - `EUID := x, RUID := x, SUID := x`
- **Normal users** can only change `EUID` to `RUID` (themselves) or to `SUID` (going backward)
- This allows a process to **permanently** reduce its privileges:
    - e.g., when Apache (runs as root to use port 80) forks a process to listen to user requests, it reduces the privileges of the forked process

Process Privileges

- A process can **temporarily** reduce its privileges:
- The `seteuid(x)` system call changes only the EUID and preserves the RUID and the SUID (e.g., daemon may later use the RUID to access a resource)
  - `EUID := x, RUID unchanged, SUID unchanged`
- Typical scenario: downgrade privileges ⇒ execute code ⇒ restore privileges
- **Danger:** using `seteuid` when root intends to permanently drop privileges ⟹ normal user can return to `RUID` using `setuid`!

OpenSSH UseLogin SetUID Vulnerability

Severity	CVSS	Published	Created	Added	Modified
10	(AV:N/AC:L/Au:N/C:C/I:C/A:C)	06/08/2000	07/25/2018	11/01/2004	11/09/2017

Description

OpenSSH does not properly drop privileges when the UseLogin option is enabled, which allows local users to execute arbitrary commands by providing the command to the ssh daemon.

CVE-2000-0525
(using `seteuid` instead of `setuid`)

Quiz

- Remember that we have three UIDs: **EUID** + **RUID** + **SUID**
- What is the purpose of the **SUID**?
  - If a **user** launches a **root** process with the setuid bit on:
    - On launch: **EUID** = **root**; **RUID** = **user**; **SUID** = **root**
    - After `seteuid`(**user**): **EUID** = **user**; **RUID** = **user**; **SUID** = **root**
    - Can return to **root** with `seteuid`(**root**)

Capability leaking 

- Even when using `setuid`, to permanently drop privileges, previously obtained capabilities are automatically not revoked

```
void main() {
   int fd;
   /* Assume that /etc/zzz is an important system file, and it is owned by root with permission
0644. Before running this program, you should create the file /etc/zzz first. */
   fd = open("/etc/zzz", O_RDWR | O_APPEND);
   printf("fd is %d\n", fd);
   if (fd == -1)    {
      printf("Cannot open /etc/zzz\n");
      exit(0);      }
   // Permanently disable the privilege by making the effective uid the same as the real uid
   setuid(getuid());
   printf("Real user id is %d\n", getuid());
   printf("Effective user id is %d\n", geteuid());
   write(fd, "Malicious Data\n", 15);   }
```

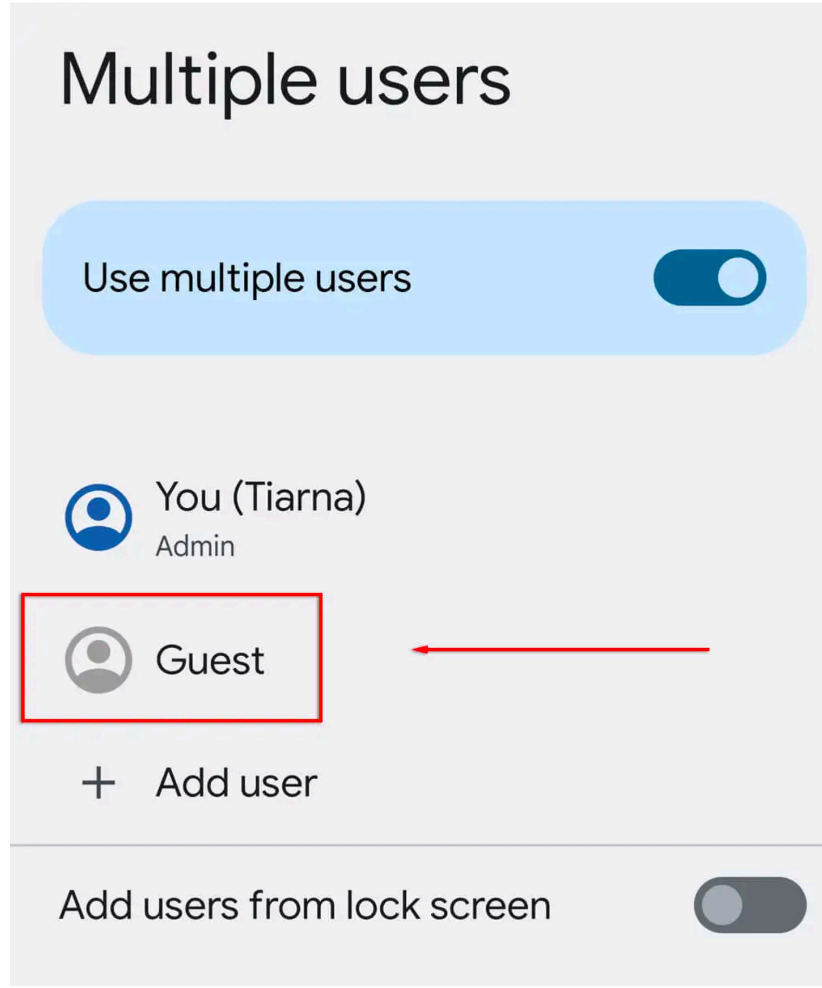
- Task 9 from the Environment Variable and Set-UID Program SEED LABS

