

Fundamentos de Segurança Informática (FSI)

2022/2023 - LEIC

Manuel Barbosa
mbb@fc.up.pt

Hugo Pacheco
hpacheco@fc.up.pt

Aula 9

Segurança de Sistemas

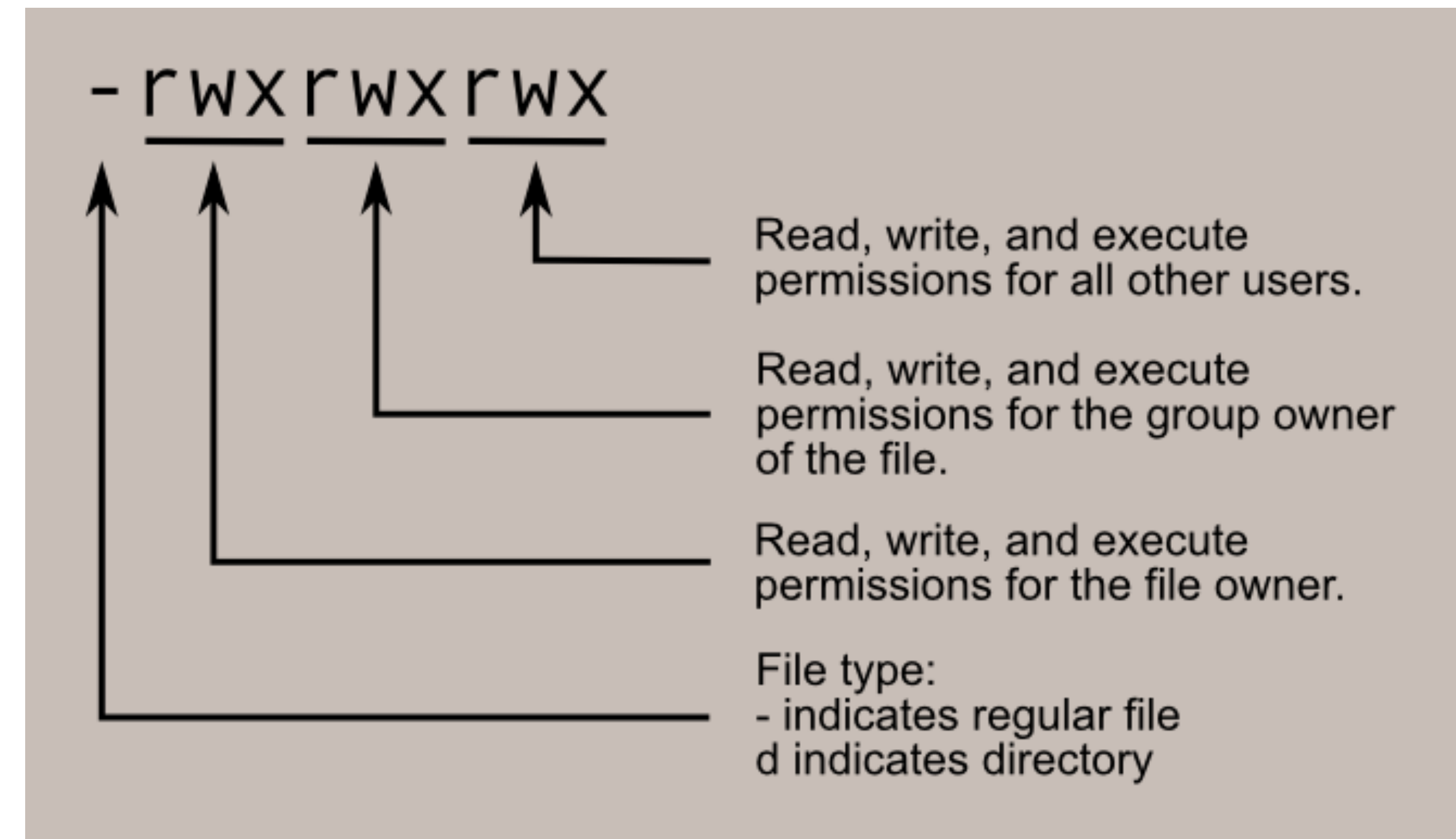
(Parte 3)

Sistema de Ficheiros

- Veremos como exemplo os sistemas *n?x:
 - **atores**: utilizadores, processos
 - **recursos**: ficheiros e pastas
 - **ações**/acessos:
 - r/w/x para ficheiros: evidente qual o significado
 - r/w/x para pastas: listar conteúdo, criar conteúdo adicional, "entrar" na pasta
 - alterar as permissões?

