Fundamentos de Segurança Informática (FSI)

2021/2022 - LEIC

Manuel Barbosa
mbb@fc.up.pt

Aula 10
Segurança de Sistemas 4
Confinamento/Isolamento

Confinamento no SO: Evolução

- Vimos formas de confinamento de processos usando chroot, jail e freebsd jail
- Mesmo princípio: Kernel reconhece processos lançados por jail e impõe restrições
- Na última década o suporte no Kernel tem sido melhorado para permitir mais expressividade e implementar melhores formas de confinamento:
  - namespaces: cada processo vive num mundo em que se podem controlar os recursos visíveis
  - cgroups: permite controlar a “quantidade” recursos utilizados por um processo e a sua prioridade
- Tudo isto são mecanismos de monitorização/mediação: nesta aula veremos outros

Exemplo Chromium: Layered

Sandbox types summary

Name	Layer and process	Linux flavors where available	State
Setuid sandbox	Layer-1 in Zygote processes (renderers, PPAPI, NaCl, some utility processes)	Linux distributions and Chrome OS	Enabled by default (old kernels) and maintained
User namespaces sandbox	Modern alternative to the setuid sandbox. Layer-1 in Zygote processes (renderers, PPAPI, NaCl, some utility processes)	Linux distributions and Chrome OS (kernel >= 3.8)	Enabled by default (modern kernels) and actively developed
Seccomp-BPF	Layer-2 in some Zygote processes (renderers, PPAPI, NaCl), Layer-1 + Layer-2 in GPU process	Linux kernel >= 3.5, Chrome OS and Ubuntu	Enabled by default and actively developed
Seccomp-legacy	Layer-2 in renderers	All	Deprecated
SELinux	Layer-1 in Zygote processes (renderers, PPAPI)	SELinux distributions	Deprecated
Apparmor	Outer layer-1 in Zygote processes (renderers, PPAPI)	Not used	Deprecated

System Call Interposition

Racional

- A superfície de ataque está limitada a system calls:
  - através do sistema de ficheiros para alterar o sistema actual
  - através da rede para afetar o sistema local e sistemas remotos
- Solução: monitorizar system calls e bloquear as que não são autorizadas
- Opções de implementação:
  - dentro do kernel, com um mecanismo em kernel space (`seccomp` no linux)
  - fora do kernel, com um mecanismo em user space (aka, *program shepherding*)
  - esquemas híbridos (e.g., Systrace)

Primeiras soluções: ptrace

- Process tracing no linux pode ser feito através da system call `ptrace`:
  - permite a um processo monitor ligar-se a processo alvo (descendente)
  - é notificado quanto o processo alvo faz uma *system call*
  - o monitor pode terminar o alvo se a chamada não for autorizado
- E.g. o sistema janus permite usar este mecanismo para SCI de forma genérica

```
int main()
{   pid_t child;
    long orig_eax;
    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    }
    else {
        wait(NULL);
        orig_eax = ptrace(PTRACE_PEEKUSER,
                          child, 4 * ORIG_EAX,
                          NULL);
        printf("The child made a "
               "system call %ld\n", orig_eax);
        ptrace(PTRACE_CONT, child, NULL, NULL);
    }
    return 0;
}
```

https://www.linuxjournal.com/article/6100

Primeiras soluções: limitações

- A utilização de ptrace é uma adaptação e tem complicações:
  - é ineficiente porque obriga a interceptar todas as chamadas
  - abortar uma system call => abortar processo
  - Ocorrem situações de race-conditions, e ataques TOCTOU:
    - TOC: tudo OK no *Time of Check*
    - processo altera estado do sistema
    - TOU: vulneravel em *Time of Use*