File System

- **Everything is a file** (economy of mechanism):
    - How to minimize the number of system calls and the attack surface?
        - Use the same interface for the file system and other resources
    - In *n?x: sockets, pipes, I/O devices, kernel objects, etc.
    - **The access control mechanisms are always the same!**

File System (Android)

- Android systems execute on top of a Linux sub-system (SELinux), with **MAC**:
  - Restricting access to apps and resources
  - Solution: each app has its own user
  - Problem: multiple users? (e.g. in tablets)
    - Ad-hoc solution: `u1_a23`



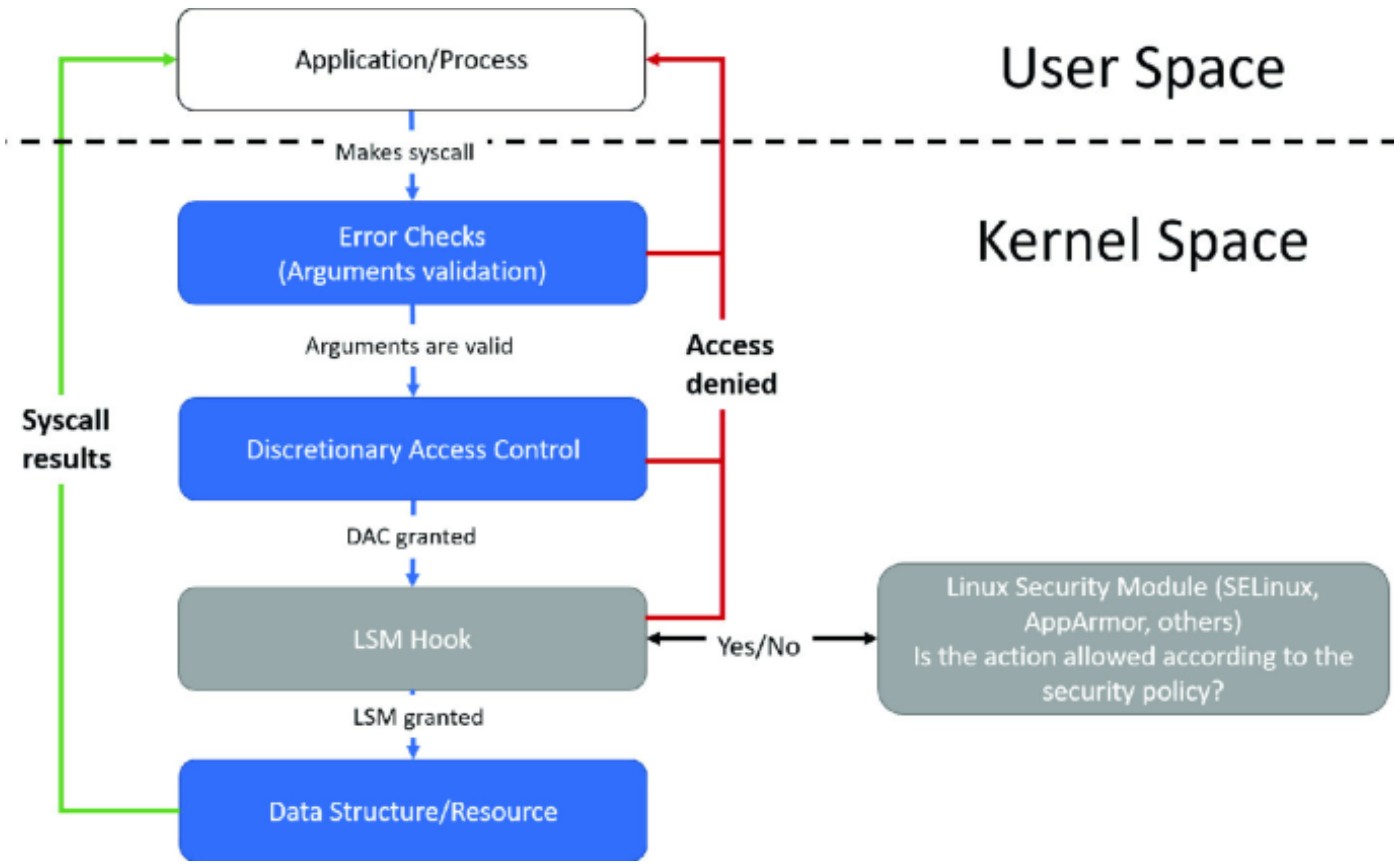
- Additionally: **Manifest Permissions** per app