Sistema de Ficheiros

- Cada utilizador pertence a um grupo: permite uma forma de RBAC
- Cada recurso tem um *owner* e um *grupo*
 - as permissões são atribuídas de forma independente a
 - owner (ACL com uma única entrada)
 - membros do grupo associado ao recurso (RBAC rígido)
 - todos os outros utilizadores (RBAC rígido)



Sistema de Ficheiros

- Superuser:
 - antigamente um utilizador especial (*root*)
 - hoje em dia um papel/role: `sudo`
 - `uid = 0` utilizado para identificar esse utilizador/papel
 - boas práticas: utilização mínima (privilégio mínimo 📖📖)



Sistema de Ficheiros

- Alteração de permissões:
 - sempre permitido ao **superuser** / administrador
 - permissões podem ser alteradas pelo **owner** (chmod)
 - **owner** pode ser alterado pelo **superuser** (chown)
 - **grupo** poder ser alterado por **owner** e **superuser** (chgrp)
- **Discretionary Access Control:** o **owner** pode alterar permissões
- **Mandatory Access Control:** apenas o **superuser** pode (e.g., SELinux)

Sistema de Ficheiros

- Permissões de **processos**:
 - os **utilizadores** interagem com o sistema através de **processos**
 - cada **processo** tem associado um ***effective user id***
 - determina as permissões do processo
 - em geral: **uid do utilizador** que lançou o processo
 - **existem exceções**: e.g., sudo ou mudar a password usando `passwd`


Sistema de Ficheiros

- Como funciona o login?
 - o sistema executa um processo login como root/super user
 - esse processo **autentica o utilizador** (tem acesso às credenciais no sistema)
 - altera o seu próprio `uid` e `gid` para os associados ao utilizador
 - lança o processo de `shell`
- **Crítico:** o login executa *drop privileges*
 - Se incorretamente implementado, continua a ser possível *capability leaking* 😈
- O reverso (*elevate privileges*) deve ser **impossível** (e o `passwd`?)

Sistema de Ficheiros

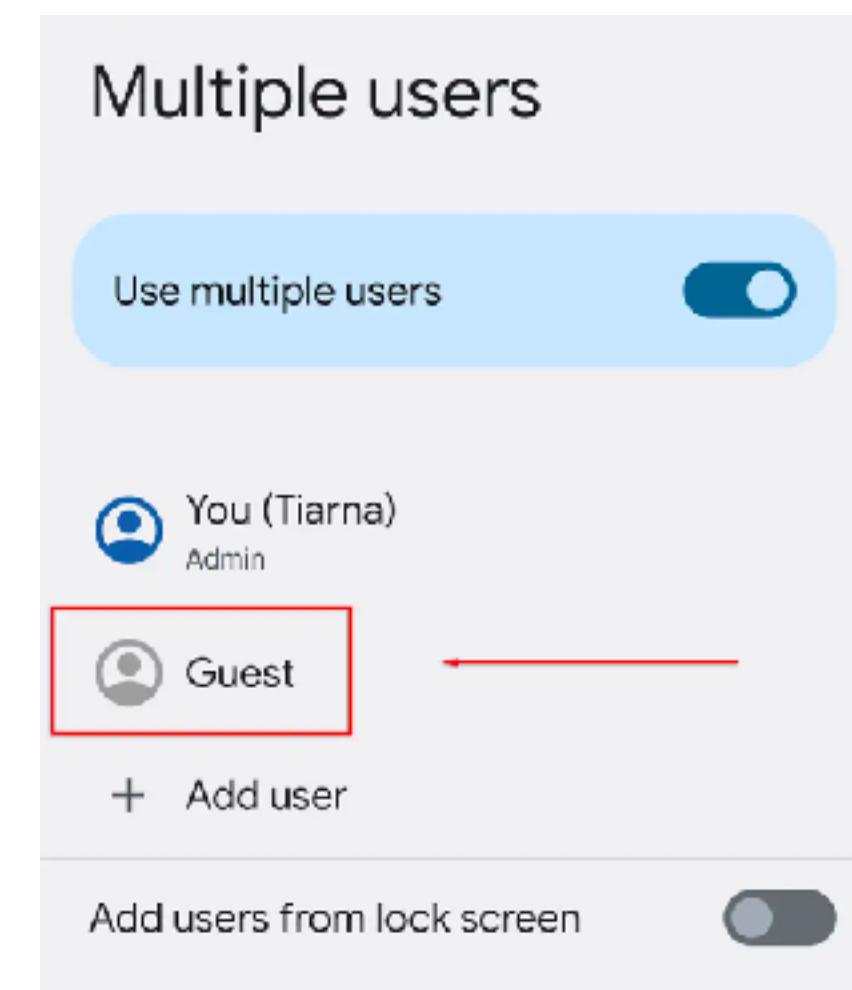
- O `bit setuid` associado a um ficheiro:
 - Permite fixar o utilizador associado um processo ao owner do executável (e não ao utilizador que executa)
 - Pode ser ativado pelo *superuser* e pelo *owner* do ficheiro
 - Implicações:
 - se o *owner* tiver muitos privilégios \Rightarrow permite elevação de privilégios!
- No caso do `passwd` o *owner* é o utilizador *root*.