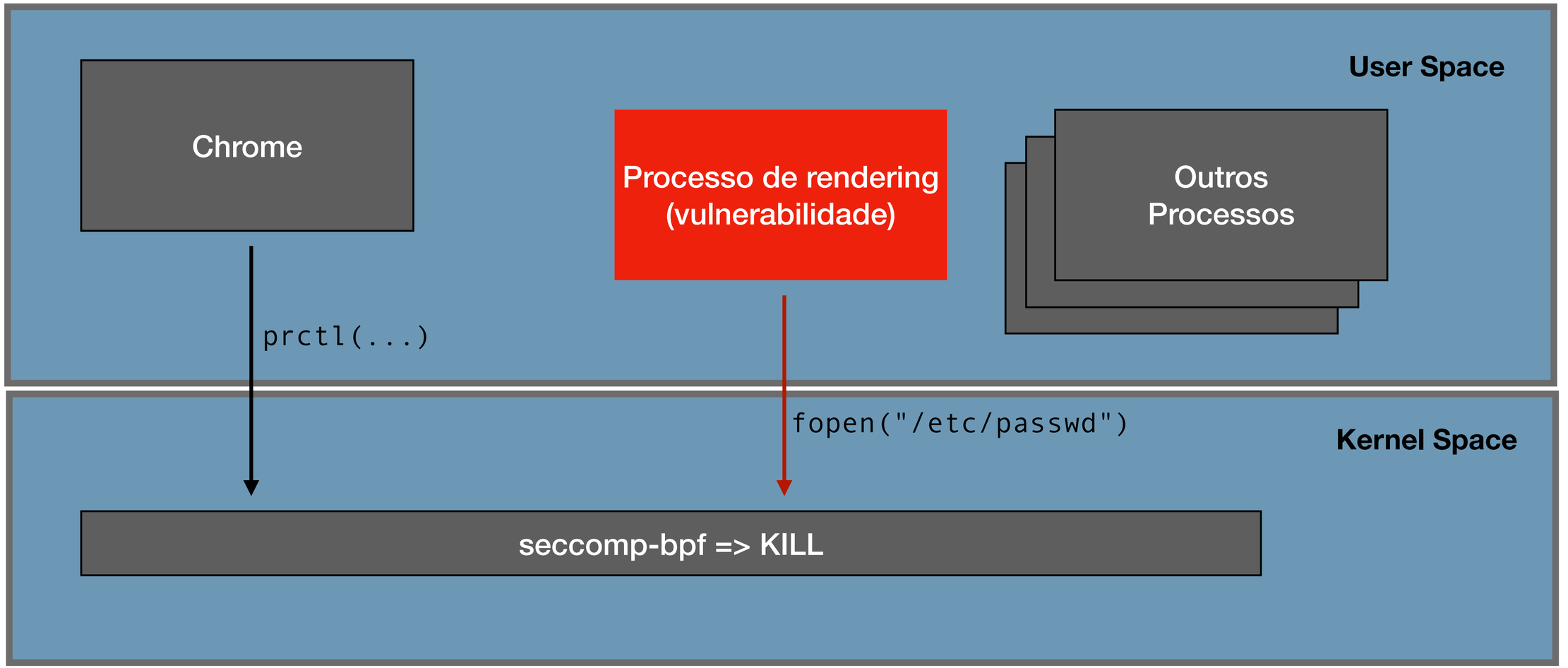
```
proc1: open("me")

O monitor verifica e autoriza

proc2: symlink me -> /etc/passwd

O sistema operativo executa
```

A monitorização e a execução ocorrem de forma não atómica.

Linux: seccomp+bpf

- seccomp = Secure Computing Mode:
  - Processo chama prctl() e entra em secure mode
  - Só pode terminar/retornar ou utilizar ficheiros já abertos
  - Uma violação leva o kernel a terminar o processo
- seccomp+bpf => é possível configurar de forma mais fina as system calls
  - Política configuráveis utilizando regras “Berkeley Packet Filter”
  - Muito utilizado: Chromium, Docker, etc.

