

Fundamentos de Segurança Informática (FSI)

2021/2022 - LEIC

Manuel Barbosa
mbb@fc.up.pt

Aula 9

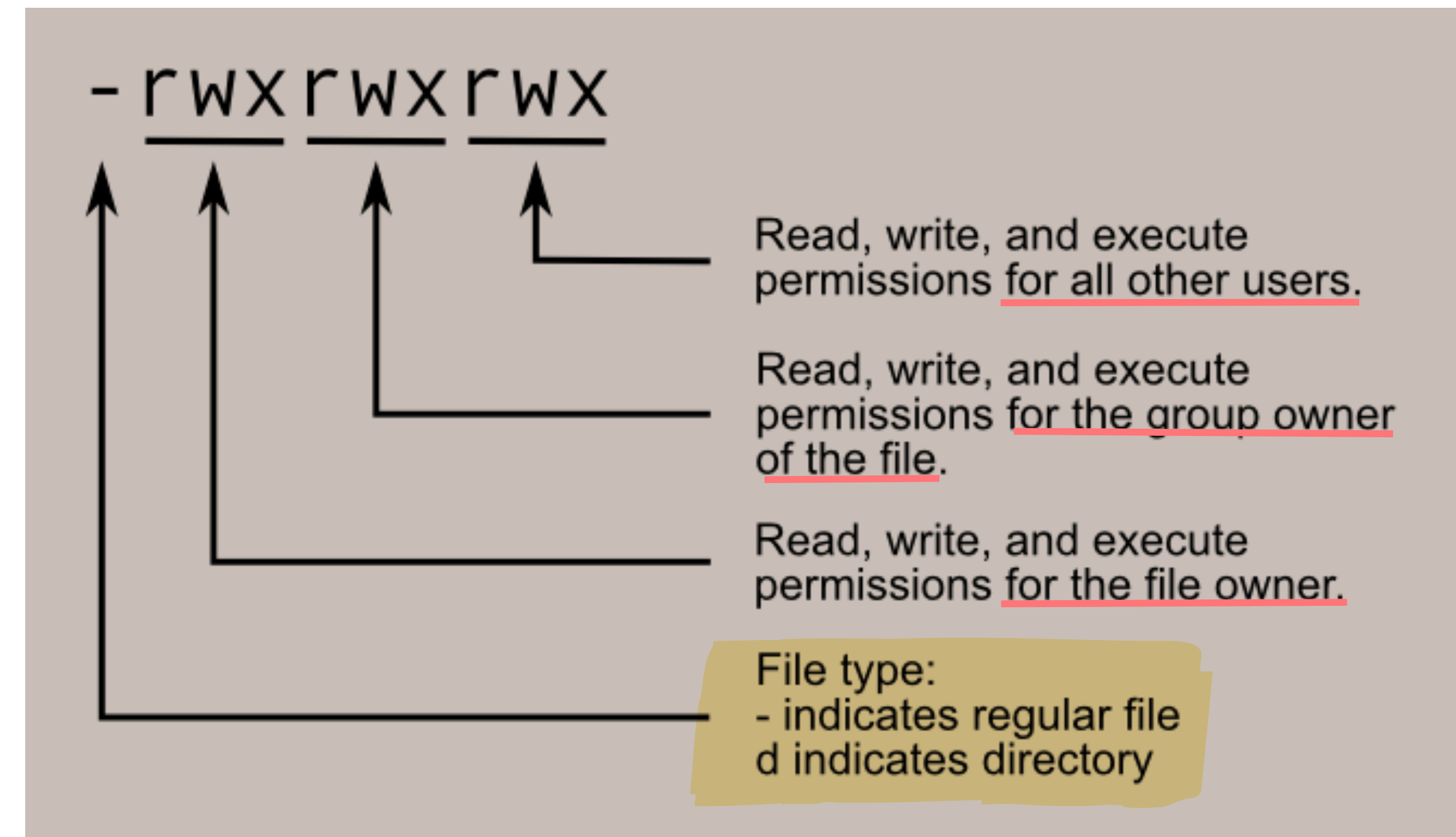
Segurança de Sistemas 3

Sistema de Ficheiros

- Veremos como exemplo os sistemas *nix:
 - 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*
↳ em nome de ...
 - uid = 0 utilizado para identificar esse utilizador/papel
 - boas práticas: utilização mínima