Sistema de Ficheiros

- **Tudo é um ficheiro** (economia de mecanismos - como minimizar o número de system calls/superfície de ataque?
- utilizar a mesma interface construída para o sistema de ficheiros para outros recursos
- Em *n?x: sockets, pipes, dispositivos de I/O, objetos do kernel, etc.
- **O sistema de controlo de acessos é sempre o mesmo!**

Exemplo de utilização: Android

- Os sistemas Android executam sobre um sub-sistema Linux (SELinux), com **MAC**:
 - restringir o acesso de aplicações a recursos
 - solução: cada aplicação tem o seu próprio utilizador
 - problema: múltiplos utilizadores? (e.g. em tablets)
 - solução ad-hoc: u1_a23



- Adicionalmente:
Manifest Permissions
por aplicação

```
<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" />
```

Privilégios de Processos

- Quando executamos um processo, tipicamente executa com o UID do utilizador que o lançou
 - pode aceder aos mesmos recursos
- Alguns processos são executados com o UID do owner do ficheiro executável (bit setuid = 1)
- Os processos do kernel arrancam com UID = 0 (root/super user)
 - acesso a todos os recursos \Rightarrow privilégio máximo!

Privilégios de Processos

- A transição de privilégios é mais complexa do que parece à partida
- Um processo tem, de facto, três UIDs:
 - **Effective User ID (EUID)**: determina as permissões
 - **Real User ID (RUID)**: utilizador que lançou o processo
 - **Saved User ID (SUID)**: utilizado em transições, lembra o anterior

Privilégios de Processos

- O que é possível fazer em tempo de execução?
- O utilizador *root* pode usar a system call `setuid(x)` para alterar estes valores para UIDs arbitrários:
 - `EUID` => `x`, `RUID` => `x`, `SUID` => `x`
- Utilizadores comuns apenas podem mudar `EUID` para `RUID` (eles próprios) ou `SUID` (andar para trás)
- Isto permite a um processo reduzir os próprios privilégios:
 - quando o Apache (corre como root para usar porta 80) cria um processo para atender um utilizador reduz os privilégios do processo descendente

Privilégios de Processos

- É possível fazer uma redução temporária de privilégios:
- A system call `seteuid(x)` altera apenas o **EUID** e preserva o **RUID** e o **SUID** (e.g., daemon precisa de usar **RUID** para criar um recurso)
- Utilização típica: baixar privilégios \Rightarrow executar código \Rightarrow restaurar privilégios
- Perigo: usar `seteuid` quando root pretende baixar privilégios permanentemente (porquê?)
 \Rightarrow é possível voltar para **RUID** usando `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

(usar `seteuid` em vez de `setuid`)

Capability leaking

(exemplo do SEED Lab)


```
void main() {
    int fd; char *v[2];
    /* 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);

    // Print out the file descriptor value
    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());

    // Execute /bin/sh
    v[0] = "/bin/sh"; v[1] = 0;
    execve(v[0], v, 0);
}
```

Privilégios de Processos

- **Complexidade:**
 - mesmo com um sistema tão simples
 - existe um sistema de transições entre estados de confiança
 - onde é muito fácil cometer erros
- **Importante:** registo de compromissos 

