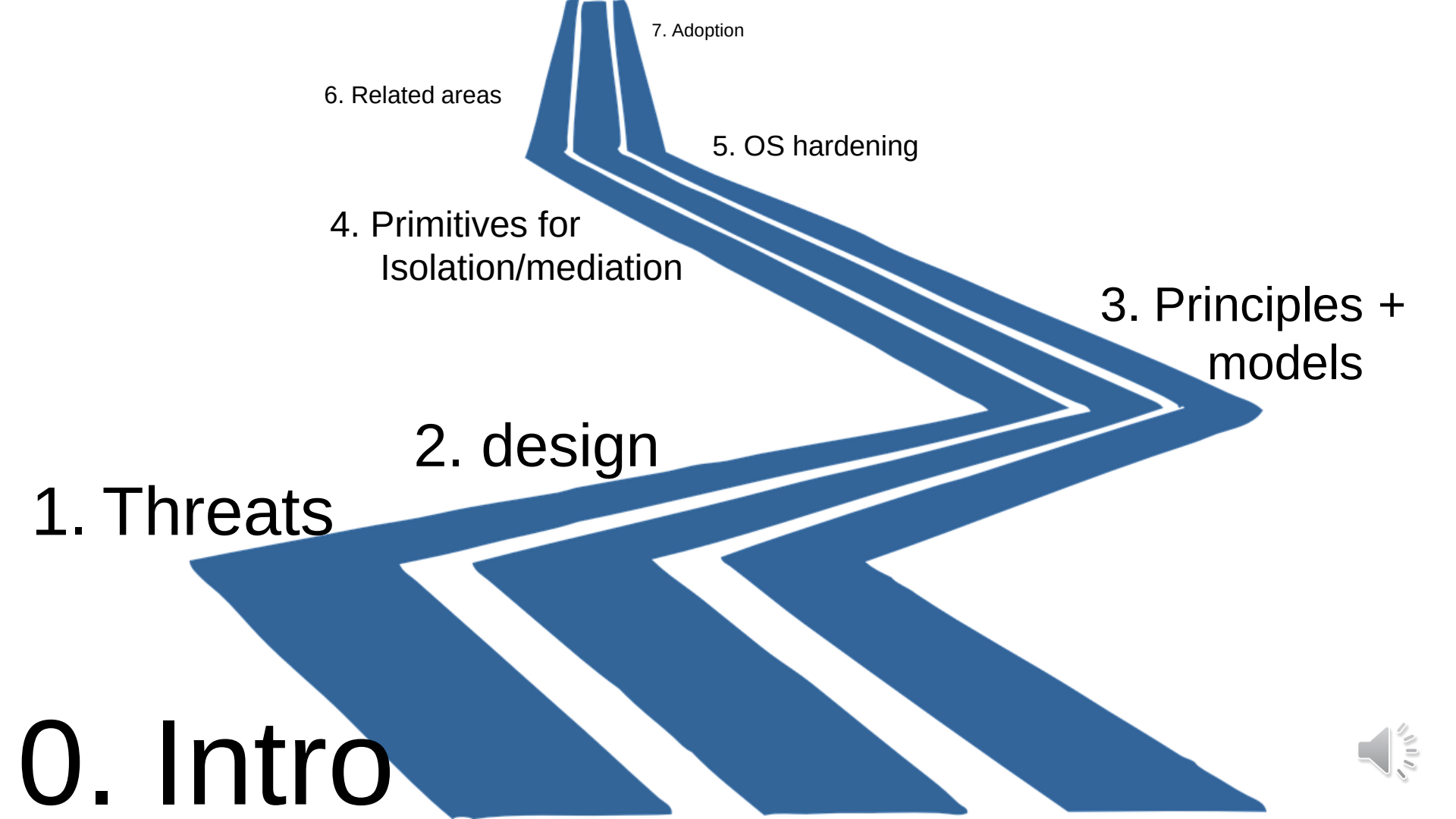
CyBOK

Operating Systems & Virtualisation Security Knowledge Area

Herbert Bos

VUSec


7. Adoption
6. Related areas
5. OS hardening
4. Primitives for
Isolation/mediation
3. Principles +
models
2. design
1. Threats
0. Intro

