Fundamentos de Segurança Informática (FSI)

2024/2025 - LEIC

Systems Security (Part 4)

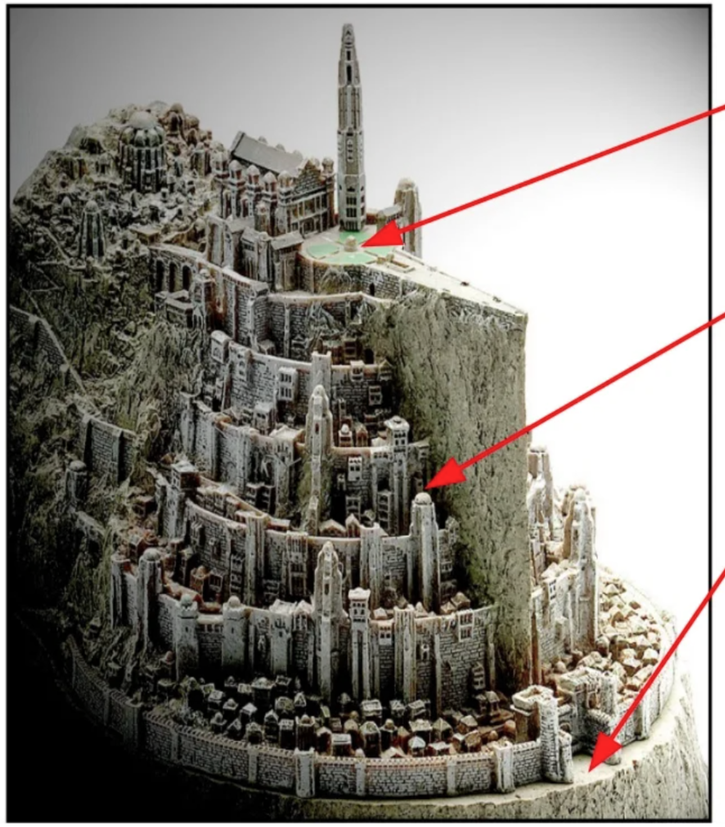
Hugo Pacheco
hpacheco@fc.up.pt

Confinement

Confinement in the OS

- We have seen confinement of processes using jails (lightweight containers)
- OS support: kernel recognises processes inside a jail and imposes further restrictions
- Last decade: **kernel support has improved to support better and more expressive forms of confinement:**
  - namespaces: availability of resources
  - cgroups: amount and priority of resources
  - limit root in the container
- These are all **mediation mechanisms**

  ⇒ today we will see a few more



[image: Minas Tirith from LOTR]

Brief history of containers

2000

FreeBSD Jails expand on Unix chroot to isolate files

Jails

2001

VServer

Linux-VServer ports kernel isolation, but requires recompilation

2004

Solaris Zones bring the concept of snapshots

SOLARIS

Zones

2006

cgroups

Google

Google introduces Process Containers, merged as cgroups

2008

Red Hat adds user namespaces, limiting root access in containers

redhat

Namespaces

2008

LXC

IBM

IBM creates LXC, providing user tools for cgroups and namespaces

2013

Docker provides simple user tools and images. Containers go mainstream

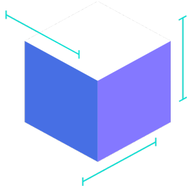
docker

Docker

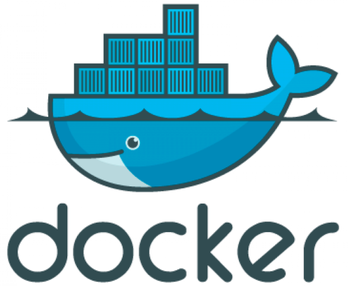
https://kccncna17.sched.com/event/CU6N/container-runtime-and-image-format-standards-what-it-means-to-be-oci-certified-i-jeff-borek-ibm-stephen-walli-microsoft

OPEN CONTAINER INITIATIVE

- A single open specification



Confinement: Docker



- Focus in **single applications in isolated and reproducible environments**
  - Docker Engine: create, run and distribute containers
  - Docker Hub: share containers
- Built on top of LXC (< 1.0, `libcontainers` replaced LXC ≥ 1.0)
  - More portable: abstractions for container configuration (network, storage, logging, etc)
  - Creation of containers in incremental layers (similar to `git` history)
  - Composition of containers (`docker-compose`) (used in the SEED LABS)
- The `docker` daemon (which manages containers, virtual disks, network, etc) runs as root ⇒ 

Shocker (2014)

- Vulnerability (CVE-2014-3519)
- Process associated to the container inadvertently has a kernel capability `CAP_DAC_READ_SEARCH`
  - Access to system call that can read files

`CAP_DAC_READ_SEARCH`
* Bypass file read permission checks and directory read and execute permission checks;
* **Invoke** `open_by_handle_at(2)`.