Conclusão

- O sistema de controlo de acessos em *n?x é essencialmente uma implementação de Access Control Lists, com algum *batching*
- Vantagem \Rightarrow simples e funciona na prática
- Desvantagem \Rightarrow pouco robusto e pouco flexível
 - uma falha num processo tipo *passwd* ou *ssh* (`eu id = 0`) tem consequências catastróficas
 - *root* utilizado para muita coisa \Rightarrow erros de administração
 - Um atacante ganhando root, não é possível tirar-lhe privilégios

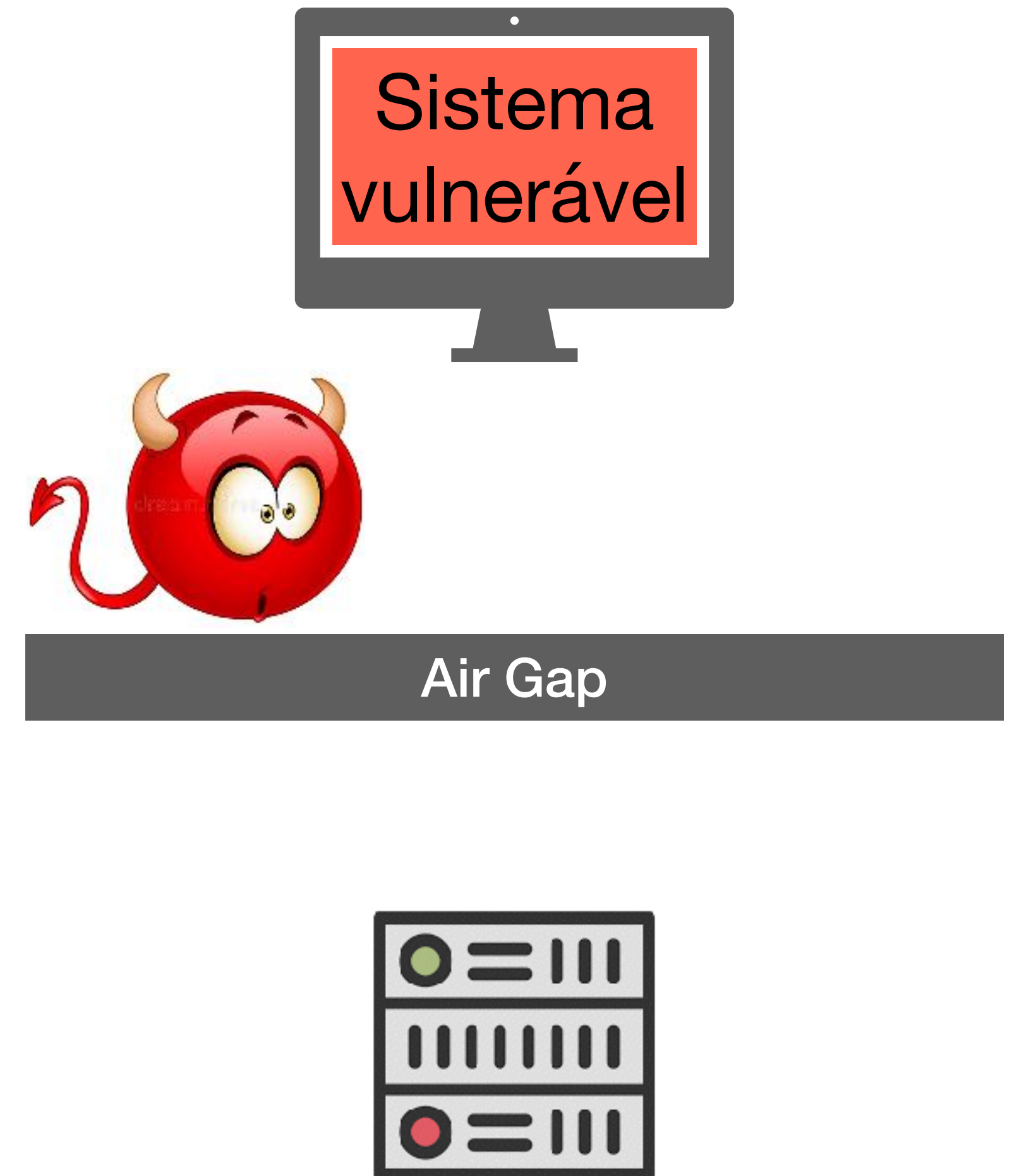
Confinamento (prelúdio)