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

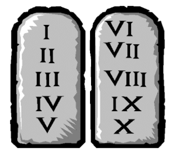
https://android.stackexchange.com/questions/207725/are-multiple-users-protected-from-each-other-differently-than-apps

File System (LSM)

- **Linux Security Modules** (since kernel 2.6)
- Access control models implemented as Loadable Kernel Modules on top of regular **DAC**
- A truly generic reference monitor



Process Privileges

- **Complexity:**
  - Even with such a simple system …
  - … there is a mechanism of transitions between states of trust …
  - … where it is really easy to make mistakes.
- **Important:** compromise recording 

File System

- The access control mechanism in *n?x systems is essentially an implementation of Access Control Lists, with some batching (RBAC)
- **Advantage:** simple (without extended ACL + LSM) and works in practice
- **Disadvantage:** not very robust and flexible
  - A failure in a process such as `passwd` or `ssh (euid = 0)` has catastrophic consequences
  - `root` used (often wrongly) for many purposes ⇒ administration errors
  - Once an attacker gains `root`, there is no way to downgrade his privileges

Confinement

Why confinement?

- It is commonly necessary to **execute untrusted code** in a **trusted platform**:
  - Code originating for external sources, e.g., websites:
    - Javascript, browser extensions, applications, etc
  - Legacy code that is not up to modern standards
  - *Honeypots*, forensic *malware* analysis, etc
- **Goal**: if the **code "misbehaves"** ⇒ ***nuke it*!**