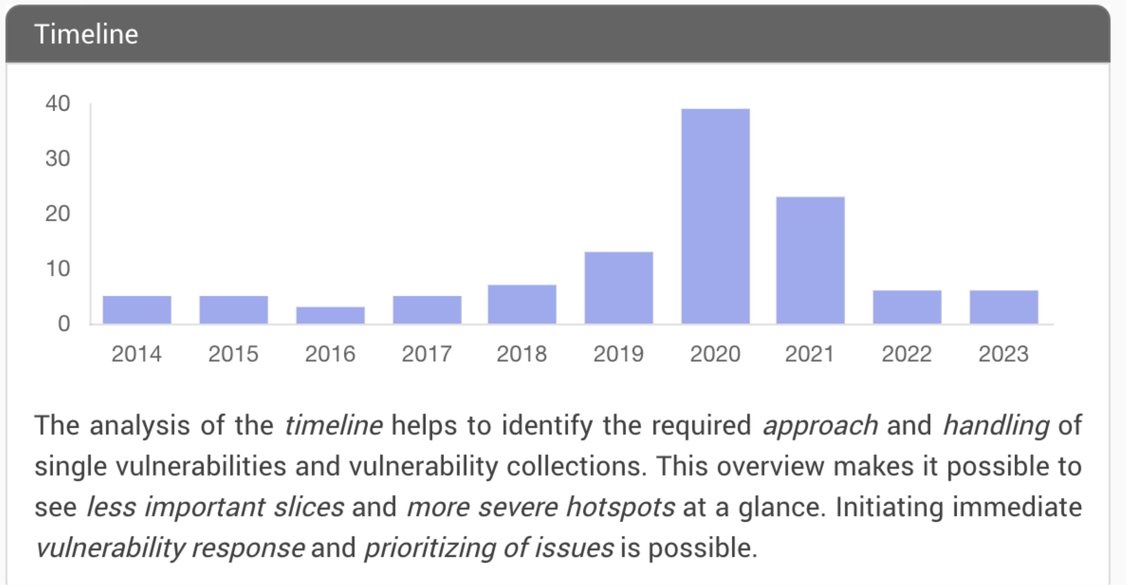
- Fixed in Docker 1.0

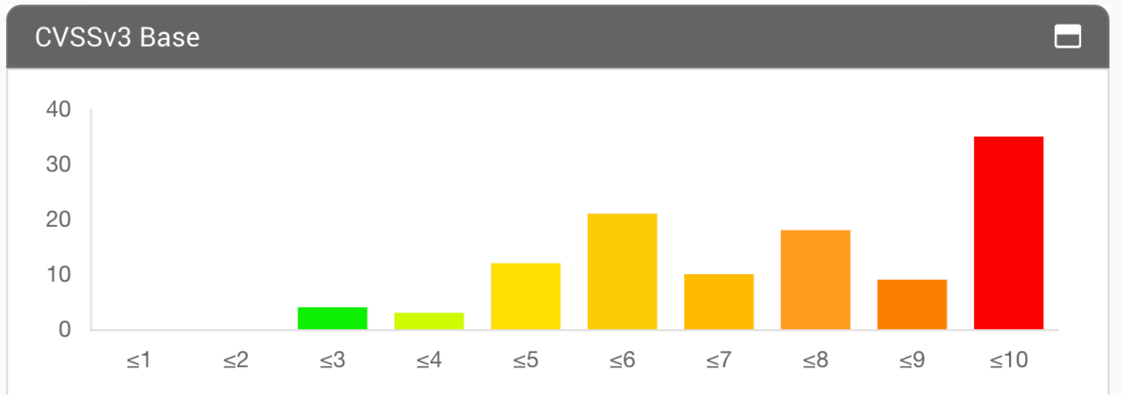
```
/* shocker: docker PoC VMM-container breakout (C) 2014 Sebastian Krahmer
 *
 * Demonstrates that any given docker image someone is asking
 * you to run in your docker setup can access ANY file on your host,
 * e.g. dumping hosts /etc/shadow or other sensitive info, compromising
 * security of the host and any other docker VM's on it.
 *
 * docker using container based VMM: Sebarate pid and net namespace,
 * stripped caps and RO bind mounts into container's /. However
 * as its only a bind-mount the fs struct from the task is shared
 * with the host which allows to open files by file handles
 * (open_by_handle_at()). As we thankfully have dac_override and
 * dac_read_search we can do this. The handle is usually a 64bit
 * string with 32bit inodenumber inside (tested with ext4).
 * Inode of / is always 2, so we have a starting point to walk
 * the FS path and brute force the remaining 32bit until we find the
 * desired file (It's probably easier, depending on the fhandle export
 * function used for the FS in question: it could be a parent inode# or
 * the inode generation which can be obtained via an ioctl).
 * [In practise the remaining 32bit are all 0 :]
 *
 * tested with docker 0.11 busybox demo image on a 3.11 kernel:
 *
 * docker run -i busybox sh
 *
 * seems to run any program inside VMM with UID 0 (some caps stripped); if
 * user argument is given, the provided docker image still
 * could contain +s binaries, just as demo busybox image does.
 *
```

http://stealth.openwall.net/xSports/shocker.c

Confinement: Docker

- Many other CVEs:
  - Code injection
  - Symlinks
  - Kernel capabilities
  - Trojans
  - …
-  : Full root, resource consumption, …
- Always stay up-to-date!





https://vuldb.com/?product.docker

Confinement: Chromium

- Defense in depth 🕎: various platform-specific evolving security layers
- **Software Fault Isolation (Layer-1)** + **System Call Interposition (Layer-2)**

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

https://chromium.googlesource.com/chromium/src/+/lkgr/docs/linux/sandboxing.md

System Call Interposition

SCI: Rationale

- **The attack surface is limited to system calls (complete mediation 🕎):**
  - Through the file system to change the actual system
  - Through the network to affect the local system and remote systems
- **Solution: monitoring system calls and blocking the non-authorised ones**
- Implementation:
  - **Inside the kernel, with a kernel-space mechanism (`seccomp` in linux)**
  - outside the kernel, with a user-space mechanism (aka, *program shepherding*)
  - hybrid mechanisms (e.g., Systrace)

First solutions: ptrace

- In linux, process tracing can be performed via the `ptrace` system call:
  - Allows a monitor process to connect to a target process (descendent)
  - It is notified when the target process invokes a system call
  - the monitor can kill the target if call is not authorised
- E.g. the Janus system uses `ptrace` for enforcing general SCI policies

```
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; }
```

```
path allow /tmp/*
path deny /etc/passwd
network deny all
```

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

First solutions: limitations

- Adapting `ptrace` leads to **complications**:
  - Requires intercepting all calls ⇒ inefficient
  - Monitor has to be duplicated on forks
  - Aborting a system call ⇒ aborting the process
  - Vulnerable to TOCTOU attacks:
    - TOC: everything OK at *Time of Check*
    - other process changes the system's state
    - TOU: vulnerable at *Time of Use*

```
Process 1: open("me")
```

The monitor verifies and authorises

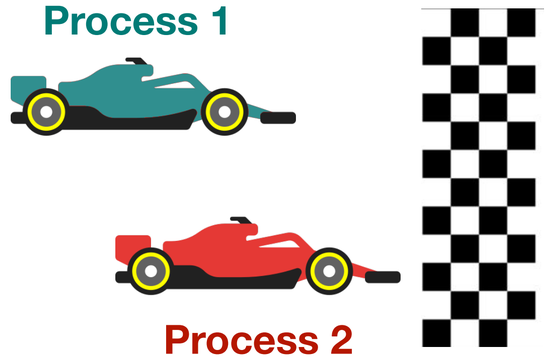
```
Process 2: symlink me -> /etc/passwd
```