Executar Código Não Confiável

- É comum ser necessário **executar código não confiável** numa **plataforma confiável**:
 - código proveniente de fontes externas, nomeadamente sites Internet:
 - Javascript, extensões de browsers, mas também aplicações
 - código legacy que sabemos não estar à altura das exigências atuais
 - *honeypots*, análise forense de *malware*, etc.
- **Objetivo**: se o **código se "portar mal"** \Rightarrow ***nuke it!***

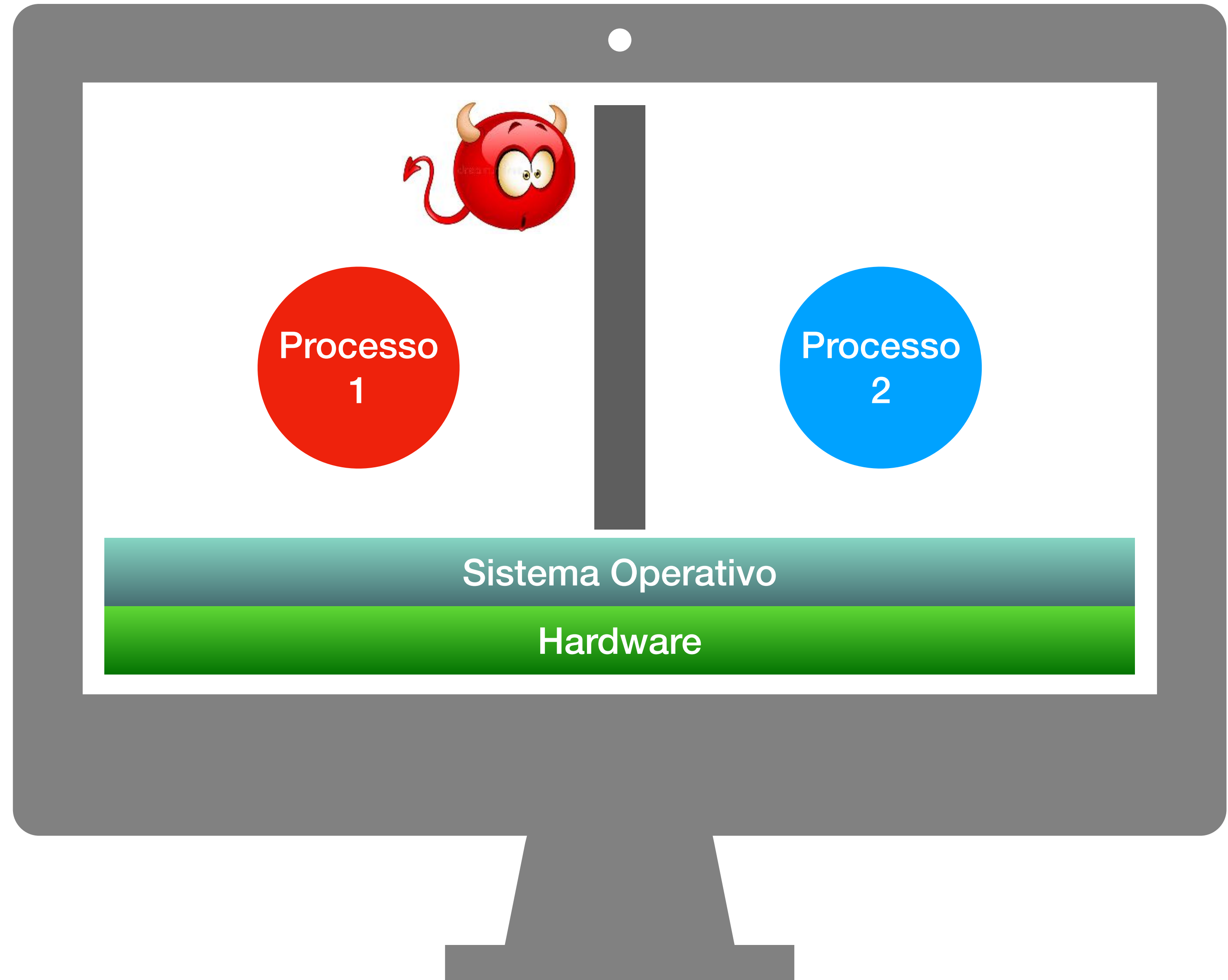
Confinamento: Air Gap

- Solução: **garantir que o código potencialmente malicioso não pode afetar o resto do sistema**
- Pode ser implementado a muitos níveis, começando no próprio hardware
- Quando o confinamento é efetuado ao nível do HW \Rightarrow ***airgap***
- Desvantagem: difícil de gerir



