Information Security

System Security 4 - Sandboxing and Isolation

1 December 2023

Motivation

- Bug-free code is hard to write
- Impact of exploits should be minimized
- Sometimes, untrusted code has to be executed
- Restrict access as much as possible

System Hardening

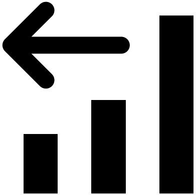
Principle of Least Privilege (PoLP)

PoLP

»Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.«

- Jerome Saltzer, Communications of the ACM

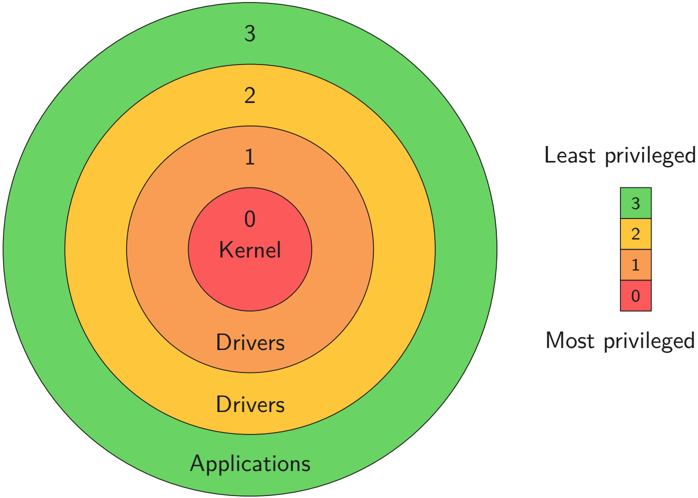
Principle of Least Privilege (PoLP)



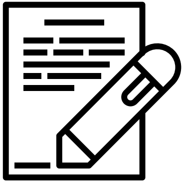
- Important design decision
- Only give permissions that are actually needed
- Fewer permissions → fewer attack surfaces

→ User account vs. admin account

Example: x86 Protection Rings



Driver Signatures



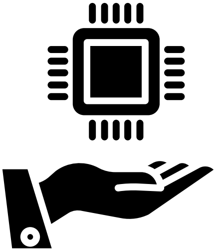
- Low-privilege user-space application can simply be executed
- Drivers have high privileges (ring 2 to 0)

→ Don't accept all drivers

- Only load drivers if they are signed by trusted vendor

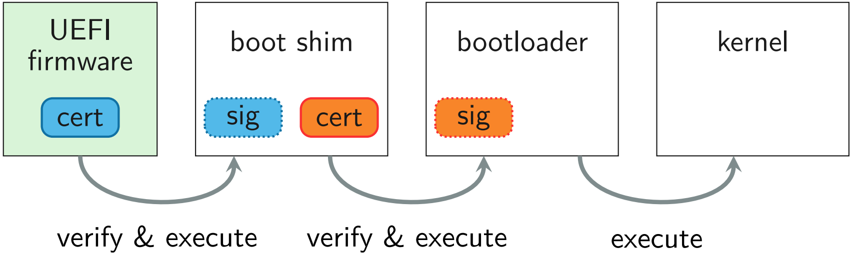
→ Root attacker cannot simply inject code into kernel

Secure Boot



- UEFI supports secure boot
- UEFI ROMs, boot loader, kernel must be signed
- Public key in firmware to verify signatures
- Control-flow only handed over on successful verification

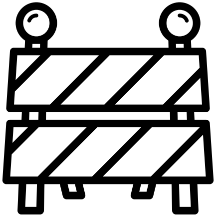
Example: Secure Boot Ubuntu



Microsoft UEFI CA certificate

Ubuntu CA certificate

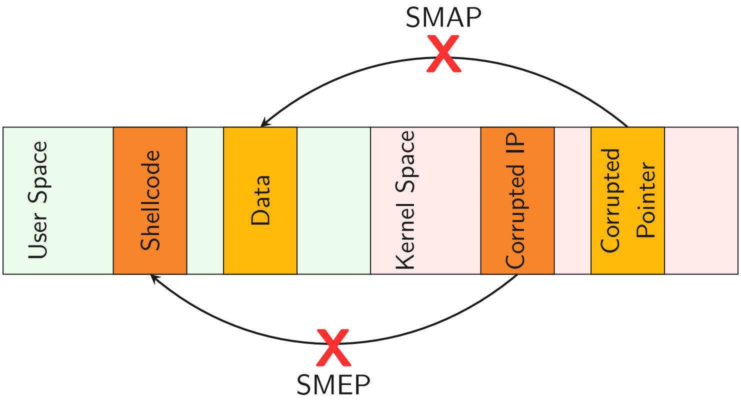
SMAP/SMEP



- Bug in kernel allows accessing all user-space memory
- Reduce the impact of kernel bugs
- Explicitly enable/disable user-space access