Linux: seccomp+bpf



Linux: seccomp+bpf

- Um processo pode instalar múltiplos filtros BPF
  - depois de instalado não pode ser removido/desativado
  - propagado para todos os descendentes (incluindo `execve`)
- Parâmetros BPF: *system call*, argumentos, arquitectura
- Filtro executado em todas as system calls e retorna
  - kill OU rejeitar e retornar erro OU permitir

Exemplo: Docker

- É possivel isolar *containers* usando `seccomp-bpf`, como?
  - No modelo docker os *containers* são sub-processos do *docker-engine*
  - Pode controlar as chamadas feitas por esses sub-processos ao *kernel* usando `seccomp-bpf`
  - A política por omissão bloqueia muitas system calls, incluindo `ptrace`
  - A política pode ser alterada fornecendo um ficheiro na linha de comando:
    - `--security-opt seccomp=/path/to/seccomp/profile.json`

https://docs.docker.com/engine/security/seccomp/

Software Fault Isolation

Software Fault Isolation

- Objetivo: limitar a zona de memória acessível a uma aplicação
  - E.g., num contexto de sandboxing => Chrome
  - Pretende-se um mínimo de overhead
    - e.g., atribui-se segmento de memória
    - usam-se operações bitwise para verificar que acesso na gama correta
  - É possível controlar o código a executar
    - Operações "perigosas" precedidas de guardas
    - Compilado na hora (JIT) ou verificado quando carregado

Software Fault Isolation

- O que são operações perigosas?
  - claramente load/store de memória
    - antes de executar acesso adicionar guarda
      - verifica segmento ou força endereço dentro do segmento
  - mas também todos os saltos, porquê?
    - um salto pode ser utilizado para executar código externo sem guardas
    - é necessário validar os endereços de destino com mecanismos semelhantes
  - a implementação pode ser difícil dependendo da arquitectura
    - por exemplo: difícil diretamente sobre assembly x86 (registos, muitas instruções de risco, etc.)

https://dl.acm.org/doi/10.1145/168619.168635

SFI essencial em sandboxing

how webassembly provides software fault isolation

> At the specification level, "linear memory" is the only space that WebAssembly programs can access with its load and store instructions. Ensuring that this is true is the job of the WebAssembly VM, which is free to do it in any way so long as it meets the specification. In the VM I work on, Wasmer, the pointers into linear memory are 32-bit offsets from a base pointer and by default we allocate 6GiB of virtual addresses so that all possible pointer accesses fall inside the linear memory (from -2GiB to +4GiB, hence 6GiB). Some of these addresses are mapped inaccessible so that accesses to them cause a trap as required by the Wasm spec. There's no need to implement linear memory this way, you could write a WebAssembly VM that uses a hashtable for linear memory accesses (key=address, value=byte) and as long as it's implemented correctly no program should be able to tell.

https://stackoverflow.com/questions/66776662/how-webassembly-provides-software-fault-isolation