0. Introduction

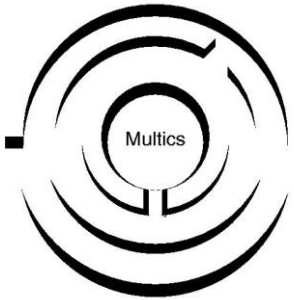
Operating systems and VMs are old

Much of the groundwork was done in the 60s and 70s

Interest in security grew when multiple security domains became important

We will see the origin and importance of principles and security primitives

Early OSs such as Multics pioneered many of these in a working system



Much has stayed the same, much has changed

Old days: security mostly focused on isolating (classes of) users from each other

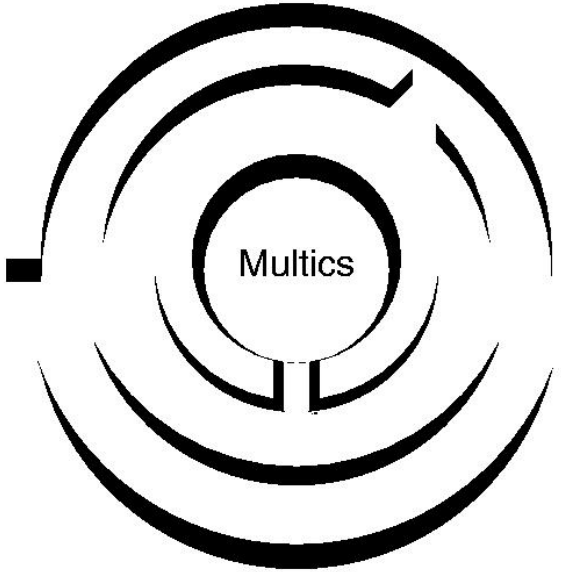
- processes / VMs

Same:

- isolation / processes / VM
- principles

Changed:

- some more security domain
- threats (and thus security mechanisms)



Much has stayed the same, much has changed

Old days: security mostly focused on isolating (classes of) users from each other

processes / VMs

Same:

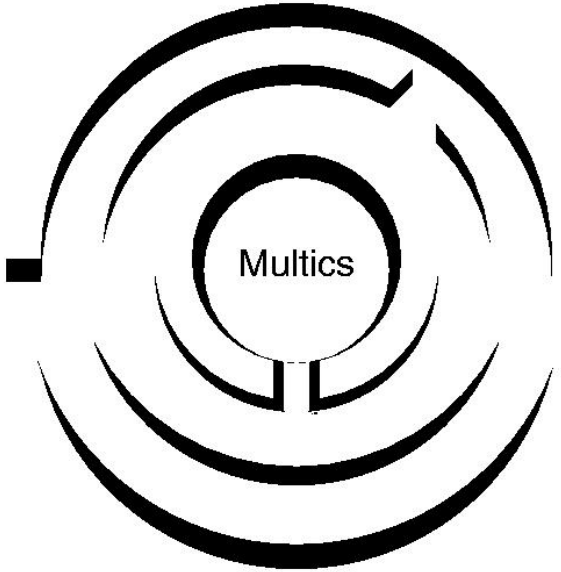
isolation / processes / VM

principles

Changed:

some more security domain

threats (and thus security mechanisms)

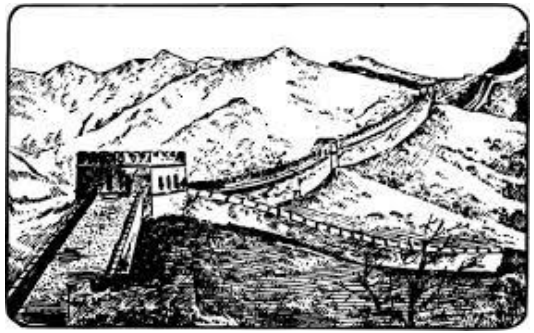


Security domains

We focus mostly on systems with multiple, mutually non-trusting security domains

process / kernel / hypervisors / trusted execution environments / etc.