↳ utilizar os privilégios que precisamos e não todos
↳ privilégio mínimo



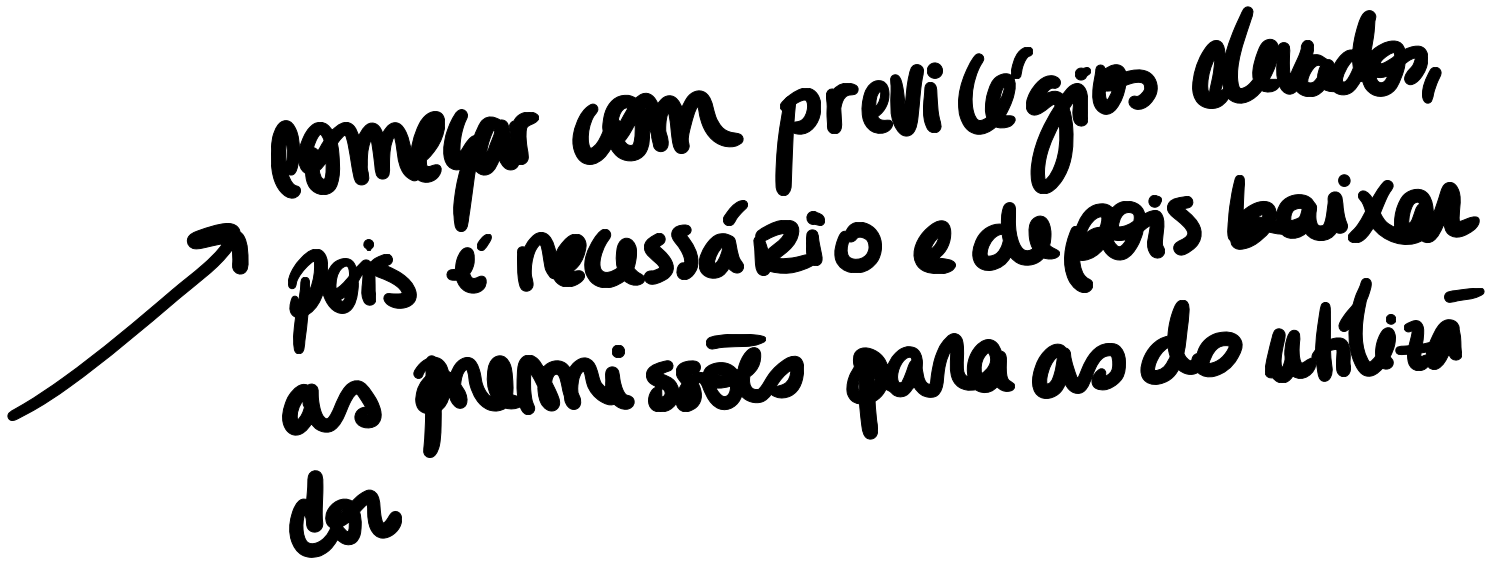
Sistema de Ficheiros

- Alteração de permissões:
 - sempre permitido ao superuser
 - permissões podem ser alteradas pelo owner (chmod)
 - owner pode ser alterado pelo superuser (chown)
 - grupo poder ser alterado por owner e superuser (chgrp)
- owner altera permissões => Discretionary Access Control → *quem é dono de um recurso, pode alterar as permissões desse recurso*
- Mandatory Access Control => apenas administrador (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., 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 
- O reverso (elevate privileges) deve ser impossível (e o passwd?)