→ SMAP and SMEP

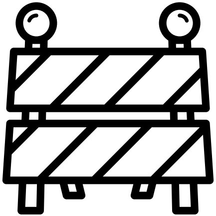
SMAP/SMEP



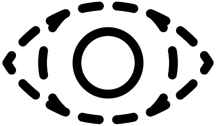
SMAP: Supervisor Mode Access Prevention

SMEP: Supervisor Mode Execution Prevention

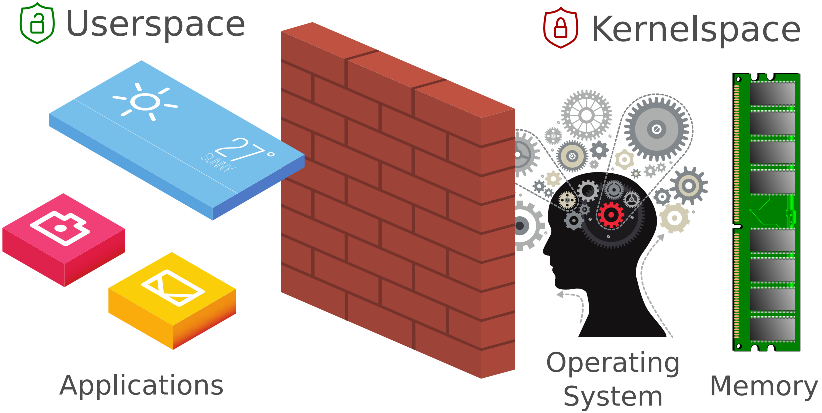
SMAP/SMEP



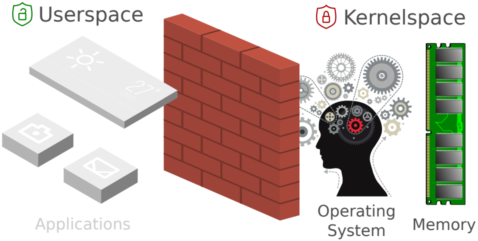
- SMEP prevents execution of user-space code → never needed
- SMAP prevents access to user-space data → sometimes needed
- `stac` and `clac` instructions → enable/disable access
- Every user-space data access surrounded by `stac`/`clac`
- Supported in Linux, macOS, soon in Windows 10



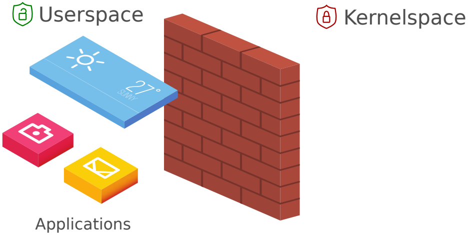
- KAISER/KPTI is other way round
- Unmap the kernel in user space
- Kernel addresses are then no longer present
- Protection against microarchitectural attacks (e.g., Meltdown)


Userspace
Kernelspace
27°
SUNNY
Applications
Operating System
Memory

Kernel View



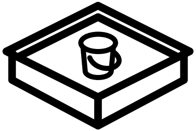
User View



context switch

Sandboxing

Sandboxing



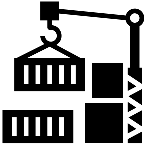
- A sandbox is a restricted environment for a program
- Resources of the process are strictly controlled:
  - own filesystem
  - no network connection
  - limited amount of memory
  - limited CPU time
  - ...
- Different approaches to sandboxing

Types of Sandboxing

- Multiple types of sandboxes
- Different advantages, disadvantages, and use cases

Language-level Sandboxing

Rule-based Execution

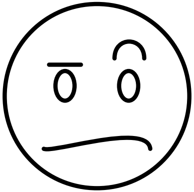


Container

Virtualization

Language-level Sandboxing

Untrusted Code

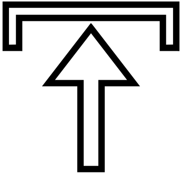


- Do not run native code
- Restrict untrusted code on the language level
- Languages without dangerous functionality (I/O, syscalls, ...)