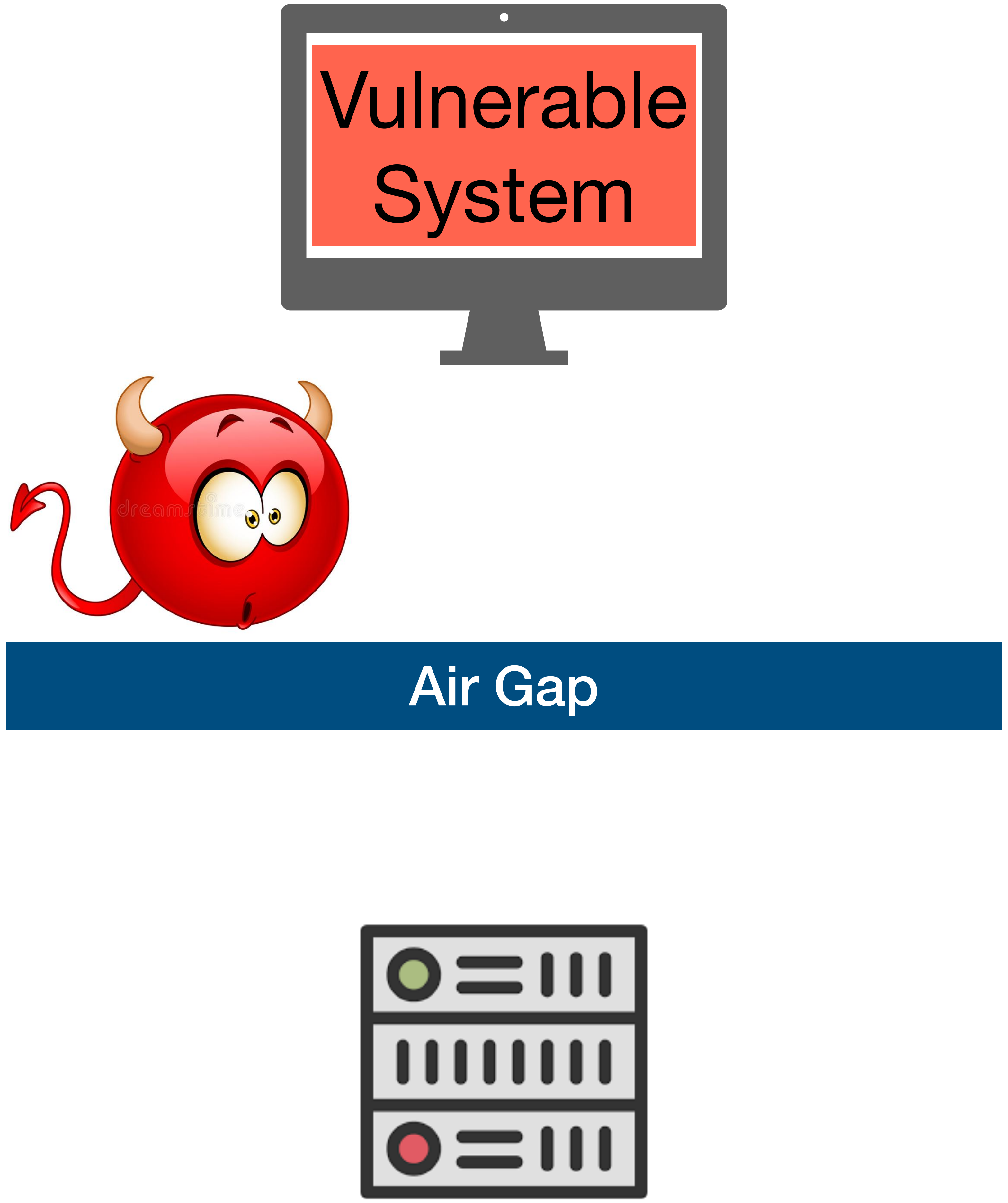
Confinement: Air Gap

- Solution: **guarantee that the potentially malicious code cannot affect the rest of the system**