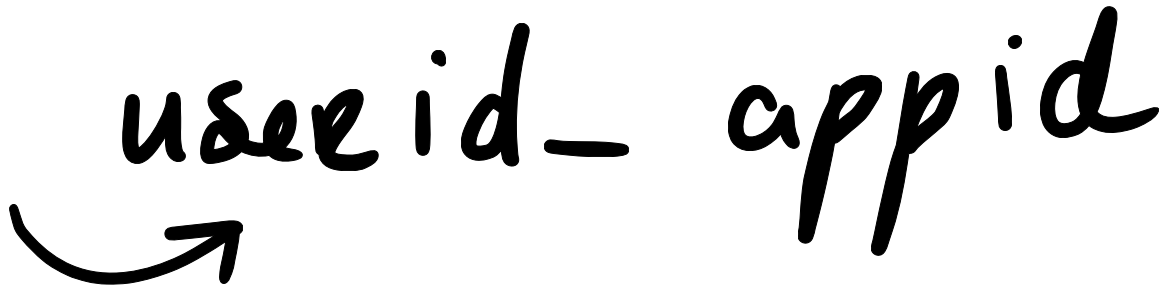
Sistema de Ficheiros

- 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
 - permite elevação de privilégios!
- No caso do `passwd` o *owner* é o utilizador *root*.

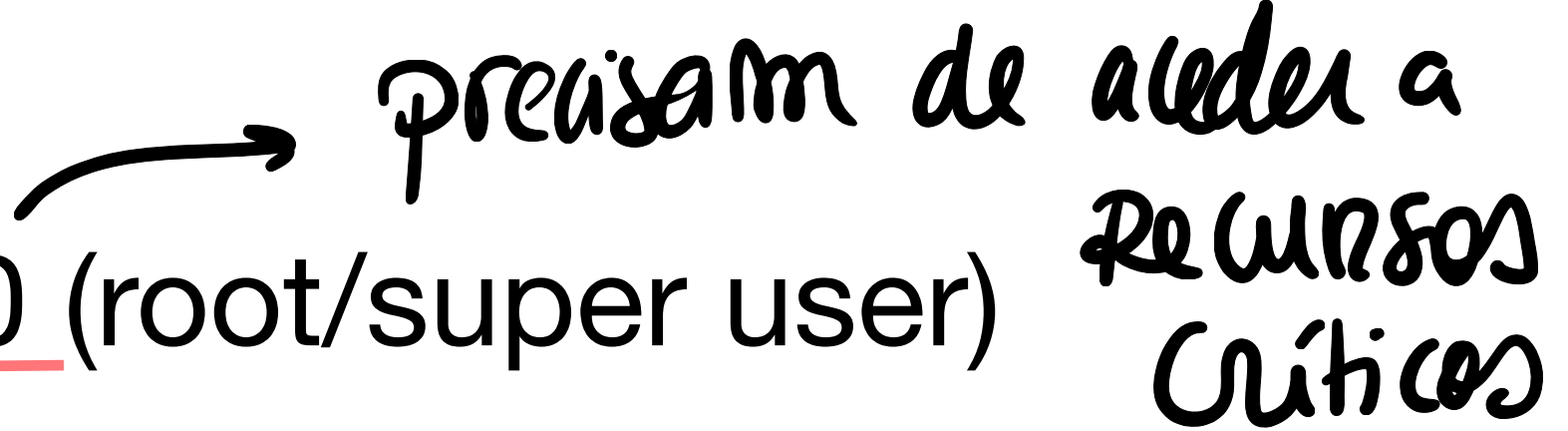
Sistema de Ficheiros

- Tudo é um ficheiro:
 - como minimizar o número de system calls/superfície de ataque?
 - utilizar a mesma interface construída para o sistema de ficheiros para outros recursos
 - Em *nix: 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
- Problema:
 - restringir o acesso de aplicações a recursos
 - solução: cada aplicação tem o seu próprio utilizador
 - problema: múltiplos utilizadores?
 - solução ad-hoc: u1_a23 