$\rightarrow$ JavaScript, WebAssembly

- Access resources $\rightarrow$ ask user for permission

Language-level Sandboxing

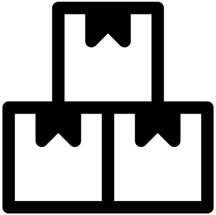


- Used in web browsers $\rightarrow$ website provides untrusted code
- Code cannot...
  - ...interact with the OS (syscalls)
  - ...communicate with other applications
  - ...access arbitrary memory (no pointers)
  - ...use unlimited memory
  - ...crash (memory safety)
- No malicious activity possible (in theory)

Interpreter

- Security guaranteed by the interpreter/runtime environment
- Interpreter does not provide dangerous functions
- Languages are memory safe
- A lot easier than sandboxing native code

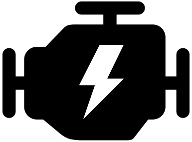
eBPF



- eBPF allows running sandboxed user code in kernel
- Originally to filter network packets
- Certain properties verified first:
  - Program must terminate

  → No loops/recursions (halting problem)
  - Jumps back only if they don't form loops
  - Call only to allowed functions
- Only loaded if analyzed and verified

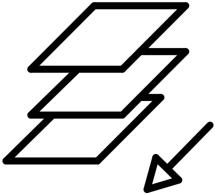
Runtime Environment



- Runtime environments and interpreters are complex
- Chrome JavaScript engine: $\approx$ 1.9 million lines of code (2019)
- Complexity introduces bugs $\rightarrow$ sandbox escape

$\rightarrow$ Additionally sandbox the interpreter

Site Isolation



- Chrome uses additional site isolation
- Every tab is a process

→ Exploited tab cannot access other tabs or browser

Rule-based Execution

Rule-based Execution

- Rules what an application is allowed to do
- Usually multiple rules for an application
- Rules can be whitelists or blacklists
- Multiple rules are combined to a policy/profile

Syscall Filtering

- Applications can use seccomp-bpf to restrict syscalls
- First define which syscalls are required
- Then block all other syscalls
- Attacker is restricted to syscalls the application uses

→ In many cases no `exec`

Seccomp

- `seccomp-bpf` is used by many (commercial) sandboxes
  - Docker
  - Firejail
  - Mbox
  - LXD
  - minijail
- It is even possible to block certain syscall parameters (e.g., no `read` except from standard input)

Seccomp Example

```c
#include <stdio.h>
#include <seccomp.h>
#include <sys/prctl.h>
int main() {
  printf("step 1: init\n");
  prctl(PR_SET_NO_NEW_PRIVS, 1);
  prctl(PR_SET_DUMPABLE, 0);     // ptrace not allowed
  scmp_filter_ctx ctx;
  ctx = seccomp_init(SCMP_ACT_KILL); // blacklist everything
  // whitelist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0);
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 0);
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 1,
                   SCMP_A0(SCMP_CMP_EQ, 1));
  seccomp_load(ctx);
  fprintf(stdout, "step 2: only 'write' to stdout\n");
  fprintf(stderr, "step 3: should be blocked\n");
}
```

Seccomp Example

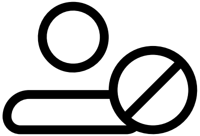
- Compile with seccomp support

```
% gcc seccomp.c -lseccomp -o seccomp
```

- Run protected application

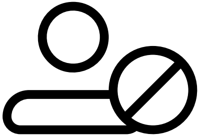
```
% ./seccomp
step 1: init
step 2: only 'write' to stdout
[1]    23414 invalid system call  ./seccomp
```

Mandatory Access Control



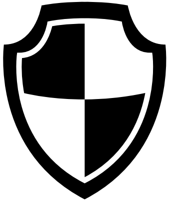
- General approach: mandatory access control system (MAC)
- Applies to many resources, not only syscalls
- Rules have a
  - *Subject*: Process or thread
  - *Operation*: Access, write, execute, ...
  - *Object*: File, TCP port, shared memory, syscall, ...
- OS enforces policy (i.e., set of rules)