Confinamento: Processos

- O SO permite partilhar HW:
 - oferece visão virtual de memória/recursos a cada utilizador/processo
 - garante que as ações em P1 não afetam o contexto de P2 e vice-versa
- **Dificuldades de confinamento:**
 - processos do mesmo utilizador
 - processos como administrador
 - vulnerabilidades do SO

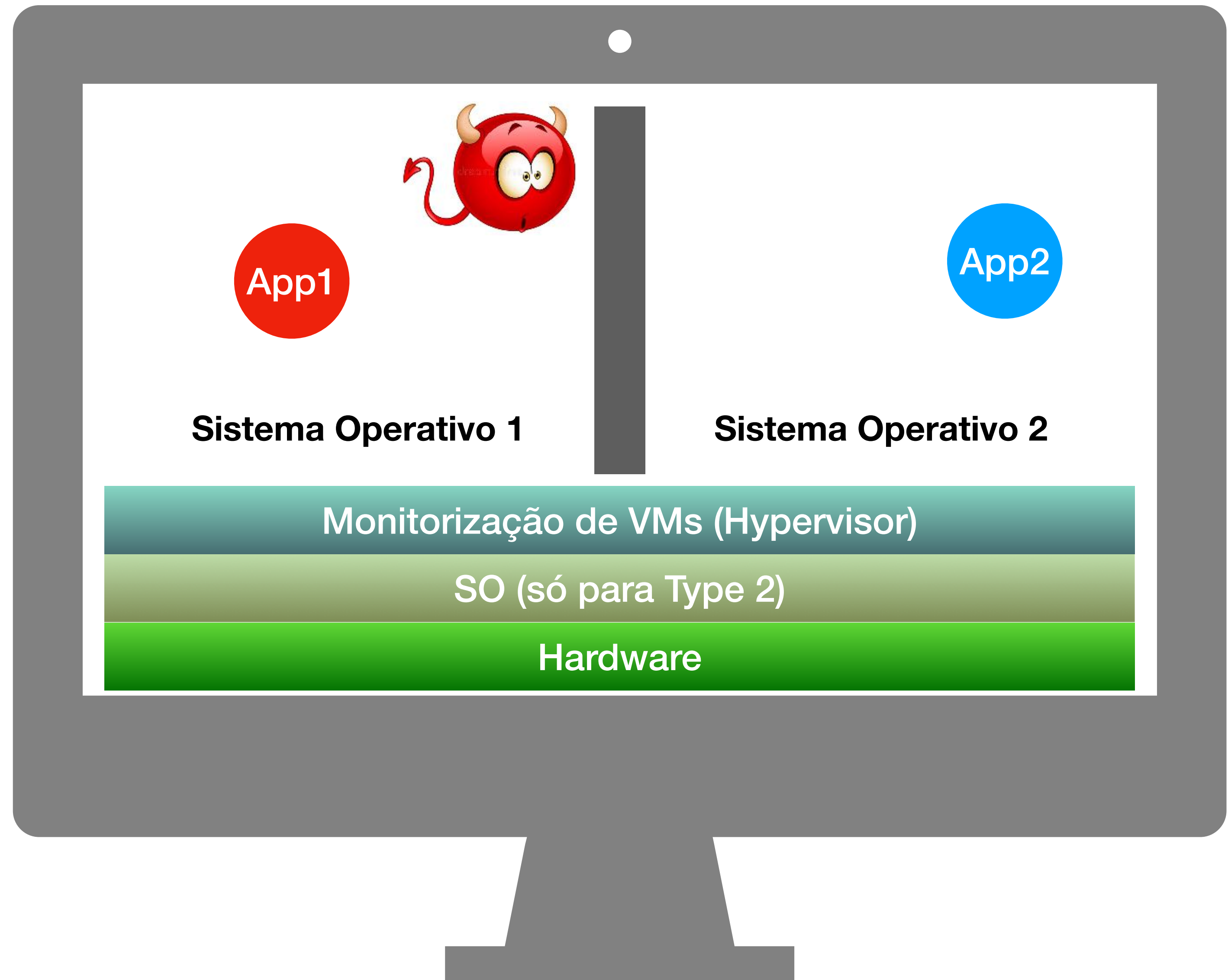


Confinamento: SFI + SCI

- **Software Fault Isolation (SFI):** nome genérico para isolamento de processos que partilham o mesmo espaço de endereçamento
- **System Call Interposition (SCI):** nome genérico para mediação de todas as system calls, concentrando os pontos de acesso a operações privilegiadas num número pequeno de pontos que podem ser monitorizados
- Nos sistemas *n?x vimos dois mecanismos da família SFI/SCI:
 - **Isolamento de Memória:** virtualizar o espaço de endereçamento e monitorizar os acessos nos mecanismos de tradução de endereços
 - **Separação Kernel vs Userland:** fornecer um número limitado de system calls, e mapeando sub-sistemas nos mecanismos de manipulação de ficheiros