The OS then executes

race conditions

⇑

monitoring and execution are not enforced atomically



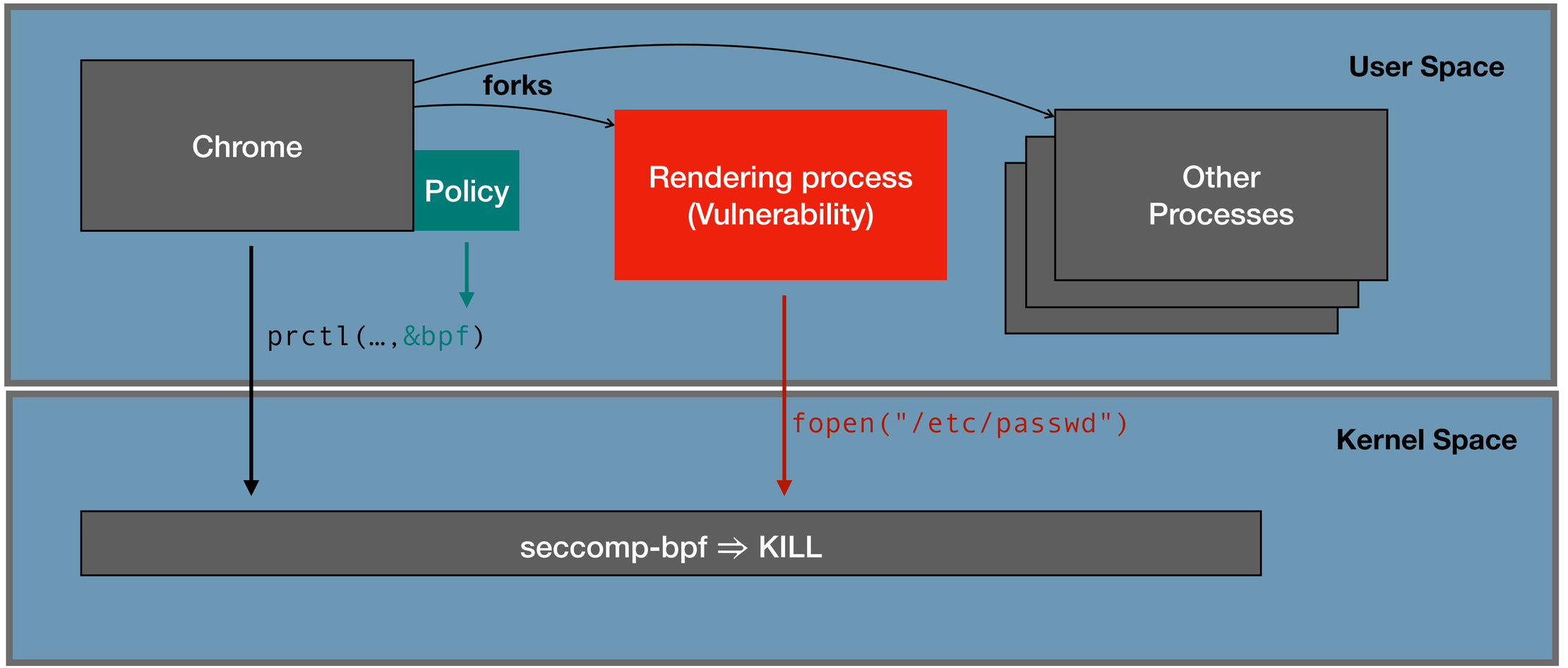
Linux Kernel: seccomp+bpf

- **seccomp = Secure Computing Mode:**
  - Process calls `prctl()` and enters secure mode
  - Cannot call any syscall, except to terminate/return or use already opened files
  - A violation leads the kernel to terminate the process
- **seccomp+bpf = more fine-grained configuration of system calls**
  - Configurable policies using “Berkeley Packet Filter” rules
  - Immune to TOCTOU attacks
  - Widely adopted: Chromium, Docker, etc.

https://book.hacktricks.xyz/linux-hardening/privilege-escalation/docker-security/seccomp