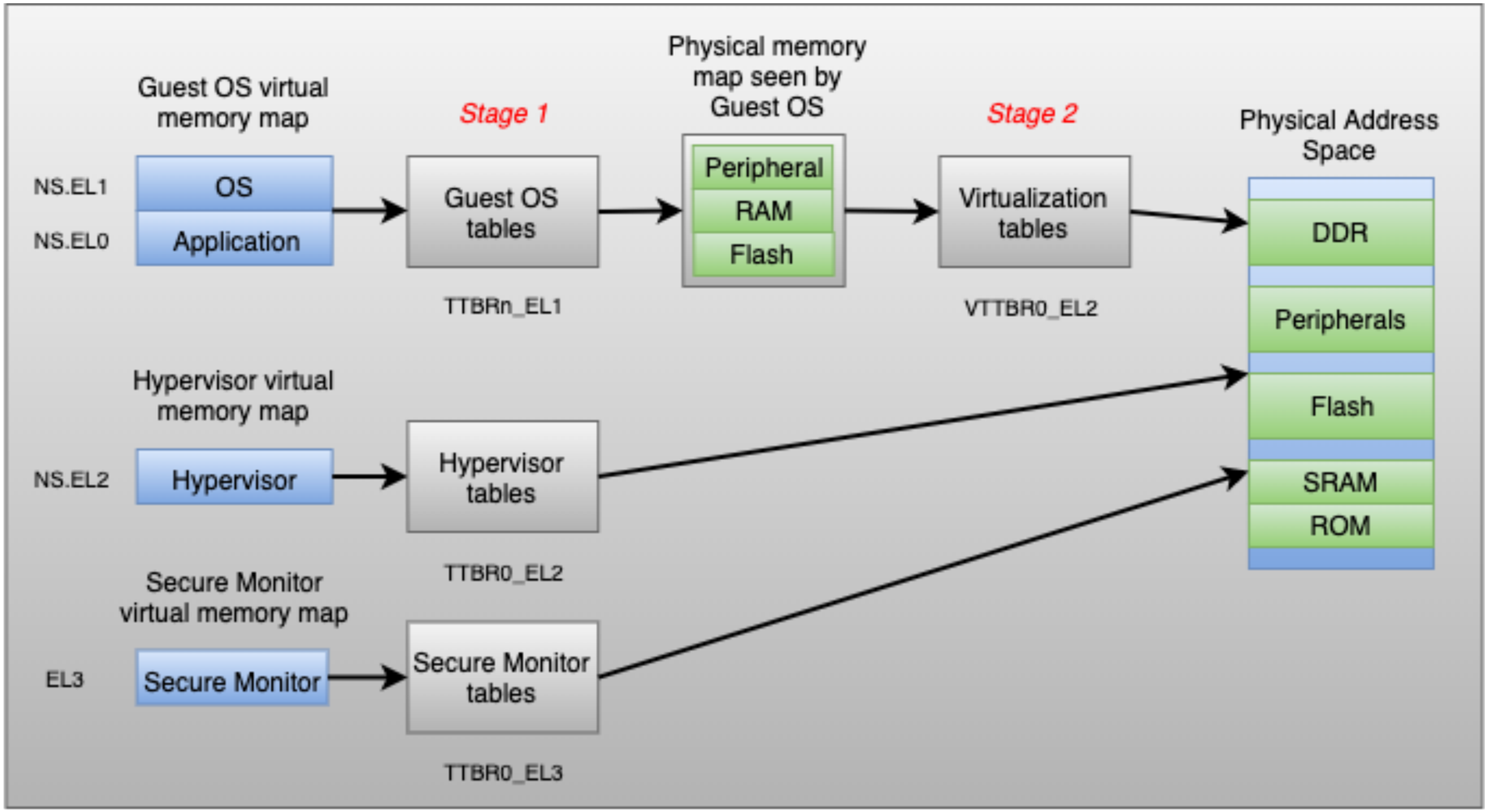
Máquinas Virtuais

Máquinas Virtuais

- Nos primórdios da computação:
  - HW escasso e muito caro => natural mesmo HW ser utilizado para diversos fins
- Nos anos 70-90 => computador pessoal!
- Século XXI => dispositivos pessoais cada vez mais limitados a interfaces
  - o poder computacional está do lado dos providers (na cloud)
  - gestão racional => aproveitamento flexível do HW => co-localização
- A virtualização é uma tecnologia fundamental hoje em dia
  - **necessário garantir isolamento pela natureza do serviço!**