Policies



- Policies are created/installed by administrator
- Users cannot override policies
- Policies can be enforced ($\rightarrow$ kill application on violation)...
- ...or just logged for later analysis

Implementations



- In Windows as Mandatory Integrity Levels
- Implemented in Linux as Linux Security Modules (LSM)
- Different modules in the kernel
  - SELinux
  - AppArmor
  - Smack
  - TOMOYO Linux

Example Application

- Show `man` section 3 (C Library Functions)
- For example, `man` page of `fopen`

```
% c fopen
```

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char* argv[]) {
    if(argc > 1) {
        char* args[] = {"man", "3", argv[1], NULL};
        execvp(args[0], args);
    }
}
```

Example AppArmor Policy

```
#include <tunables/global>

/usr/bin/c {
  #include <abstractions/base>
  #include <abstractions/bash>
  #include <abstractions/consoles>
  #include <abstractions/evince>

  /bin/dash ix,
  /bin/less mrix,
  /etc/groff/man.local r,
  /etc/manpath.config r,
  /usr/bin/c mr,
  /lib/x86_64-linux-gnu/ld-*.so mr,
  /usr/bin/groff mrix,
  /usr/bin/grotty mrix,
  /usr/bin/less mrix,
  /usr/bin/locale mrix,
  /usr/bin/man mrix,
  /usr/bin/nroff mrix,
  /usr/bin/nroff r,
  /usr/bin/preconv mrix,
  /usr/bin/tbl mrix,
  /usr/bin/troff mrix,
  /var/cache/man/oldlocal/index.db rk,
  owner /home/*/.lesshst r,
}
```

Example AppArmor Policy

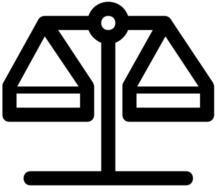
- Without policy: user can spawn shell from `man`
- Enforce policy

```
sudo aa-enforce /usr/bin/c
```

- Application still works, but “!/bin/bash” in `man` results in

```
sh: 1: /bin/zsh: Permission denied
```

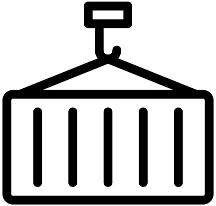
Rule-based Execution



- SELinux, AppArmor, and seccomp are widely used
- Not easy to create good policies...
- ...but secure and efficient for good policies
- Policies for popular applications can be found online

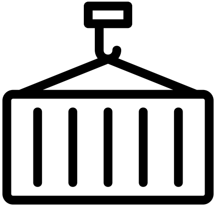
Container

Container



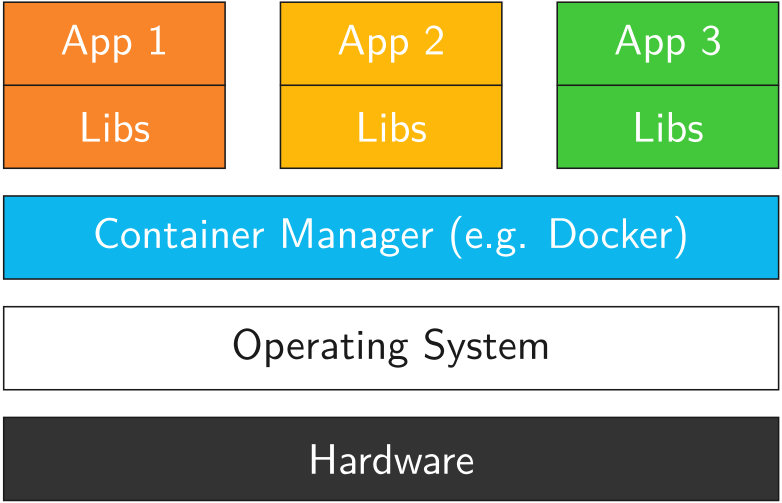
- Containers are operating-system-level virtualization
- Allows multiple isolated user-space instances
- Every container is assigned resources (e.g., part of memory, folders, ...)
- Application in container can only see assigned resources

Container

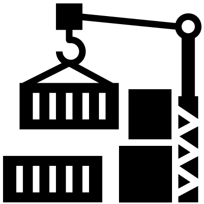


- One or multiple applications per container
- Own libraries but share the operating system
- File-system layer with copy on write
- “It works on my computer” $\rightarrow$ ship the environment

Docker


App 1
Libs
App 2
Libs
App 3
Libs
Container Manager (e.g. Docker)
Operating System
Hardware

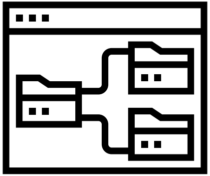
Container Manager



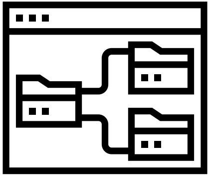
- Different container manager: Docker, OpenVZ, LXC, chroot, ...
- Require operating-system support
- Kernel is responsible for
  - Resource virtualization
  - Application isolation

→ namespaces and cgroups are the basis on Linux

Namespaces



- Namespaces isolate system resources between processes
- Default: all processes in same namespace
- Process can be started in new namespace
- Limits what the process (and it's children) can see
- Cannot inferfere with other namespaces



- Resources which can be isolated using namespaces

Process ID	Process sees only own and children processes
Mount	Own mounts for process
Network	Own network stack with virtual ethernet ports
IPC	Interprocess communication isolation
UTS	Own hostname and domain name
User ID	Own set of users which can map to host users
Cgroup	Hides the control group

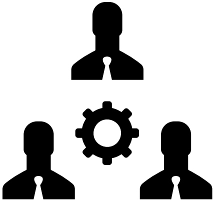
Namespace Example