https://www.kernel.org/doc/html/v4.18/userspace-api/seccomp_filter.html

Linux Kernel: seccomp+bpf

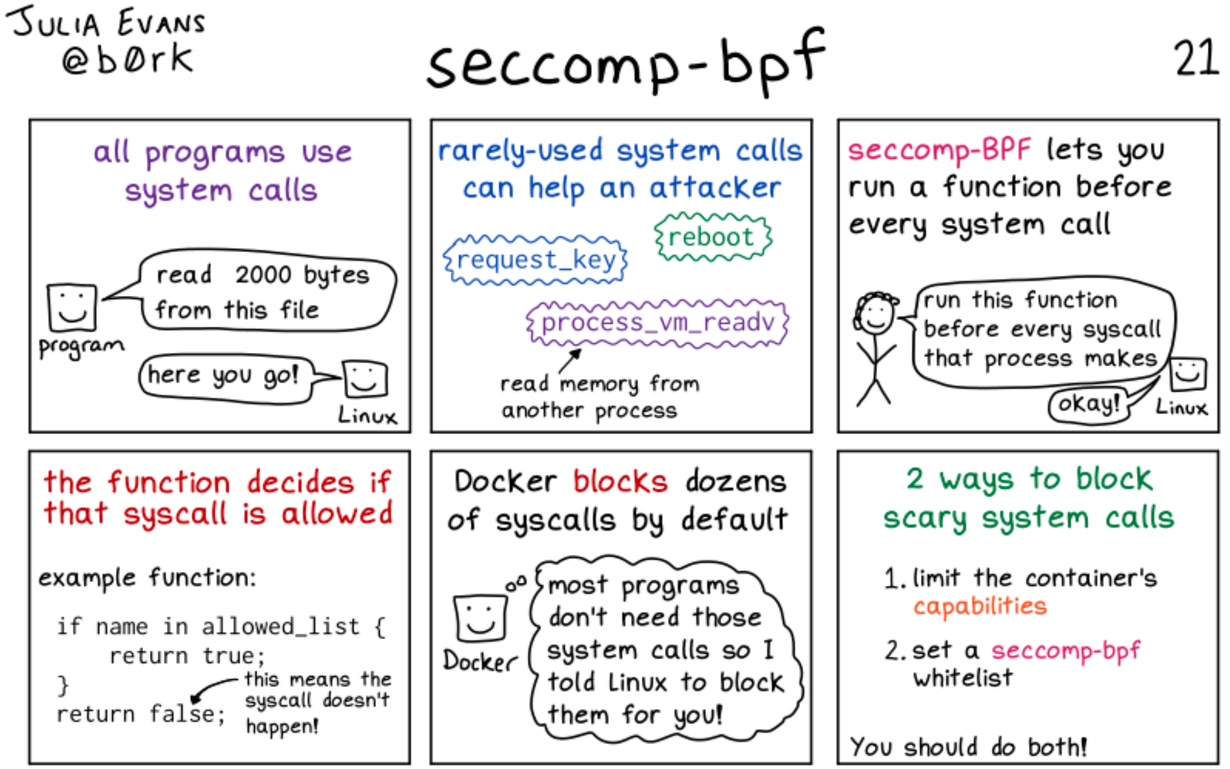


Linux Kernel: seccomp+bpf

- A process **can install multiple BPF filters**
  - After installed, cannot be removed or deactivated
  - Propagated to all descendant processes (including `execve`)
- BPF parameters: system call, arguments, architecture
- Filter **executed by the kernel in all system calls**, and returns:
  - `SECCOMP_RET_KILL`: kill process
  - `SECCOMP_RET_ERRNO`: reject system call and return an error
  - `SECCOMP_RET_ALLOW`: allow the system call

Docker: seccomp+bpf

- Docker **isolates containers using** `seccomp-bpf`, how?
  - **Containers are sub-processes of the docker-engine**
  - Controls calls made by sub-processes to the kernel using `seccomp-bpf`
  - **The default policy blocks many system calls by omission**, including `ptrace`
  - The **policy can be refined** via a command-line configuration file:



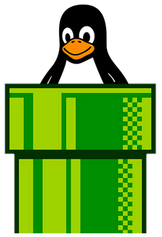
https://docs.docker.com/engine/security/seccomp/

```
—security-opt seccomp=/path/to/seccomp/profile.json
```

Linux Kernel: mitigations

- Vulnerabilities in the Linux Kernel ⇒ privilege escalation 
- Two famous CVEs: overwriting data in read-only files

  - Dirty COW (CVE-2016-5095): race condition on copy-on-write (COW) feature.
    Why? To support debuggers…
  - **Attack:** calling `ptrace()` or writing to `/proc/self/mem`

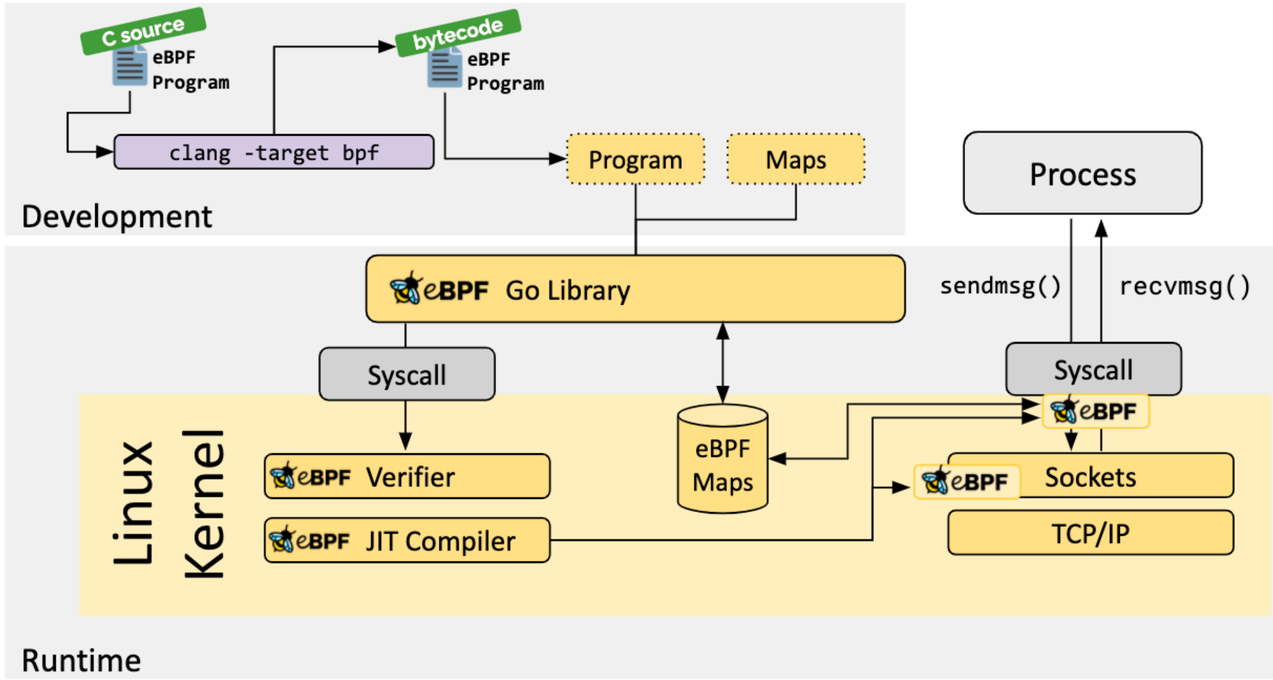


  - Dirty Pipe (CVE-2022-0847): zero-copy optimisation on memory pages.
    Why? To avoid expensive copy…
  - **Attack:** calling `splice()` to push data to a pipe

- Without or until updating the kernel, a **possible mitigation** is to block specific system calls and to limit capabilities or access to certain files

extended BPF (eBPF)

- Introduced in 2014, currently “hot topic” (e.g., Kubernetes)
- Typically: rigid kernel
- Novel idea: efficiently and securely **run userspace programs inside the kernel** (e.g., monitoring); combines both SCI and SFI
  - ***hooks* on syscalls or other events**
  - **formal verification** of *bytecode* (*memory safety*, *information flow security*, *termination*)
  - **JIT compilation** of *bytecode* for *in-kernel code*