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)  precisam de aceder a Recursos Críticos
- acesso a todos os recursos => 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

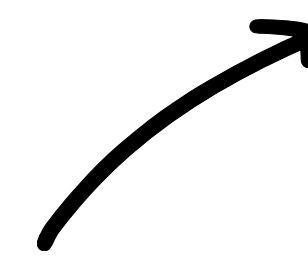
Privilégios de Processos

- 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:
 - $EUID \Rightarrow x$, $RUID \Rightarrow x$, $SUID \Rightarrow x$
- Utilizadores comuns apenas podem mudar EUID para RUID ou SUID
≠ Root
- 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
 - ↳ *recurso privilegiado*
 - ↳ *garantir que o processo tem apenas os privilégios que precisa*

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 ==> executar código ==> restaurar privilégios
- Perigo: usar `seteuid` quando root pretende baixar privilégios permanentemente (porquê?)
==> é possível voltar para RUID usando `setuid`!

RUID
SUID
permanece nos
anteriores



Privilégios de Processos

- Complexidade:

- mesmo com um sistema tão simples

Pouca coisa que podemos efetivamente mudar

- existe um sistema de transições entre estados de confiança
 - onde é muito fácil cometer erros

+ Convém sempre documentar as operações de administração, para que, se as coisas correrem mal, saibamos o que se passou

Conclusão

- O sistema de controlo de acessos em *nix é essencialmente uma implementação de Access Control Lists, com algum *batching*

- Vantagem => simples e funciona na prática

- Desvantagem => pouco robusto e pouco flexível

▶ Alguém que consiga mudar privilégios para root, toma o controlo de todo o sistema

- uma falha num processo tipo *passwd* ou *ssh* (*eu id* = 0) tem consequências catastróficas

- *root* utilizado para muita coisa => erros de administração
↳ tudo ou nada

Confinamento (prelúdio)