So: not simple embedded devices with a single security domain



security domains in the real world

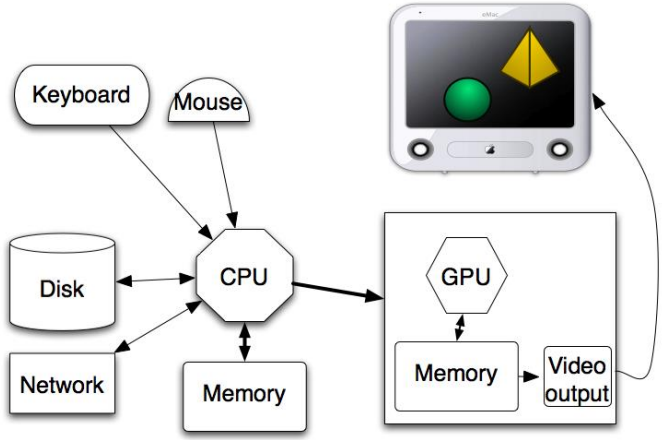
Also: we generally take an OS perspective

only mention VMs when the distinction is important

Operating systems

Manage the system resources in a safe/secure way. Traditionally, things such as:

- CPU time
- Memory
- Network
- Disk / FS
- ...



However, this may no longer be enough

- Attacks use micro-architectural features: TLBs, caches, MMU, speculation
- For security, we have to manage these resources also

Operating systems / hypervisors

Lowest levels of software stack form the bedrock on which security is built.

OS is able to execute privileged instructions, not available to normal programs

OS typically offers means to authenticate

OS typically offers means to isolate security domains / users

While application should enforce security beyond this point, OS guarantees that only authorised processes can access files, memory, CPU, or other resources.

Two aspects of operating system security

Security *by* the OS

Security *of* the OS

1. Threats

Threats

Attackers are interested in violating the security guarantees provided by the OS:

- leak confidential data (e.g., crypto keys),
- modify data that should not be accessible (e.g., to elevate privileges)
- limit the availability of the system and its services (e.g., by crashing the system or hogging its resources).

Threats may originate in

- the processes running on top
- the data arriving from elsewhere (e.g., network packets)
- the devices connected (e.g., malicious devices)

Sometimes, we even include the OS or hypervisor itself in our list of threats

Known Threats

Attack	Description
Malicious extensions	Attacker manages to convince the system to load a malicious driver or kernel module (e.g., as a Trojan).
Bootkit	Attacker compromises the boot process to gain control even before the operating system gets to run.
Memory errors (software)	Spatial and temporal memory errors allow attackers (local or remote) to divert control flow or leak sensitive information.
Memory corruption (hardware)	Vulnerabilities such as Rowhammer in DRAM allow attackers (local or remote) to modify data that they should not be able to access.
Uninitialised data leakage	The operating system returns data to user programs that is not properly initialised and may contain sensitive data.
Concurrency bugs and double fetch	Example: the operating system uses a value from userspace twice (e.g., a size value is used once to allocate a buffer and later to copy into that buffer) and the value changes between the two uses.
Side channels (hardware)	Attackers use access times of shared resources such as caches and TLBss to detect that another security domain has used the resource, allowing them to leak sensitive data.
Side channels (speculative)	Security checks are bypassed in speculative or out-of-order execution and while results are squashed they leave a measurable trace in the micro-architectural state of the machine.
Side channels (software)	Example: when operating systems / hypervisors use features such as memory deduplication, attackers can measure that another security domain has the same content.
Resource depletion (DoS)	By hogging resources (memory, CPU, buses, etc.), attackers prevent other programs from making progress, leading to a denial of service.
Deadlocks/hangs (DoS)	The attacker brings the system to a state where no progress can be made for some part of the software, e.g., due to a deadlock (DoS).

Table 1: Known attack methods / threats to security for modern operating systems

2. Role

The role of OSs in security

At a high level, OS is tasked with managing resources of a system to guarantee a foundation on which we can build secure applications with respect to:

- confidentiality
- integrity
- availability

Aim: to provide isolation of security domains and mediate of all operations that may violate the isolation

Some OSs even regulate the flow of information: “Top secret data can only flow to programs with sufficient authority.”

Data plane vs control plane

Isolation involves both data plane and control plane:

- In memory isolation, OS operates at control plane when configuring the MMU
- When operating on syscall arguments, the OS often operates at both planes.

Lack of separation between planes may lead to vulnerabilities.

> Example: OS reuses memory that previously belonged to one security domain (with access isolation enforced by MMU) in another domain without overwriting the (possibly sensitive) data on that page.

The *design* of the OS has security implications

Where the earliest systems supported only single security domains, this is no longer the case.