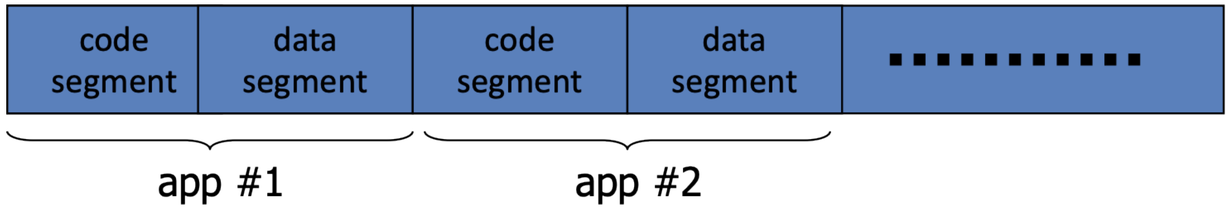


https://ebpf.io/

Software Fault Isolation

Software Fault Isolation

- Consider a particular form of sandboxing, designed for minimum overhead
- **Goal: limit the memory region available to an application (least privilege 🕮)**
  - Assign a memory segment to each application and enforce that access is within the correct region
  - Control code that is executing
    - Dangerous operations **must be** preceded by guards
  - Usually by instrumenting binaries, but can also be enforced at compile time (e.g., CompCert https://doi.org/10.1007/978-3-030-17184-1_18)



- **Alternative: using different address spaces (Inter-Process Communication overhead)**

Software Fault Isolation

- What are **dangerous operations**?
  - Clearly includes **load/store** operations on memory
    - Before access, add guard, verify segment or force segment address
  - But also **jumps**, why?
    - A jump may be used to execute external code without guards
    - Necessary to validate the target addresses with similar mechanisms
  - Implementation can be hard, depending on the architecture
    - e.g., hard directly on x86 assembly (registers, many dangerous instructions, etc.)

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

Google NaCl

Salt (Sodium Chloride)

- Google Native Client: sandbox for native code (e.g. games) on Chrome
  - Pioneering project to migrate many C/C++ libraries to the browser POSIX-compliant (file system, sockets, threads, etc)
  - Code compiled “ahead-of-time”, much faster than JavaScript compiled “just-in-time”
  - SFI + CFI
- Overhead of 5% in ARM and 7% in x86-64
- Discontinued in 2016 ⇒ WebAssembly
  - Nowadays: C/C++ $\Rightarrow^{LLVM}$ WebAssembly

Doom in NaCl

https://developer.chrome.com/docs/native-client/reference/sandbox_internals/x86-64-sandbox/

SFI essential in sandboxing

how webassembly provides software fault isolation

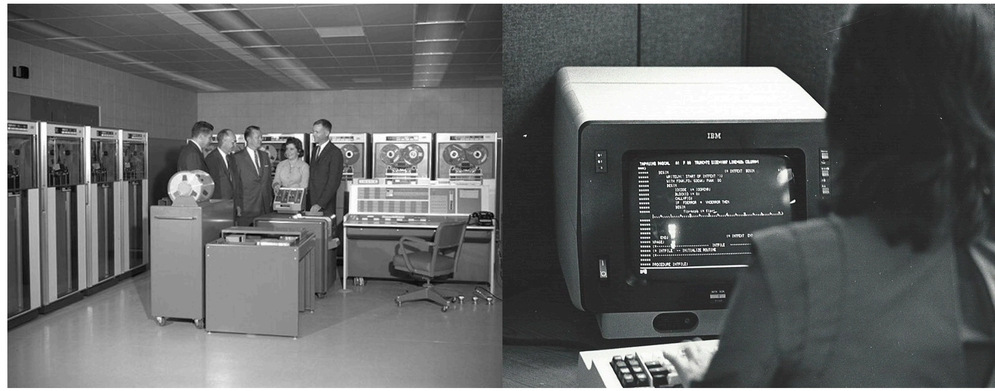
> At the specification level, "linear memory" is the only space that WebAssembly programs can access with its load and store instructions. Ensuring that this is true is the job of the WebAssembly VM, which is free to do it in any way so long as it meets the specification. In the VM I work on, Wasmer, 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

Virtual Machines

Virtual Machines: History

- **VMs 1960's**
  - Few computers, many users
  - VMs allow users to share a single computer



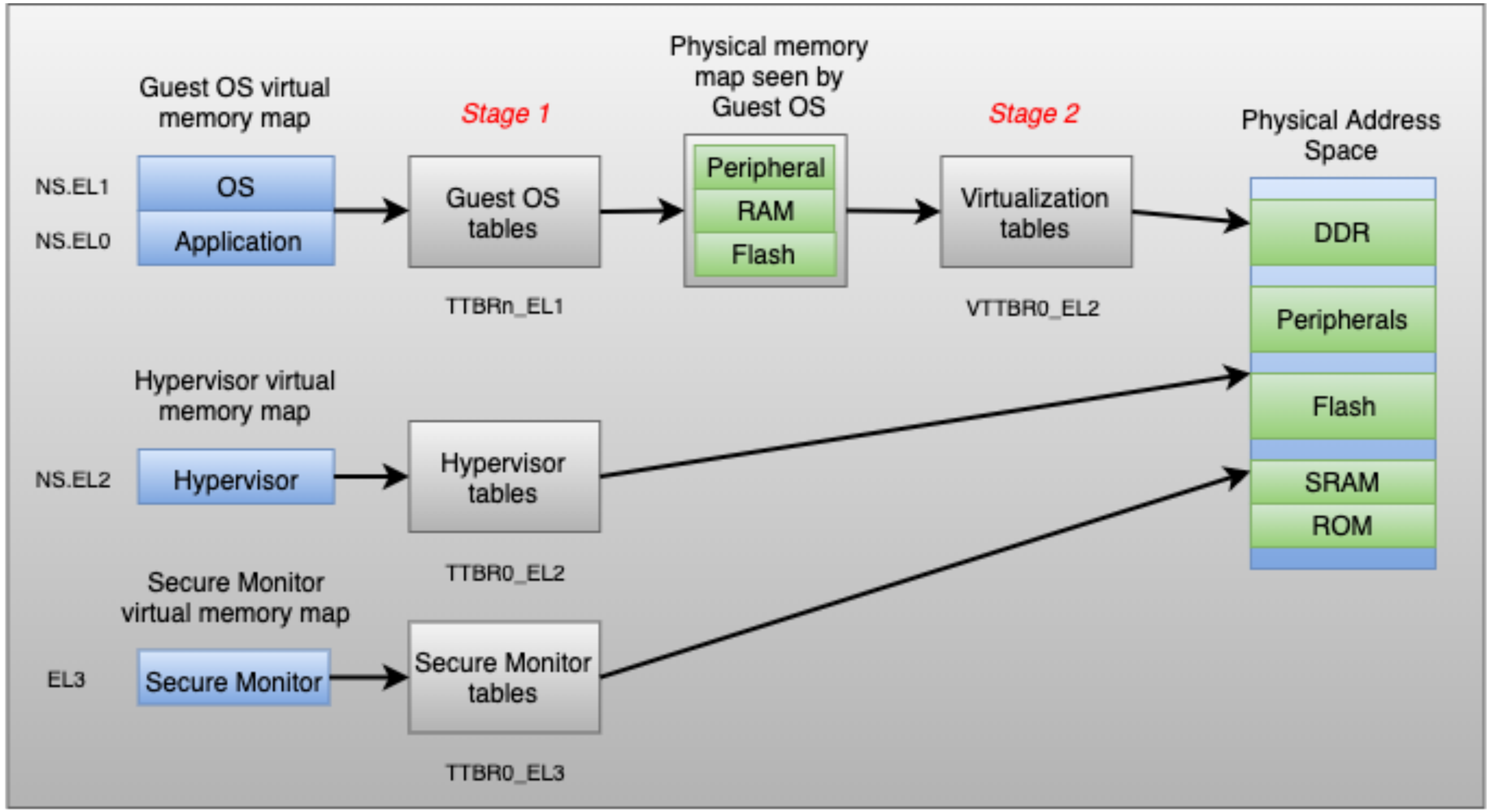
- **VMs 1970's – 2000** (non-existent)
  - Age of personal computers