Executar Código Não Confiável

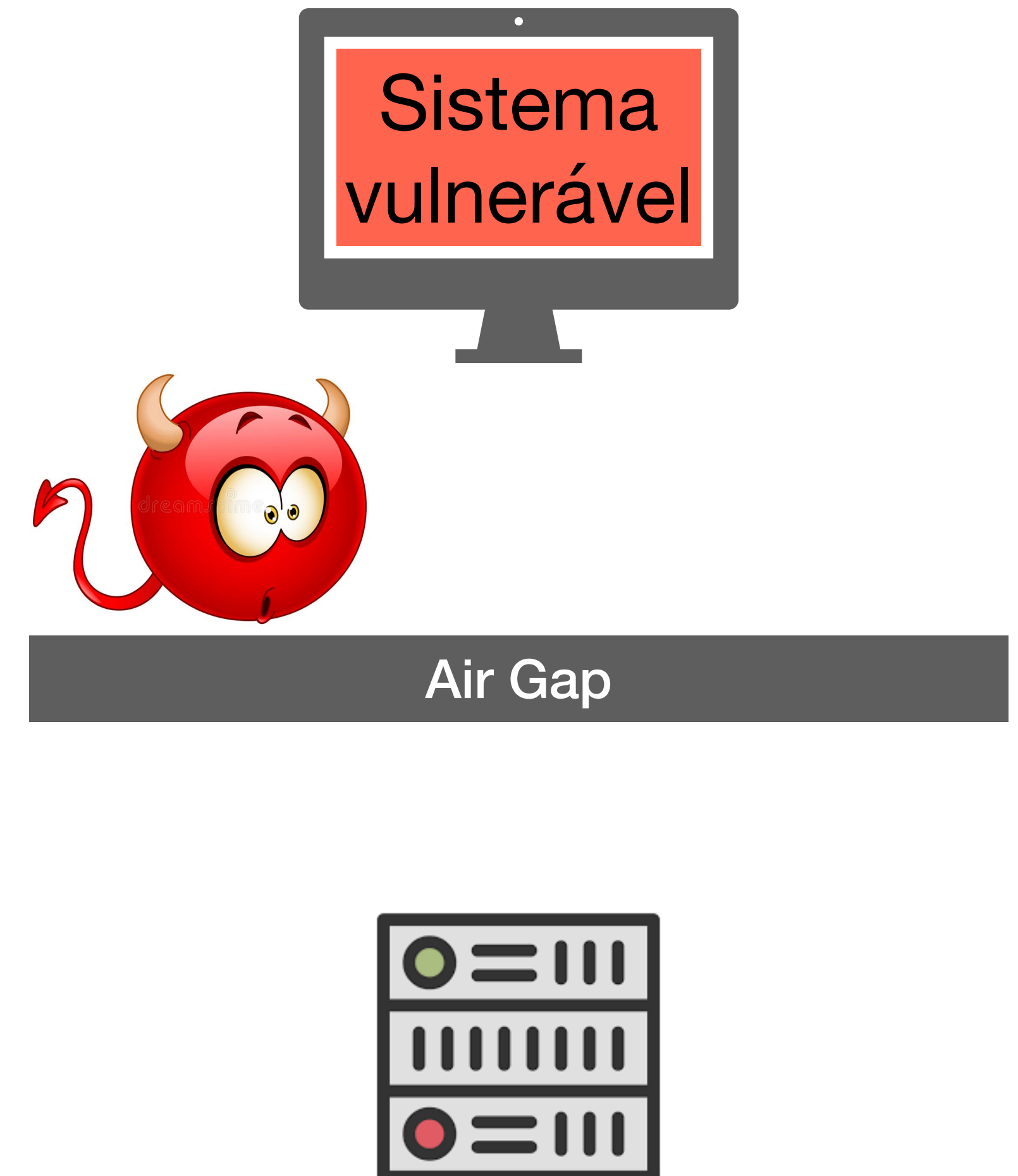
- É comum ser necessário executar código não confiável numa plataforma confiável:
 - código proveniente de fontes externas, nomeadamente sites Internet:
 - Javascript, extensões de browsers, mas também aplicações
 - código legacy que sabemos não estar à altura das exigências atuais
 - ↳ *qualquer código*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

honeypot → sistema vulnerável criado de propósito para estar junto dos outros, para garantir a deteção de ataque (pois é 'apetecível' devido "isca")