```
% top
Tasks: 339 total, 1 running, 252 sleeping, 0 stopped
KiB Mem : 24423136 total, 13416376 free,  5017528 used
```

Create process-id namespace and start shell

```
% sudo unshare --fork --pid --mount-proc /bin/bash
$> top
Tasks: 2 total, 1 running, 1 sleeping, 0 stopped
KiB Mem : 24423136 total, 13043112 free,  5416304 used
```

Control Groups



- Control groups (cgroups) handle management and accounting of resources
- 12 different controllers (e.g., CPU, memory, I/O, ...)
- Every controller can have multiple cgroups
- A process (and its children) are in one cgroup per controller
- Controller and cgroups are in `/sys/fs/cgroup/`

Control Group Example

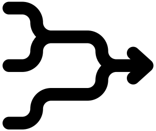
```c
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
int main() {
 mkdir("/sys/fs/cgroup/memory/ml", 0600); // create cgroup
 FILE* f =
   fopen("/sys/fs/cgroup/memory/ml/memory.limit_in_bytes", "w");
 fprintf(f, "%d", 64*1024*1024); // 64MB memory limit
 fclose(f);
 f = fopen("/sys/fs/cgroup/memory/ml/cgroup.procs", "w");
 fprintf(f, "%d", getpid()); // restrict own pid
 fclose(f);

 setgid(1000); setuid(1000); // drop privileges
 execv("/bin/bash", NULL);
}
```

Control Group Example

```
% sudo swapoff -a
% sudo ./memlimit
$> whoami
mschwarz
$> stress -m 1 --vm-bytes 60000000
stress: info: [5432] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
$> stress -m 1 --vm-bytes 70000000
stress: info: [5434] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: FAIL: [5434] (415) <-- worker 5435 got signal 9
stress: WARN: [5434] (417) now reaping child worker processes
stress: FAIL: [5434] (451) failed run completed in 0s
$> dmesg
Memory cgroup out of memory: Kill process 5435 (stress) score 908 or
    sacrifice child
Killed process 5435 (stress) total-vm:76600kB, anon-rss:59184kB, file-rss
    :196kB, shmem-rss:0kB
```

Combining Everything



- Control groups limit physical resources
- Namespaces isolate system resources (including cgroups)
- Combine both $\rightarrow$ restrict resources for process(es)
- Basis of nearly all containers on Linux (e.g. Docker)

Example with Docker

Installing Docker

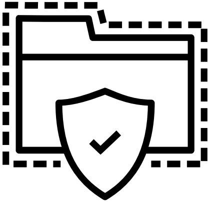
```
curl -fsSL get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

Using Docker:

```
docker run --rm -it ubuntu bash
```

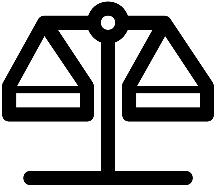
Starts a shell inside a container

Securing Containers



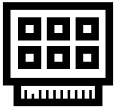
- Only use containers from trusted sources
- Limit the number of shared resources
- Keep host system up to date and patched
- Add seccomp (limit functionality) $\rightarrow$ additional security
- Unauthorized users should not interact with container manager