- May be implemented at various levels, starting in the HW

- When confinamento is physical at the level of HW ⇒ ***airgap***

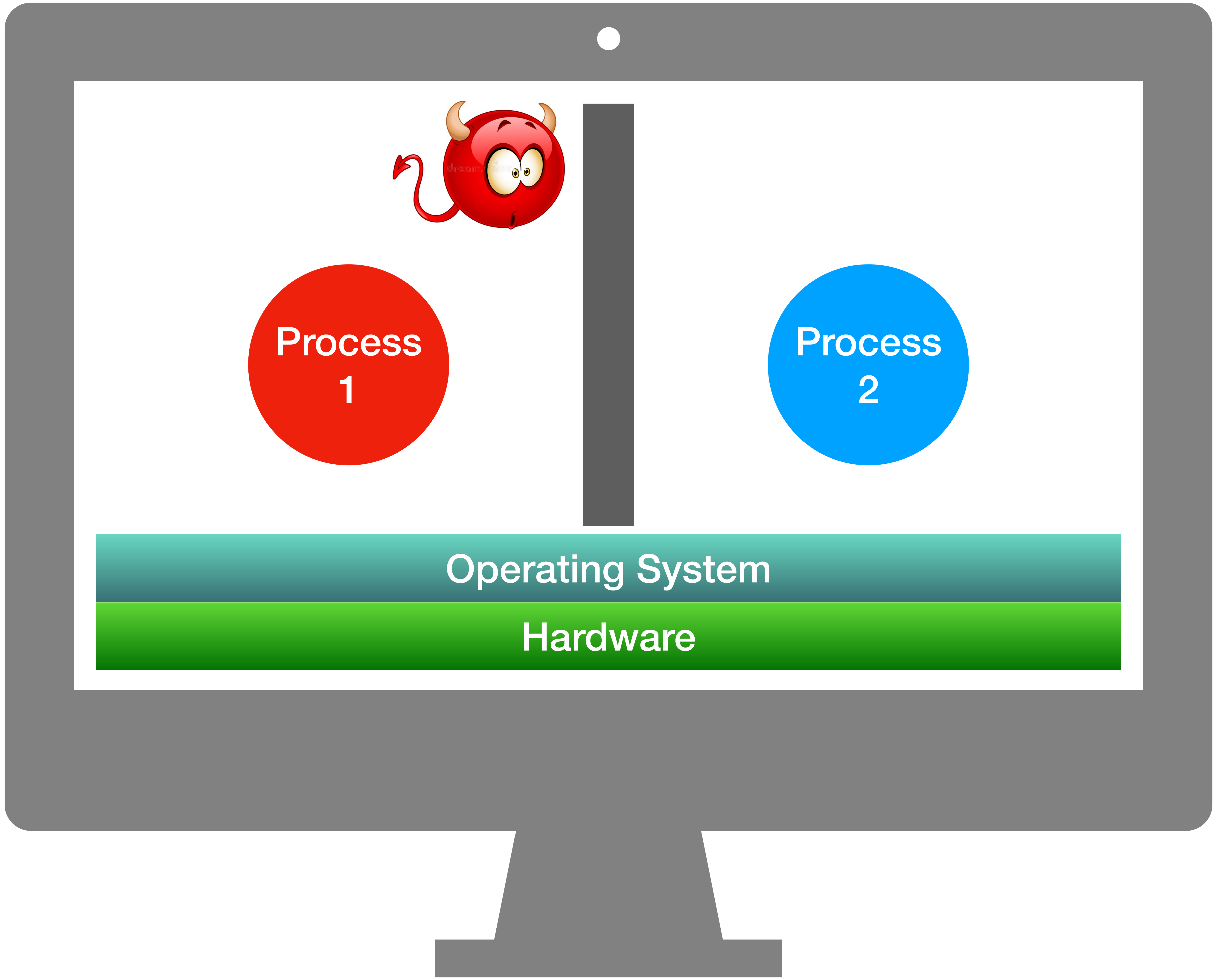
- Disadvantage: hard to manage



https://www.schneier.com/blog/archives/2013/10/air_gaps.html

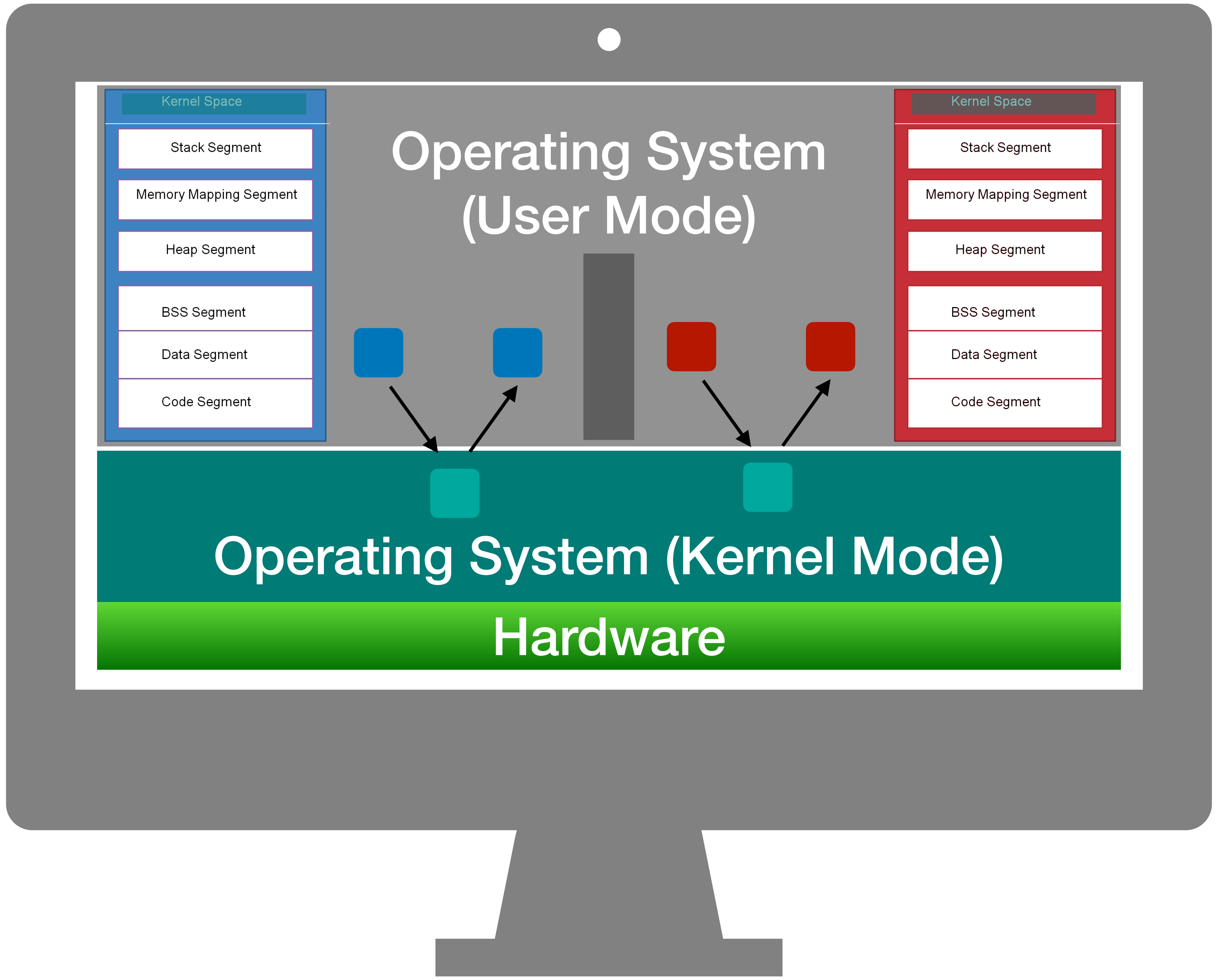
Confinement: Processes

- The OS may share HW:
  - Offers a virtual view of memory/resources to each user/process
  - Ensures that the actions of P1 do not affect the context of P2 and vice-versa
- **Challenges:**
  - Processes from the same user
  - Processes as administrator
  - OS vulnerabilities