Máquinas Virtuais

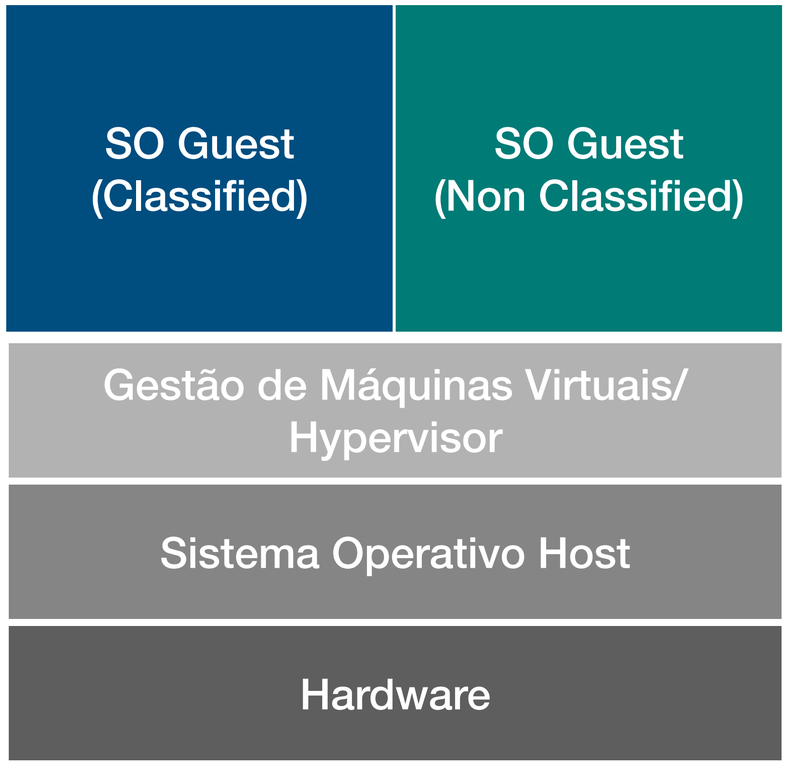
- São um mecanismo muito útil para isolamento em cenários práticos:
  - provável que um OS seja vulnerável/infetado/comprometido
  - possível que um *hypervisor* seja eventualmente vulnerável
  - improvável que ambos sejam vulneráveis simultaneamente
- Um *hypervisor* é (geralmente) mais simples do que um OS
  - facilita a validação => mas não existe risco zero
  - HW preparado para virtualização => menor risco
    - e.g., memória virtual com diversos níveis de tradução de endereços



https://www.starlab.io/blog/deep-dive-mmu-virtualization-with-xen-on-arm

Máquinas Virtuais

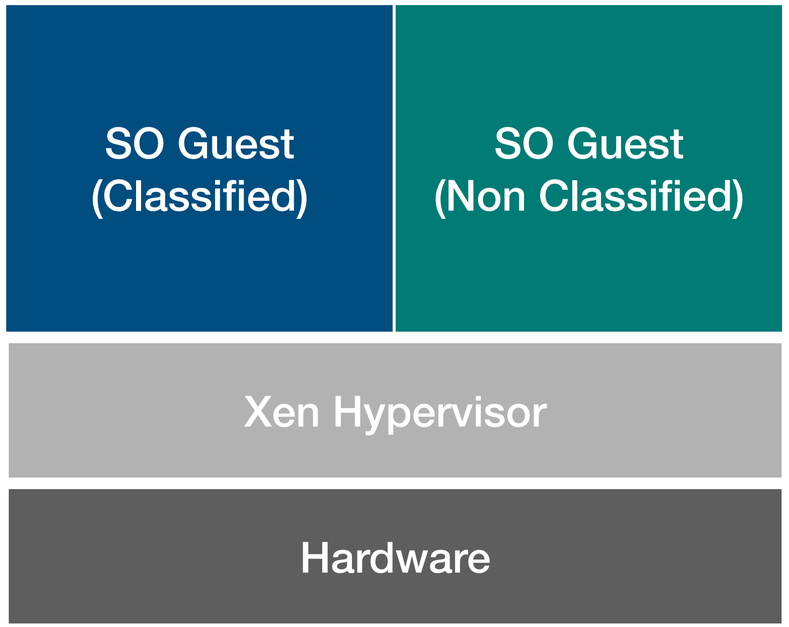
- Exemplo de aplicação prática
  - NSA NetTop => tratar dados "classified" e "non-classified" no mesmo HW
  - Solução => SOs e Virtualização comerciais para construir isolamento
  - Racional => para informação privilegiada sair do OS comprometido é necessário corromper diversos sub-sistemas
  - Problema: canais subliminares (mais à frente)