Confinamento: Máquinas Virtuais

- O Hypervisor permite partilhar HW:
 - oferece visão virtual de HW a cada SO
 - garante que as ações em SO1 não afetam o contexto de SO2 e vice-versa
- Dois tipos principais de Hypervisor:
 - **“Type 1”** ou “bare metal”:
SO fino sobre HW (e.g., clouds)
 - **“Type 2”** ou “hosted”:
camada de software sobre o SO (e.g., computadores pessoais)



Máquinas Virtuais (História)

- **VMs 1960's**
 - poucos computadores, muitos utilizadores
 - VMs permitem a utilizadores partilhar um único computador
- **VMs 1970's – 2000 (não-existente)**
- **VMs > 2000**
 - muitos serviços, menos utilizadores
 - Print server, Mail server, Web server, File server, ...
 - VMs altamente utilizadas em computadores pessoais e clouds



Confinamento: Sandboxing

- Confinamento adicional dentro de uma aplicação
- Por exemplo os browsers são aplicações:
 - internamente criam um ambiente de execução isolado para código proveniente de fontes externas
 - interpretador de JavaScript/ WebAssembly, etc., com monitorização incorporada

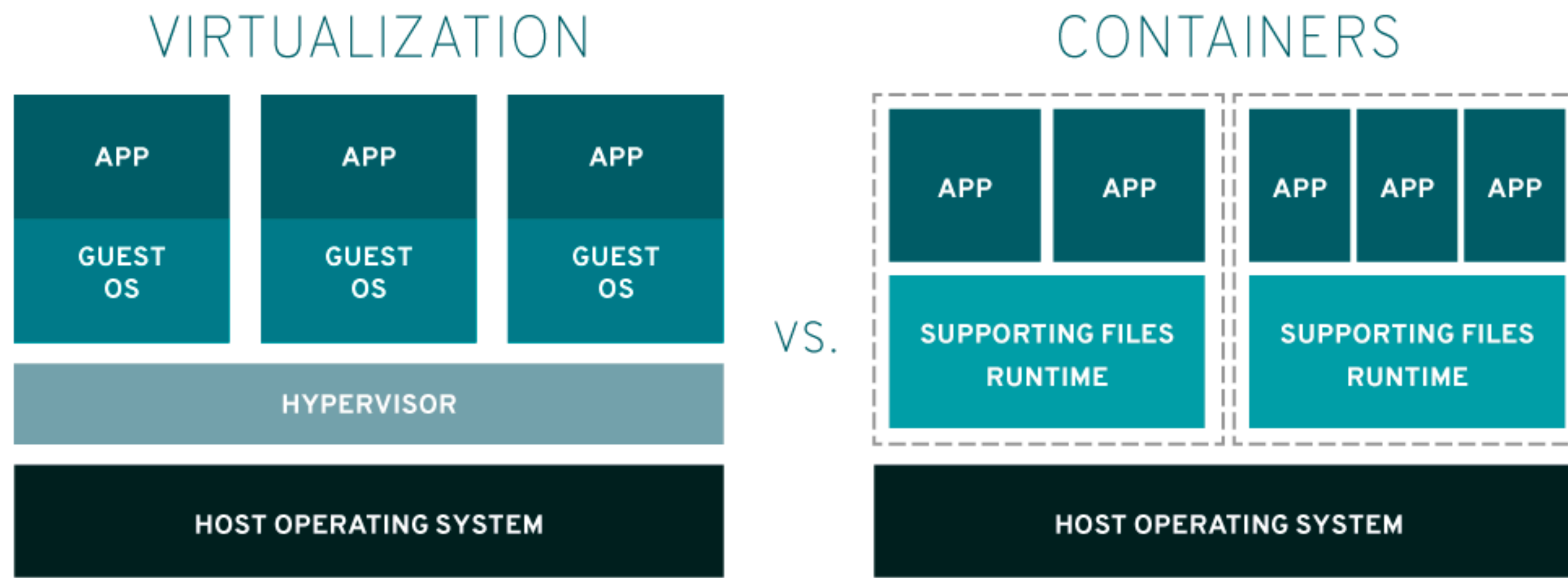


Confinamento: Implementação

- O componente central é chamado **reference monitor**
- Mesma lógica para SO, Hypervisor, sandboxes, ...
- Faz **mediação completa** 🗂️ de todos os pedidos de acesso a recursos
 - implementa uma política de proteção de recursos/isolamento
- Tem de ser sempre invocado \Rightarrow todas as aplicações são mediadas
- Tem de ser onnipresente
 - quando morre o reference monitor \Rightarrow morrem todos os processos
- Tem de ser simples o suficiente para poder ser analisado

Confinamento: containers

- **Máquinas Virtuais** correm SOs inteiros por cima do hardware (+ isolamento)
- **Containers** partilham o kernel do SO original e isolam os componentes userland (- recursos)



<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>

Exemplo Antigo: chroot

- O comando `chroot` permite criar *jails*
 - Primeira versão em 1979
 - pode ser utilizado apenas por *root*
 - transforma o ambiente visto pelos utilizadores/processos na shell:
 - a pasta actual passa a ser a raiz do filesystem
 - *system calls* que acedem a ficheiros são interceptadas e as paths recebem um prefixo correspondente à pasta actual
 - consequência \Rightarrow as aplicações (e.g., servidor web) não conseguem aceder a ficheiros fora da pasta actual

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

A raiz do sistema de ficheiros passa a ser
`/tmp/guest`