Advantage



- No additional OS $\rightarrow$ small overhead
- Fast start-up time
- Many containers can run on one host
- No complicated configuration of policies

Disadvantage



- Shared kernel of all containers and host
- All containers must use same OS

→ Kernel bugs are exploitable from containers

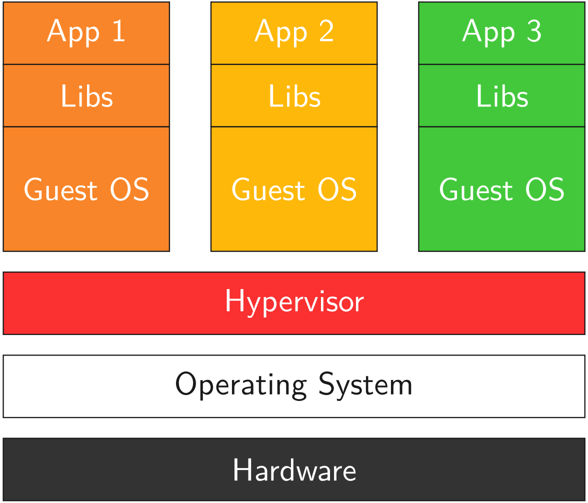
- Exploiting the kernel allows breaking out...
- ...and taking over the whole host

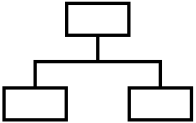
Virtualization

Virtualization

- Do not share kernel anymore
- Emulate entire system → Virtual machine
- Process runs inside own operating system
- No access to host

Virtualization





- Different types of hypervisors

  Bare metal Run directly on the hardware (e.g., Xen)

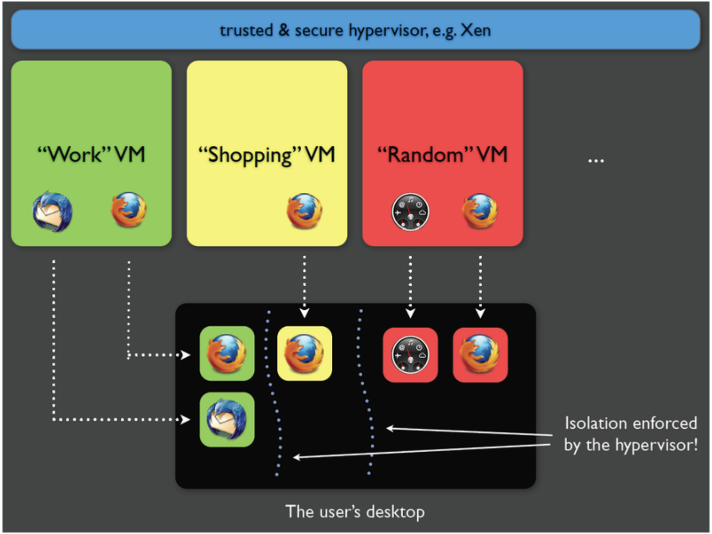
  Hosted Run on top/as part of the host OS (e.g., VirtualBox, KVM)

- Hypervisors emulates the machine hardware, e.g., graphic card
- OS and application are unaware of running inside VM

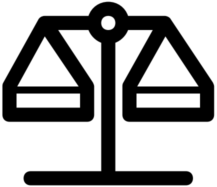
Qubes OS

- Qubes OS is a security-focused OS
- Provides security through isolation
- Multiple security domains, isolated by hypervisor
- All applications run inside (different) VMs
- Malicious software is limited to one domain

Qubes OS Security Domains

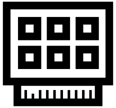


Advantage



- Virtualization provides best isolation
- Considered secure
- Applications not limited in functionality

Disadvantage



- Large resource overhead compared to containers
- Requires a guest OS for every isolated application
- Runtime overhead (e.g., paging, traps to hypervisor)
- Still not 100 % secure

VM Escape

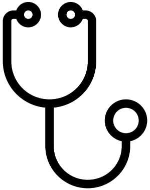
- VM escape: breaking out of a VM → interact with hypervisor

→ Access to host and all other machines

- VM escape usually using memory safety violations
- Mostly: bugs in drivers of emulated devices
- Extremely powerful, but complicated to mount