https://en.wikipedia.org/wiki/NetTop

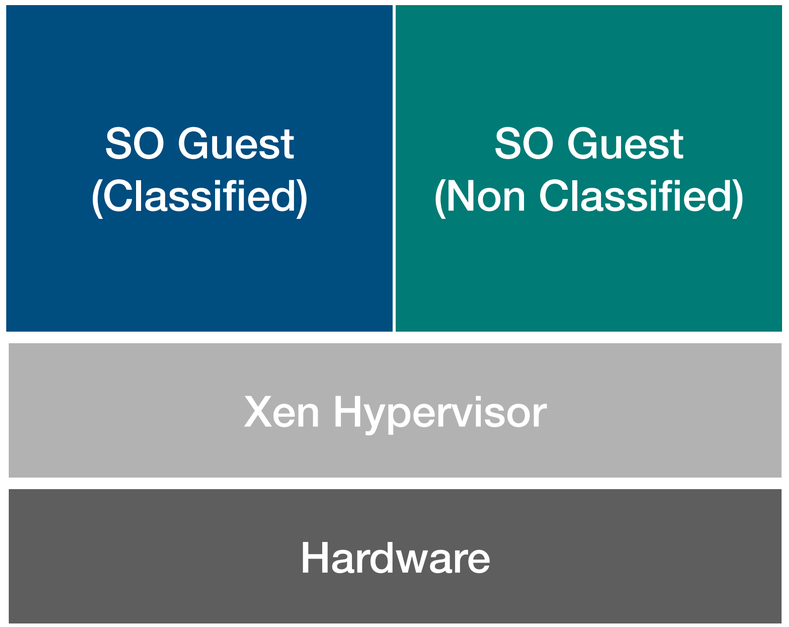
Máquinas Virtuais

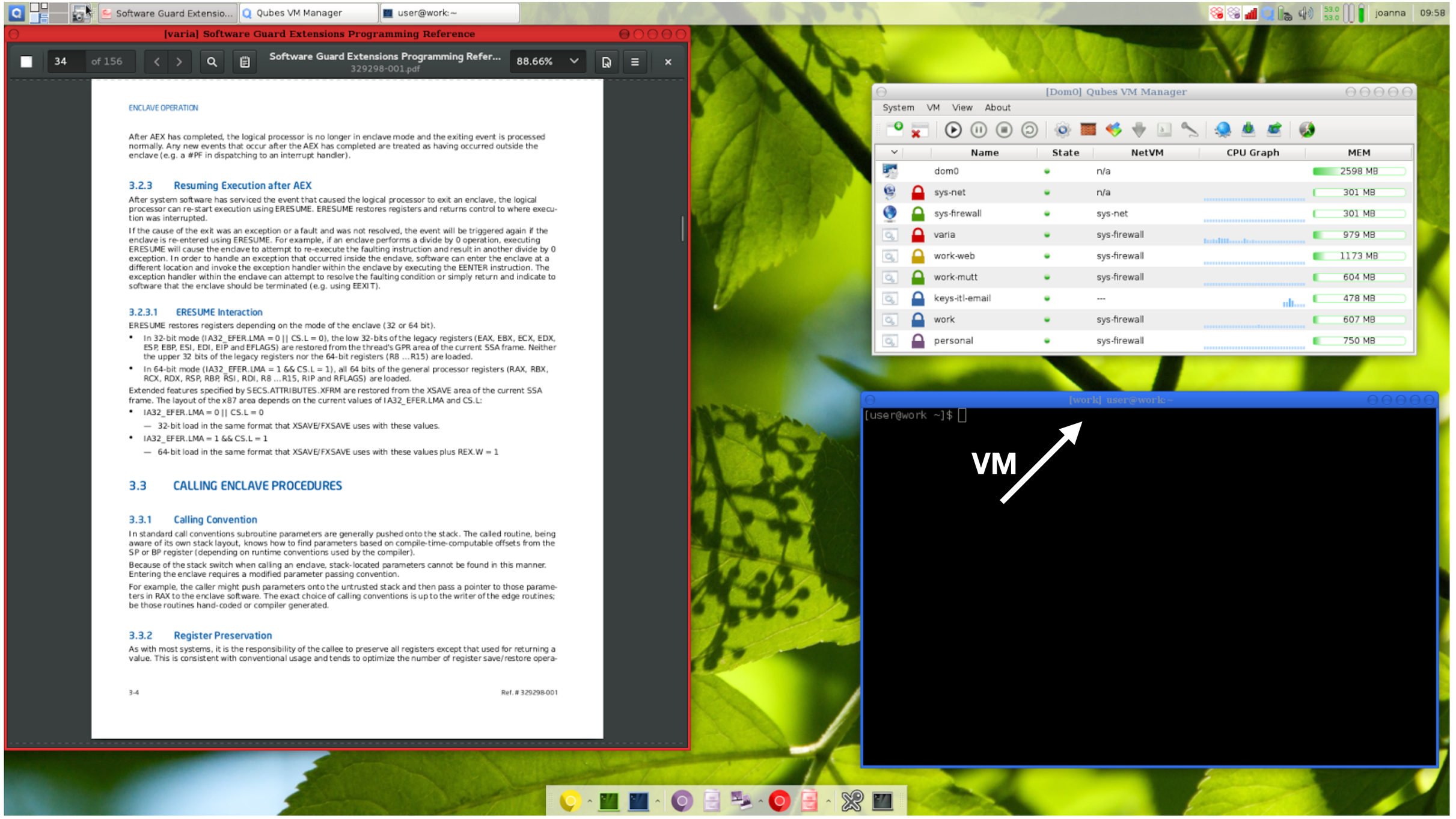
- Exemplo de aplicação prática
  - Providers cloud => eliminam SO host
  - Hypervisor gere diretamente o HW
  - Sistemas de diferentes clientes no mesmo HW!
  - Problema => o sistema de um cliente pode atacar o sistema de outro cliente co-localizado na mesma máquina?



Máquinas Virtuais

- Exemplo de aplicação prática
  - Qubes => SO orientado à virtualização
  - Utiliza também Xen Hypervisor
  - Exemplos de casos de uso:
    - VM descartável, e.g., documentos/SW duvidoso
    - VM para trabalho (Linux)
    - VM para conteúdos pessoais (Windows)





https://www.hacking.land/2017/12/qubes-os-security-oriented-operating.html?m=1

Quebras de Isolamento

- Canais subliminares:
  - observação de efeitos colaterais (e.g., atrasos no acesso a memória)
  - e.g.: um processo transmite informação a outro criando efeitos observáveis na *cache*
- Recentemente ficou claro que os processadores actuais não são ideais:
  - existem muitos mecanismos no HW partilhado que permitem fugas de informação e quebra de isolamento
  - casos célebres: Meltdown e Spectre exploram execução especulativa

https://meltdownattack.com