- **VMs > 2000**
  - Many services and end devices
    - Print server, Mail server, Web server, tablet, cellphone, PC, …
  - VMs highly used in personal computers and clouds
  - **necessary to ensure isolation!**

Virtual Machines

- A very useful mechanism to enforce isolation in practical scenarios:
  - It is likely that an OS is vulnerable/infected/compromised
  - It is possibly that an hypervisor eventually becomes vulnerable
  - It is **improbable that both are vulnerable** at the same time
- **An hypervisor is (generally) simpler than an OS**
  - **simplifies validation** ⇒ there always remains some risk
  - **HW prepared for virtualisation** ⇒ smaller risk
    - e.g., virtual memory with various levels of address translation

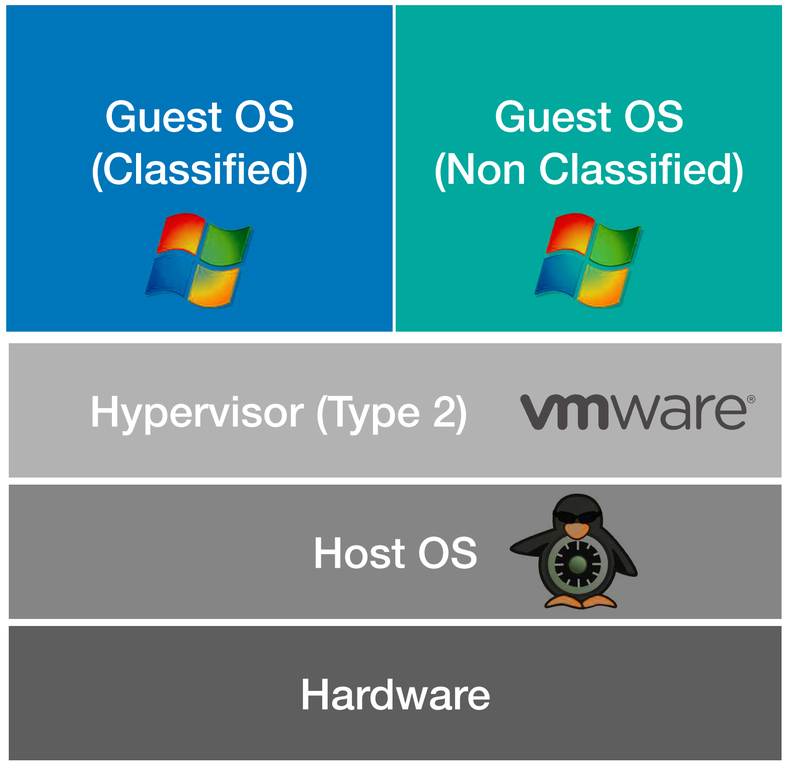
ARM Privilege Levels



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

Virtual Machines

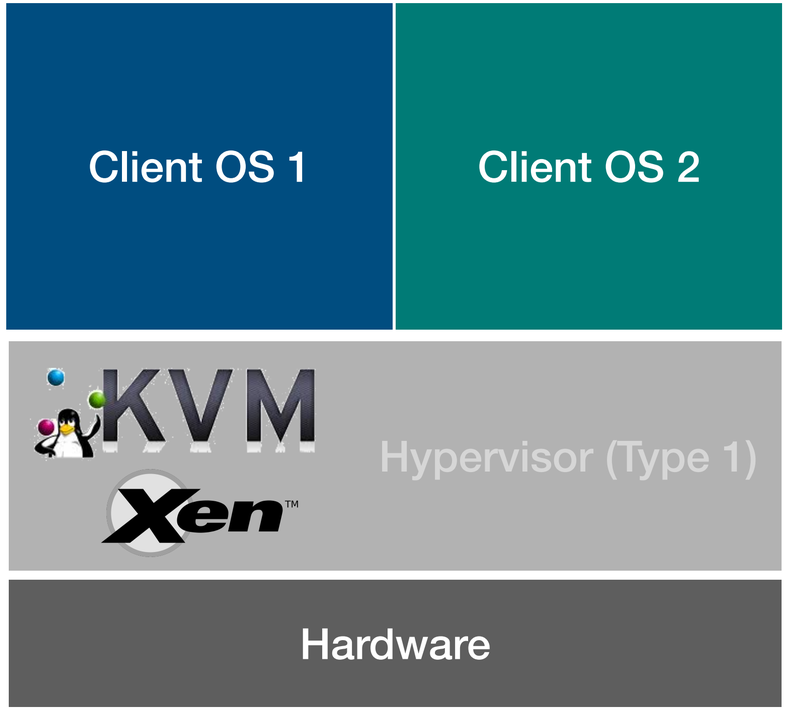
- Example of practical use:
  - NSA NetTop: handling **“classified”** and **“non-classified”** data in the same HW
  - **Solution:** OSes and commercial Virtualisation to ensure isolation
  - **Rationale:** for privileged information to leave the compromised OS, it is necessary to corrupt various sub-systems
  - **Risk:** side-channels



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

Virtual Machines

- Example of practical use:
  - Cloud provider eliminate host OS
  - Hypervisor directly manages the HW
  - Different systems for different clients in the same HW!
  - **Risk:** side-channels 