O utilizador efectivo dentro da shell passa a ser `guest`

```
fopen("/etc/passwd", "r")
```

passa a

```
fopen("/tmp/guest/etc/passwd", "r")
```


Exemplo Antigo: chroot

- Geralmente queremos dar a um utilizador/aplicação dentro de uma *jail*:
 - um ambiente com acesso a utilitários tipo `ls`, `ps`, `vi`, etc.
 - o utilitário **jailkit** permite configurar um ambiente isolado com controlo sobre o tipo de tarefas a executar:
 - inicializar o ambiente a partir de uma configuração
 - verificar que uma configuração é "segura" (o que quer dizer?)
 - lançar uma *shell* que permite aceder aos recursos configurados
- Nota: uma jail simples de `chroot` não permite limitar o acesso à rede

Fugir de uma jail

- Inicialmente: paths relativos
 - `fopen(“../../etc/passwd”, “r”)`
 - permitia fazer: `fopen(“/tmp/guest/ ../ ../etc/passwd”, “r”)`
- Um utilizador (não *root*) que consiga executar `chroot` consegue criar o seu próprio `passwd` e tornar-se *root* 😈 ⇒ vulnerabilidade em Ultrix 4.0
 - criar ficheiro `/aaa/etc/passwd`
 - executar `chroot /aaa`
 - fazendo `su root`, qual será a password pedida?

Fugir de uma jail

- É **crítico** que o **utilizador dentro de uma jail não consiga torna-se root**
- Caso contrário, existem muitas formas de escapar, nomeadamente:
 - criar um dispositivo para aceder ao disco em bruto
 - enviar sinais a processos que não estão dentro da jail
 - re-iniciar o sistema
 - etc.

Jails actuais: FreeBSD jails

- Conceito mais elaborado do que o chroot original
- Lançado com o FreeBSD 4.0 (2000)
 - Restringe ligações de rede e comunicação com outros processos
 - Restringe os privilégios de *root* dentro da *jail*
 - Objetivo inicial = confinamento, evolução \Rightarrow quasi-virtualização
- No entanto, permanecem limitações:
 - as políticas são pouco flexíveis (e.g., browser precisa de ler disco para enviar attachments no gmail)
 - as aplicações ainda estão em “contacto direto” com a rede e com o kernel



Jails actuais: Linux containers

- LXC: conceito bastante similar a FreeBSD jails
- Adaptado para Linux, ligeiramente mais recente (2008)
- Unprivileged containers: root no container é user normal fora
- Kernel Control Groups (cgroups): limita recursos (CPU, memória, rede, etc) para organizações hierárquicas de processos
- Kernel Namespaces: particionar recursos do kernel de forma a que diferentes processos vejam conjuntos de recursos diferentes



Acknowledgements

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