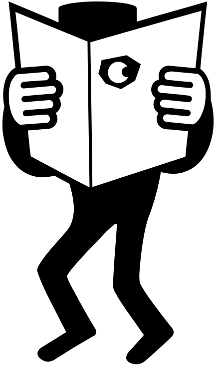
Summary

- Multiple ways of sandboxing applications
- Higher security $\rightarrow$ often more overhead
- Sandboxing mechanisms can be combined
- Generic defense $\rightarrow$ damage control
- Sometimes only solution (e.g. legacy software)



- Interaction of sandboxing with side-channel attacks?
- Run on the same hardware $\rightarrow$ shared resources
- Often just require memory accesses and timer
- Available in most sandboxes
- No real protection against side-channel attacks

Microarchitectural Attacks



Microarchitectural attacks shown from

- JavaScript (Spectre, Prime+Probe, Rowhammer, ...)
- eBPF (Spectre)
- Docker (Prime+Probe, ...)
- VMs (Spectre, Prime+Probe, Rowhammer, ZombieLoad, ...)

Some only work from VMs → Foreshadow-NG

Isolation

Motivation

- Sandboxes assume trusted system and untrusted application

→ Protects the system from harm

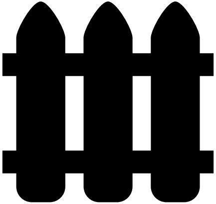
- Sometimes, we want to protect the application from the system
- Assumption: untrusted system, trusted application

→ Isolation of application

Motivation

- Applications for isolation:
  - Working with sensitive data (e.g., passwords, money)
  - Distrusting the cloud provider
  - Intellectual property (e.g., algorithms)
  - Rights management (DRM)
- Ensures security even against active attacks

Isolation



- Requires some form of hardware support
- Well-known isolation: user space - kernel space
- Protects OS against malicious applications
- Enforced by the hardware ($\rightarrow$ page table)
- Similar concepts to protect application from OS

Trusted Computing Base (TCB)

- Trusted computing base (TCB) is everything required to guarantee security
- Has to be trusted
- No security without a TCB
- Exploiting TCB → undermine entire security
- TCB should be as small as possible

Trusted Computing Base (TCB)

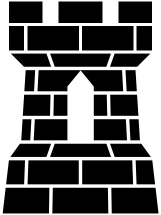
- CPU and firmware usually in the TCB
- Kernel and system programs usually in TCB

→ Protected by the hardware (→ protection rings)

- For sandboxes: sandbox in TCB
- What if we don't want to trust so many elements?

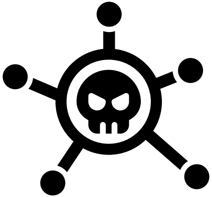
Trusted-Execution Environments

Trusted-Execution Environments



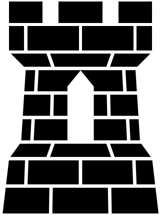
- Secure area of a CPU
- Integrity and confidentiality guarantees for code and data
- Hardware still shared with other applications
- (Nearly) no performance impacts

Trusted-Execution Environments Threat Model



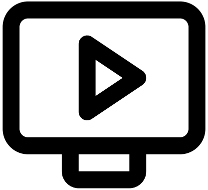
- Assumptions in TEEs:
  - Attacker controls the OS
  - Only the CPU is trusted ($\rightarrow$ TCB)
- TEE memory is encrypted and inaccessible to OS
- TEE has access to OS

Trusted-Execution Environments



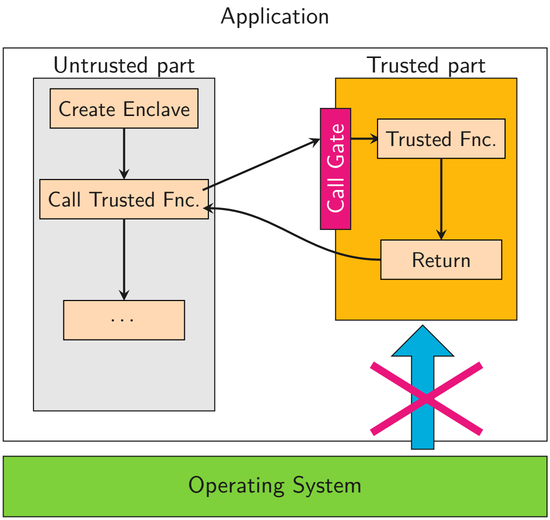
- Implementations for various CPUs
  - Intel: Software Guard Extension (SGX) and Management Engine (ME)
  - ARM and AMD: TrustZone
- Widely used in mobile phones