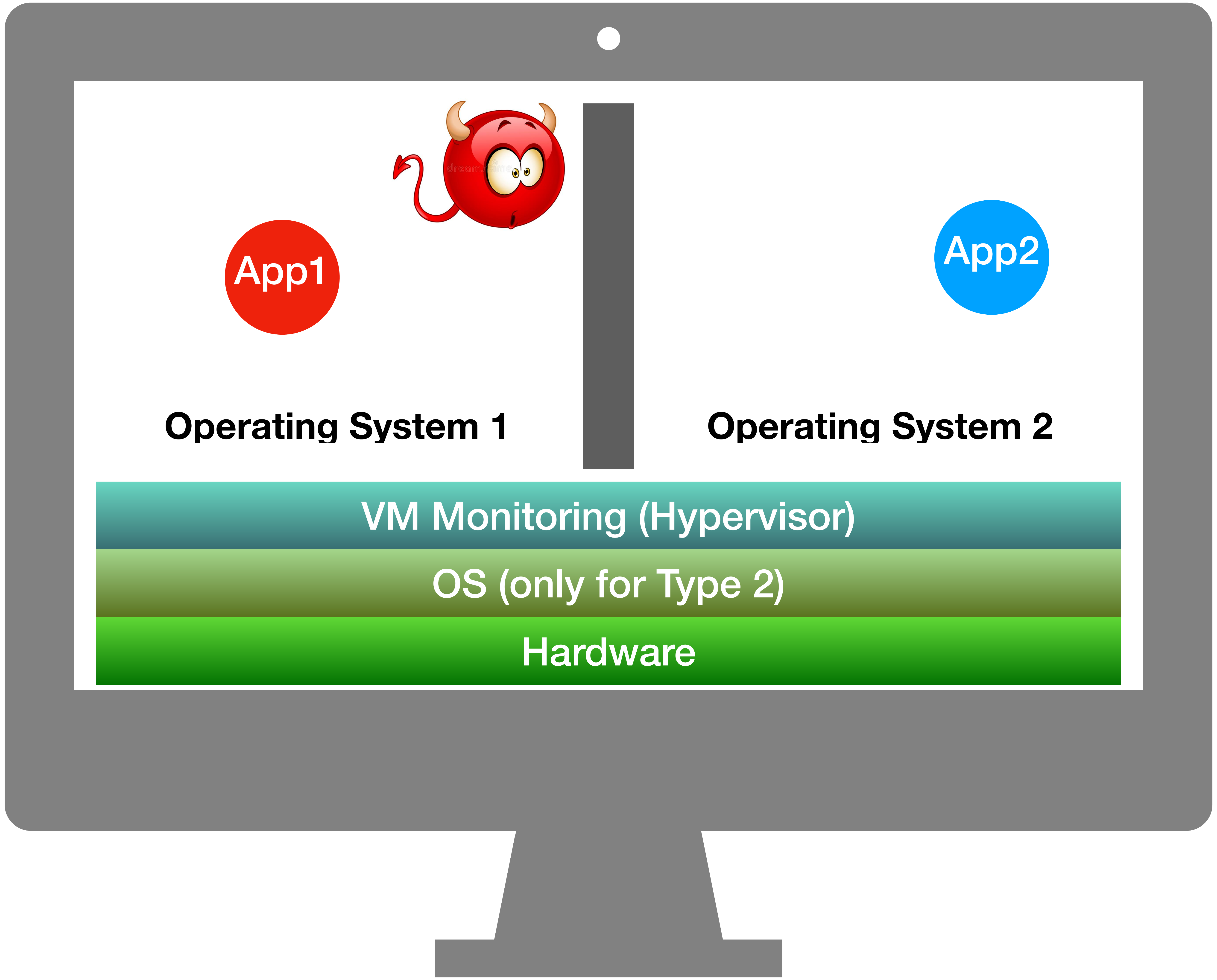
Confinement: SFI + SCI

- **Software Fault Isolation (SFI):** isolation of processes sharing userland address space
- **System Call Interposition (SCI):** mediation of all system calls
- In *n?x, we have already seen SFI/SCI:
  - **Memory Isolation:** virtualising the address space and monitoring the address translation mechanisms
  - **Kernel vs Userland Separation:** offer a limited number of system calls and map its control to universal access control mechanisms
- Altogether realise the **reference monitor concept**



Confinement: Virtual Machines

- **The Hypervisor allows sharing HW:**
  - Offers a virtual view of the HW to each OS
  - Ensures that actions in OS1 do not affect actions in OS2 and vice-versa
- **Two main kinds of Hypervisor:**
  - **"Type 1"** or "bare metal": Thin OS over HW (e.g., clouds)
  - **"Type 2"** or "hosted": software layer over OS (e.g., personal computers)