Red Pill vs Blue Pill

- Pergunta mais simples:
  - será que um programa sabe que está a correr numa máquina virtual?
- Porque é que esta pergunta é relevante?
  - Malware: geralmente analisado em ambientes virtuais => evita análise
  - Fabricante de SW exige correr em HW proprietário/HW key
  - O mesmo para conteúdos protegidos por *copyright*

Red Pill vs Blue Pill

- Pergunta mais simples:
  - será que um programa sabe que está a correr numa máquina virtual?
- Existem várias formas de detetar ambiente virtualizado
  - instruções disponíveis, features de HW específicas, etc.
  - latência no acesso a memória (*profiling*)
  - o hypervisor utiliza parte dos mecanismos de gestão do HW
    - um guest OS malicioso pode detetar a limitação de recursos

Red Pill vs Blue Pill

- Pergunta mais simples:
  - será que um programa sabe que está a correr numa máquina virtual?
- Exemplo prático:
  - como detetar *web sites* que injetam *malware*?
  - correr o browser numa VM e expo-lo sistematicamente a sites (*crawling*)
  - os sites evoluíram para detetar ambiente virtualizado!
    - exemplo: latência no rendering para o ecrã

https://bannedit.github.io/Virtual-Machine-Detection-In-The-Browser.html

Malware (prelúdio)

O que é Malware?