Real-World TEE Example

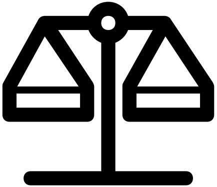


- Netflix uses Widevine DRM
- DRM in TrustZone
- Video is directly drawn on screen
- No app (not even root) can access video data

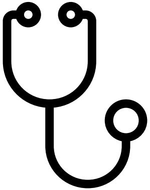
Execution Flow (SGX)



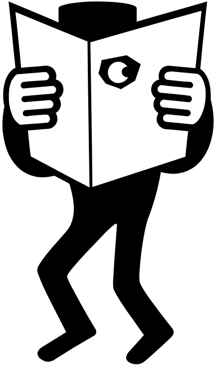
Advantages and Disadvantages



- Hardware-assisted protection of sensitive data
- Small overhead
- Could be abused for malicious software
- Bad code in TEEs is still exploitable
- No protection against side-channel attacks

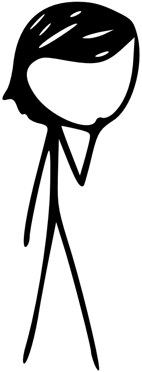


- Interaction of TEE with side-channel attacks?
- Run on the same hardware → shared resources
- Stronger attacker: malicious operating system
- No real protection against side-channel attacks



- Microarchitectural attacks shown on SGX via
  - Branch predictors
  - Caches
  - Interrupt latency
  - Page tables
  - Exceptions (cf. Foreshadow)
  - Transient-execution attacks (cf. ZombieLoad)
- Considered out-of-scope

What if?



- Enclaves are black boxes
- Protected from all applications and OS
- What if they contain malicious code?
- Can we hide zero days?

State-of-the-art Malicious Enclaves

- Side-channel attacks from SGX (Prime+Probe) → steal secrets from system
- Fault attacks from SGX (Rowhammer) → manipulate system/denial of service
- Return-oriented programming from SGX → break out of enclave

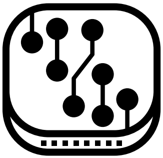
Threat Model

Intel's Statement

[...] Intel is aware of this research which is based upon assumptions that are outside the threat model for Intel SGX. The value of Intel SGX is to execute code in a protected enclave; however, Intel SGX does not guarantee that the code executed in the enclave is from a trusted source [...]

Hardware Isolation

Hardware Isolation



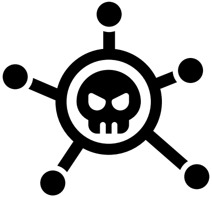
- TEE is not fully isolated ($\rightarrow$ shared hardware)
- Hardware security modules (HSM) are physically isolated
- Dedicated hardware, nothing shared
- Can be an external device or a plug-in card

External HSM

Internal HSM

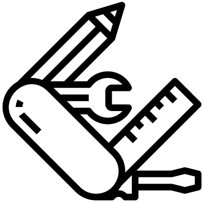
Photo by Wileyfh / CC BY

HSM Threat Model



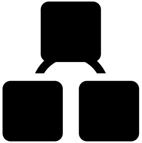
- Protection of high-value cryptographic keys
- Untrusted environment
- Attacker controls OS and has physical access
- Attacker tries to actively attack HSM

HSM Features



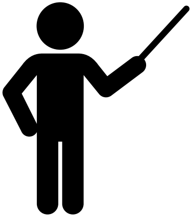
- Contains a crypto processor for
  - Secure key generation and management
  - Digital signatures
  - Data encryption/decryption
- Physical and logical protection of data
- Sometimes secure timestamp and strong random number generator

HSM Applications



- PKI environments (e.g., certification authorities)
    - Store and handle asymmetric keys
- Card payment systems (banks)
    - Manage smart cards
    - Authorize transactions
- Cryptocurrency wallets
- Handy-Signatur

Isolation Summary



- Isolation allows protecting applications in hostile environments
- Less shared resources $\rightarrow$ better isolation
- Isolation is sometimes similar to sandboxing...
- ...but mostly an orthogonal problem
- Can be combined for isolation in both directions
- Choose the methods which best fit your threat model