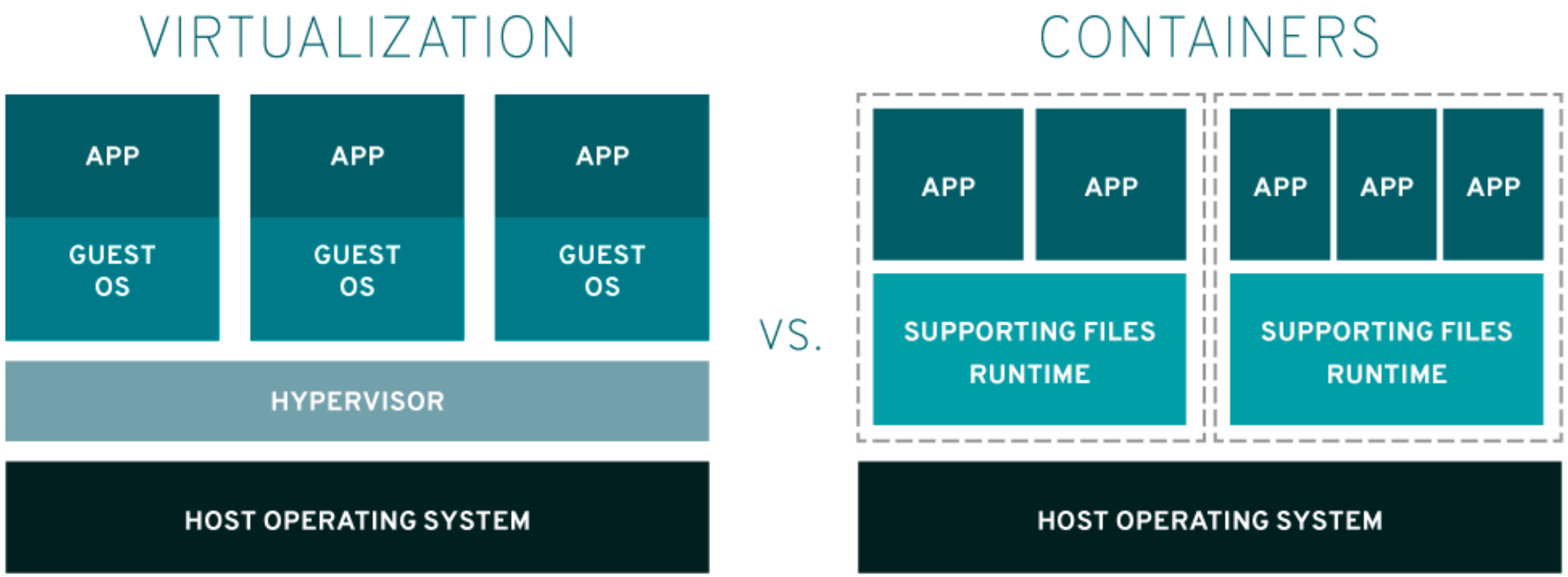


Confinement: Sandboxing

- Additional confinement within an application
- E.g., browsers are complex applications:
  - They internally create an execution environment for code originating from external sources
  - JavaScript/WebAssembly interpreters with monitoring mechanisms

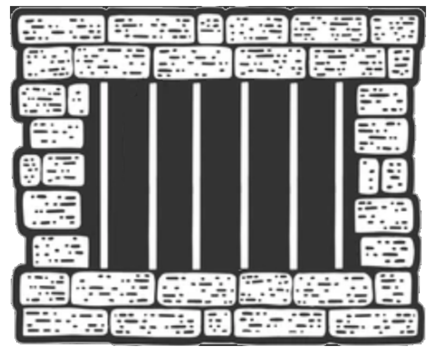
Confinement: Containers

- **Virtual Machines** run entire OSes over the HW and/or other OSes (+ isolation)
- **Containers** share the *kernel* of the host OS and isolate userland components (- resources)



https://www.redhat.com/en/topics/containers/whats-a-linux-container

https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016

Old example: chroot 

- The `chroot` command (since 1979) allows creating **jails**
  - Can only be used by `root`
  - Transforms the **shell environment** seen by users/processes:
    - The current folder becomes the root of the filesystem
    - System calls are intercepted and paths are appended the corresponding prefix
    - **Rationale:** applications cannot access files outside the current folder

```
$ chroot /tmp/guest
$ su guest
```

The root of the filesystem becomes `/tmp/guest`.

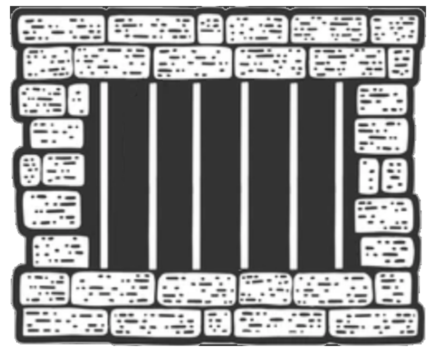
The effective user inside the shell becomes `guest`.