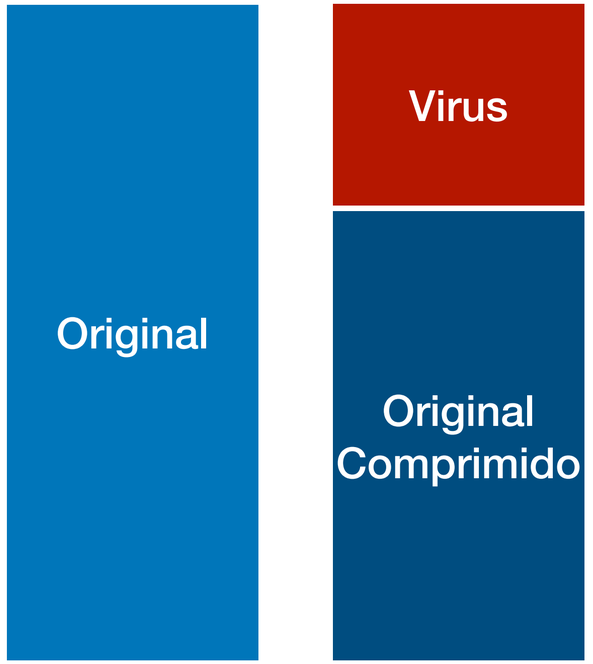
- Vimos nas aulas anteriores:
  - o estado de confiabilidade de um sistema pode ser comprometido
  - exploração de uma vulnerabilidade/execução de código malicioso
  - malware: a nossa máquina faz coisas do interesse de um atacante
- A terminologia usada é, por vezes, confusa mas, em geral classifica-se:
  - a forma como toma controlo e a utilização maliciosa

Alguma Terminologia

- Virus: código que se propaga criando um contexto em que, eventualmente, será executado
- Worm: código que se propaga autonomamente e consegue criar sozinho as condições para se executar e propagar
- (Nem sempre é tão claro)
- Rootkit: código desenhado para esconder a sua presença e permitir acesso com privilégios elevados a um atacante
- Trojan: código que aparenta ser legitimo, mas tem como objetivo (típicamente) transmitir informação para um atacante

Exemplo: Virus

1. Utilizador executa um programa infetado
2. O código do vírus está armazenado no programa
3. O vírus é executado quando o programa é executado
    1. localiza outro programa para infetar
    2. copia-se para o código desse outro programa (dissimulação cada vez mais elaborada)
    3. se lógica ativada: executa tarefa maliciosa
    4. no final pode desaparecer (apagar-se)



Que tipo de tarefa maliciosa?

- O malware executa com os privilégios do utilizador ativo no momento da tomada de controlo:
  - pode fazer o que o utilizador faz
  - explorando outras vulnerabilidades => pode escalar privilégios
- Objetivos:
  - brincadeira ou causar danos: pop-ups, apagar ficheiros, danificar HW
  - vigilância/espionagem: key logging, captura ecran, audio, câmara

Que tipo de tarefa maliciosa?

- Vantagens económicas/ciber-crime
  - botnet: rede de máquinas controladas por um atacante
    - DoS e DDoS
    - spam: vender produtos, pedir dinheiro, (spear)phishing, etc.
    - click-fraud: obter vantagens ou prejudicar terceiros (e.g., esgotar créditos)
    - lançar malware em grande escala
  - extorsão: ransomware, roubo de credenciais, chantagem