Virtual Machines

- Various CVEs (in open-source projects…)
- KVM is part of the Linux Kernel; KVM and Xen rely on Qemu for device emulation

Year	Code Execution	Bypass	Privilege Escalation	Denial of Service	Information Leak
2014	2	0	0	41	1
2015	3	0	0	29	4
2016	1	5	5	17	4
2017	6	5	6	38	8
2018	2	0	2	21	3
2019	0	0	5	20	0
2020	0	0	7	33	0
2021	0	0	0	3	0
2022	0	0	0	20	10
2023	1	0	0	0	1
2024	0	0	0	0	0
Total	15	10	25	222	31

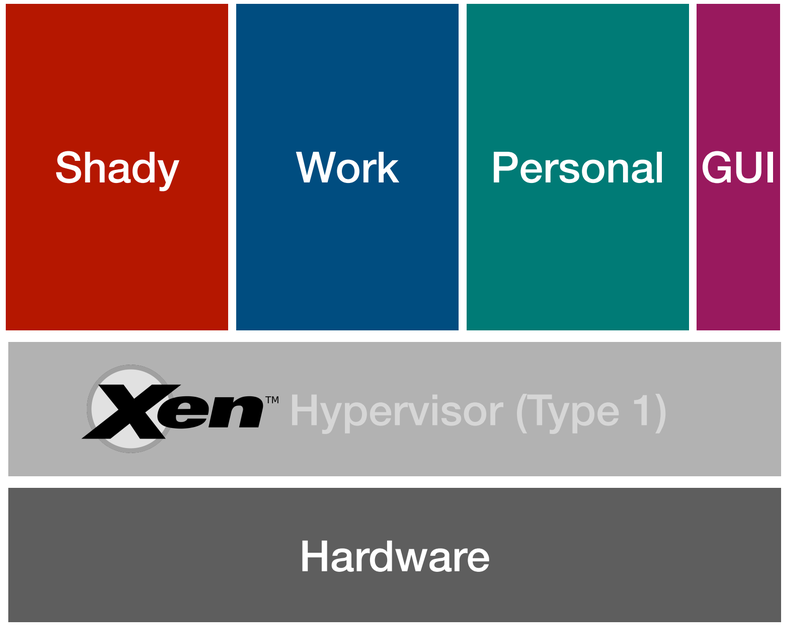
Year	Code Execution	Bypass	Privilege Escalation	Denial of Service	Information Leak
2014	26	0	0	17	1
2015	6	0	0	8	0
2016	10	0	0	80	3
2017	6	0	1	55	0
2018	5	0	0	18	0
2019	0	0	0	4	3
2020	2	0	0	17	0
2021	1	0	1	17	1
2022	5	0	0	17	0
2023	2	1	1	5	0
2024	0	0	0	4	0
Total	63	1	3	242	8

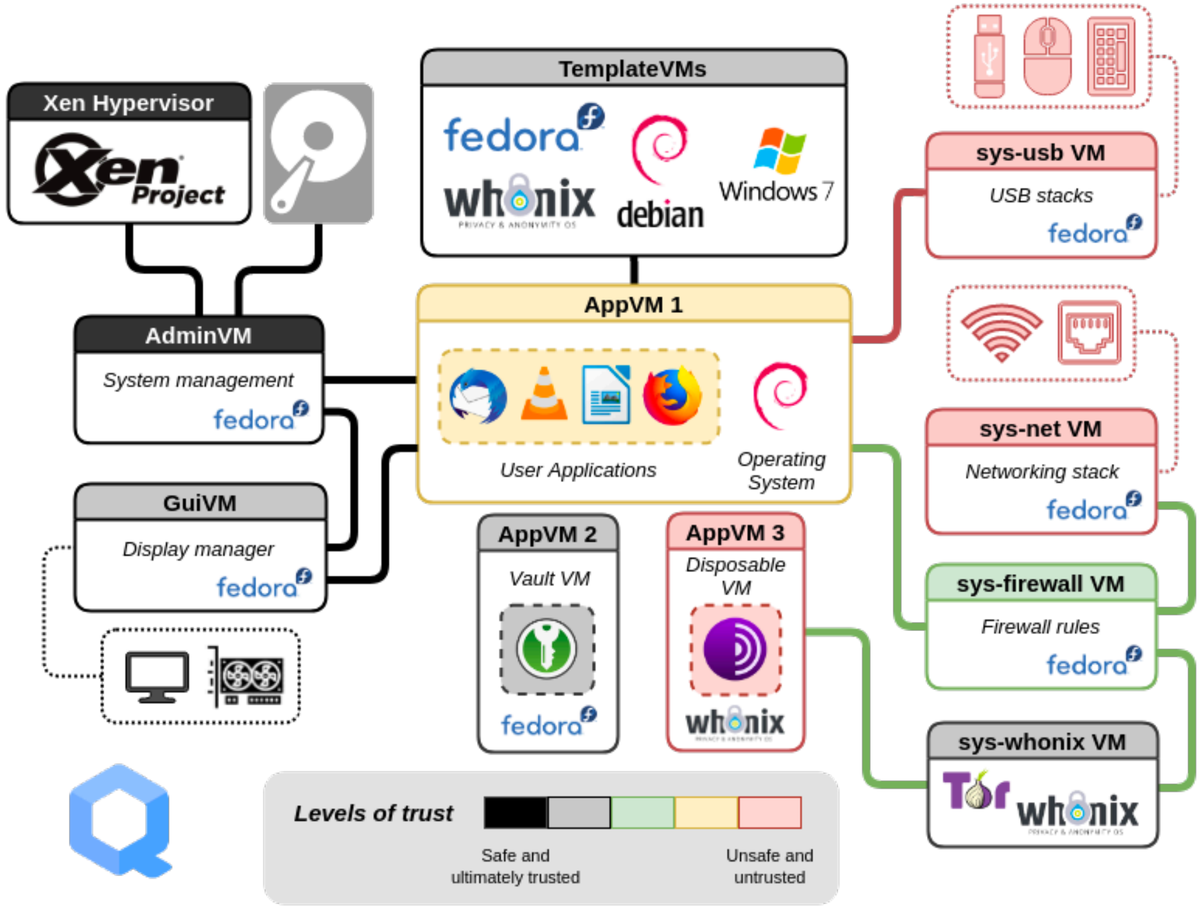
https://www.cvedetails.com/product/23463/XEN-XEN.html?vendor_id=6276