`The command:`

`fopen(“/etc/passwd”, “r”)`

becomes

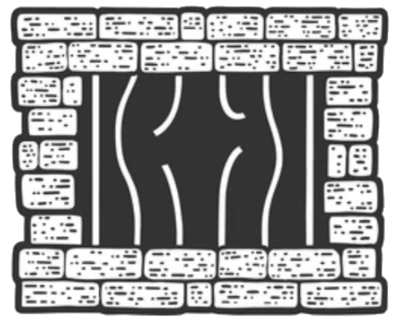
`fopen(“/tmp/guest/etc/passwd”, “r”)`

Old example: chroot 

- We generally want to give a user/application inside a jail:
  - An environment with access to utilities such as `ls`, `ps`, `vi`, etc.
- The **jailkit** utility allows to configure an isolated environment with control over the kinds of allowed tasks:
  - Initialising the environment from an initial configuration
  - Launching a *shell* that allows accessing the configured resources
- **Drawback:** a simple `chroot` jail does not restrict network communication

Escaping a jail

- Initially:
  - Relative paths: `fopen( "../../etc/passwd", "r")`
  - Allowed to execute: `fopen("/tmp/guest/../../etc/passwd", "r")`
- A non-root user that can execute chroot can create its own passwords file and become `root` ⇒ vulnerability in Ultrix 4.0
  1. Create a file `/aaa/etc/passwd`
  2. Execute `chroot /aaa`
  3. Executing `su root`, which password will be requested?

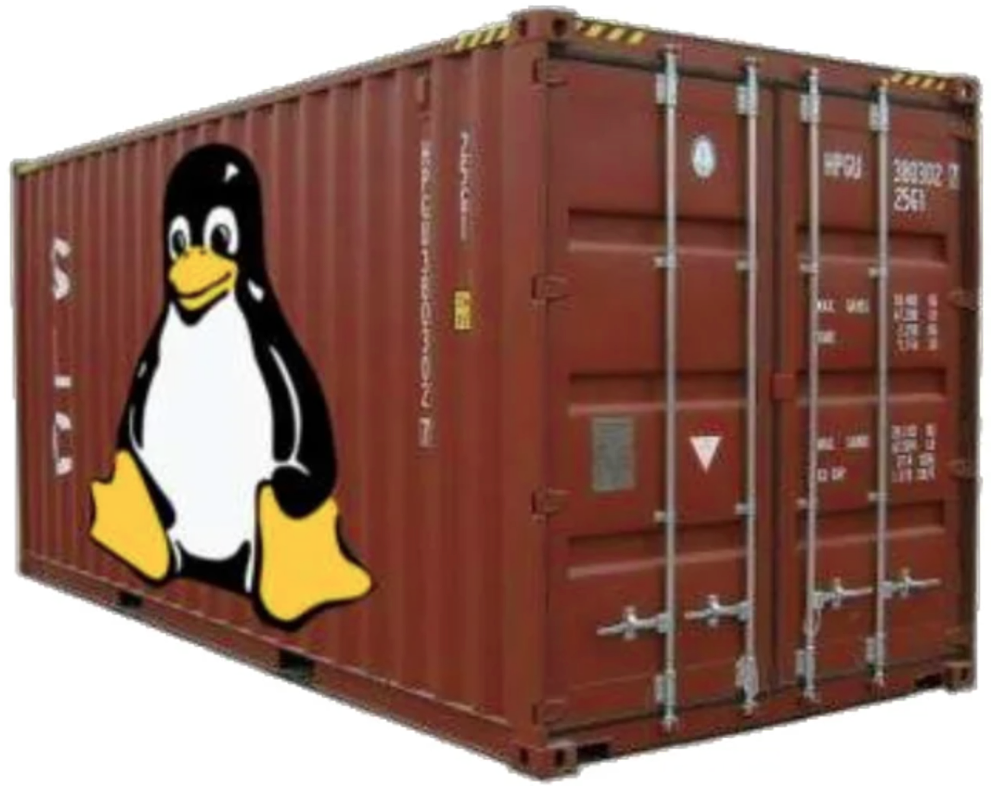
Escaping a jail 

- It is **critical** that the **user inside a jail cannot become root**
- Otherwise, there are numerous ways to escape:
  - Creating a device to access the raw disk
  - Sending signals to processes that are not inside the jail
  - Rebooting the system
  - etc.

Nowadays: FreeBSD jails

- More elaborate than the original `chroot`
- Launched with FreeBSD 4.0 (2000)
    - **Restricts** network connections and inter-process communication
    - **Restricts** root privileges inside the jail
    - First goal = confinement, Evolution ⇒ quasi-virtualisation
- **Still drawbacks:**
    - Often inflexible policies (e.g., browser needs to read from disk to send mail attachments)
    - Applications remain in “direct contact” with the network and the kernel

Nowadays: Linux containers



- LXC: concept very similar to FreeBSD jails
- Adapted for Linux, slightly more recent (2008)
  - Unprivileged containers: root inside the container is a normal user outside the container
  - Kernel Control Groups (cgroups): **limits how many resources** (CPU, memory, network, etc) may be used by hierarchical organisations of processes
  - Kernel Namespaces: partitions kernel resources and **limits which kernel resources** different processes can see

Acknowledgements

- This lecture's slides have been inspired by the following lectures:
  - CSE127: System Security I + System Security II
  - CS155: Isolation and Sandboxing