Confinamento: Air Gap

→ Não haver ligação física entre a máquina e outros componentes eletrônicos

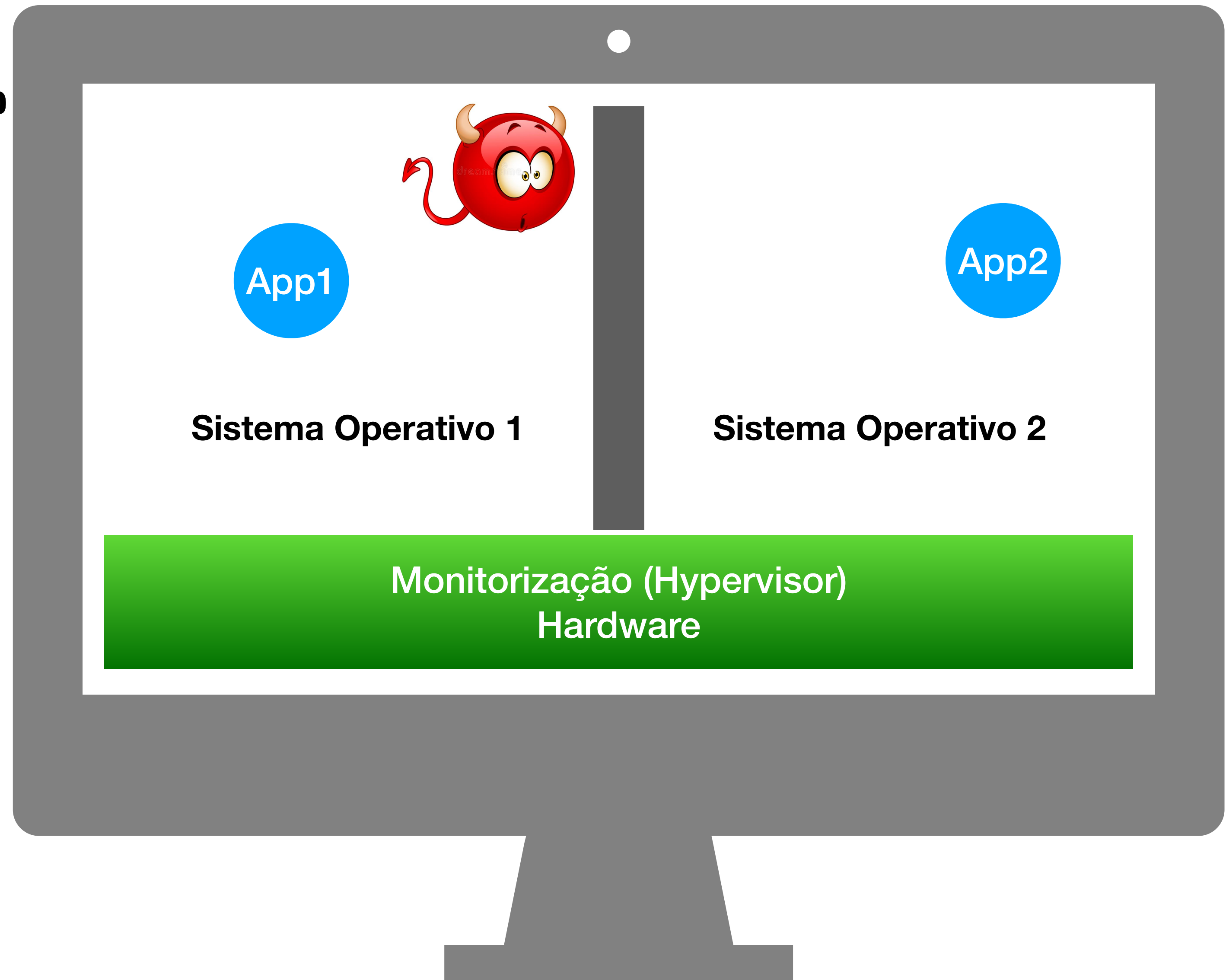
- Solução: garantir que o código potencialmente malicioso não pode afetar o resto do sistema
- Pode ser implementado a muitos níveis, começando no próprio hardware
- Quando o confinamento é efetuado ao nível do HW => *airgap*
- Desvantagem: difícil de gerir



Confinamento: Máquinas Virtuais

em cima de um SO coloca-se um hypervisor que tem como objetivo emular outros sistemas operacionais. No mesmo hardware → diferentes SO

- Os hypervisors permitem partilhar HW:
 - oferecem visão virtual de HW a cada SO
 - garantem que as ações em SO1 não afetam o contexto de SO2 e vice-versa



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 *nix 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: Sandboxing

- Confinamento 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



Browser é como se fosse uma máquina virtual para essas linguagens

Confinamento: Implementação

- O componente central é chamado *reference monitor*
- Faz mediação de todos os pedidos de acesso a recursos
 - implementa uma política de proteção de recursos/isolamento
- Tem de ser sempre invocado => todas as aplicações são mediadas
- Tem de ser onnipresente
 - quando morre o reference monitor => morrem todos os processos
- Tem de ser simples o suficiente para poder ser analisado

Exemplo Antigo: chroot

- O comando `chroot` permite criar jails
 - 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 => 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?

Consegue
suar
de
root

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 jail

- Mais elaborado do que o `chroot` original
 - Restringe ligações de rede e comunicação com outros processos
 - Restringe os privilégios de *root* dentro da *jail*
 - Objetivo inicial = confinamento, evolução => 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