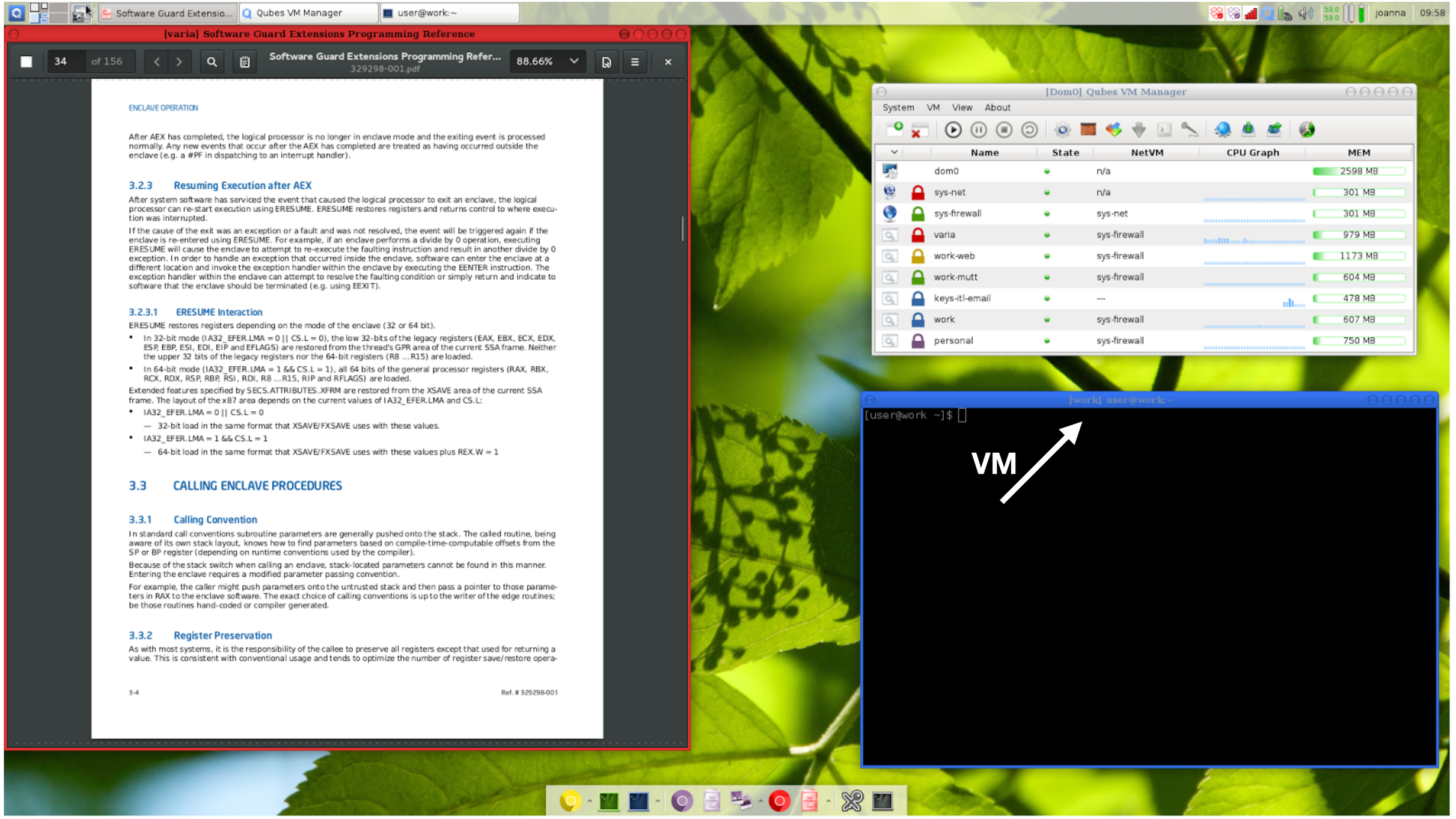
https://www.cvedetails.com/product/12657/Qemu-Qemu.html?vendor_id=7506

Virtual Machines

- Example of practical use:
  - **Qubes OS**: oriented for virtualisation
  - Multiple VMs







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

Breaking isolation

- **Side channels:**
  - Observing collateral effects (e.g., delays in access to memory)
  - e.g., a process transmits information to another while creating observable cache side-effects
- **Beware that modern processors are unable to ensure full isolation:**
  - many mechanisms in the shared HW allow leaks of information
  - famous  : Meltdown & Spectre exploit speculative execution


https://meltdownattack.com

Red Pill vs Blue Pill

- **Can a program know that it is running in a VM?**
- Why is this relevant?
  - **Malware**: generally analysed in virtual environments ⇒ avoid analysis
  - SW manufacturer demands running in **proprietary HW** / HW key
  - Same motivation for content protected by **copyright** (DRM)

Red Pill vs Blue Pill

Unsettling Truth

Contended Ignorance

[source: The Matrix]

Red Pill vs Blue Pill

- Can a program know that it is running in a VM?
- **There are various forms to detect virtualised environments**
  - Available instructions, specific HW features, etc
  - Memory latency (profiling)
  - The hypervisor uses part of the HW management mechanisms
  - A **malicious Guest OS** may detect its limited resources

Red Pill vs Blue Pill

- Can a **web page** know that it is running in a VM?
- Practical example (web):
  - How to detect websites that inject malware  ?
  - Running the browser in a VM and exploring websites (crawling)
  - Sites have evolved to detect virtualised environments! ⇒ 
    - Example: measuring latency of screen rendering

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

Red Pill vs Blue Pill

Hypervisors **offer**	Hypervisors **do not offer**
Compatibility: ensuring that software works “off the shelf”	**Transparency**: indistinguishable behaviour when virtualised (e.g., “archaic” instruction set with 8GB of RAM)
Performance: minimising impact of virtualisation	

- Joanna Rutkowska (founder of QubeOS) ⇒ can attackers revert the roles?

virtualisation rootkit (open-source, does not have to hide not to be detected)

technique to detect presence of virtualisation

https://en.wikipedia.org/wiki/Blue_Pill_(software)
http://theinvisiblethings.blogspot.com/2007/08/virtualization-detection-vs-blue-pill.html

Acknowledgements

- This lecture's slides have been inspired by the following lectures:
  - CS155: Isolation and Sandboxing