We can structure the OS in different ways:

- as a monolithic blob
- as many small components
- …

We will introduce alternatives first and then talk about security implications

OS structure

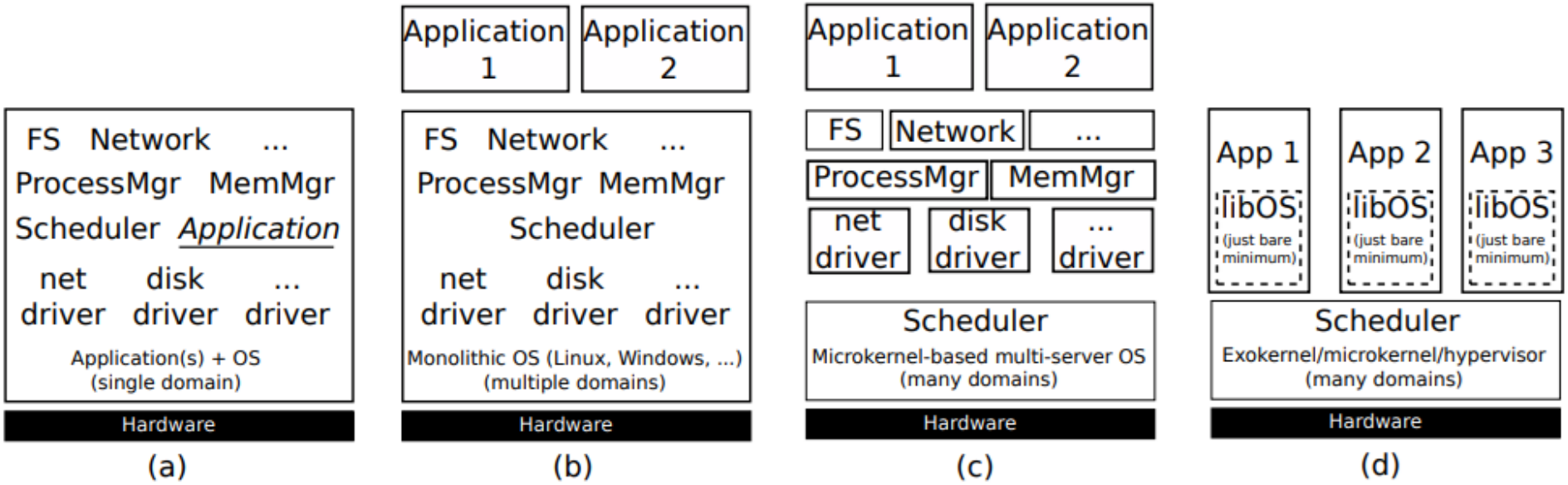
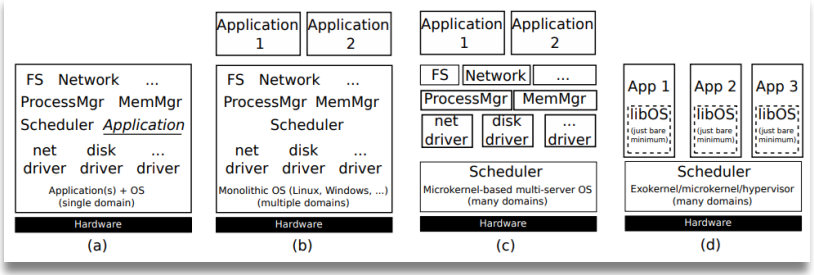


Figure 1: Extreme design choices for operating systems: (a) single domain (sometimes used in embedded systems), (b) monolithic OS (Linux, Windows, and many others), (c) microkernel-based multi-server OS (e.g., Minix-3) and (d) Unikernel / Library OS

OS structure and security implications



Design (a): **all in one**

Early OSs, many embedded systems: no isolation

Design (b): **monolithic**

Linux/Windows/...: isolates kernel + processes from processes, but 1 vuln in OS and we're toast

Design (c): **multi-server microkernel**

MINIX 3: small TCB, POLA, one compromised component doesn't take down system (but slower due to IPC)

Design (b): **libOS, unikernel**

Exokernel, Nemesis, unikraft: Minimal system to support 1 app → library OS → may run as VM → vuln ruins everything for (only) this app

What about the Internet of Things?

Really simple devices often use special OS of a few KB

Example: RIOT can be as small as 10 KB and run on 8b microcontrollers with or without features such as MMUs

The debate: Andy Tanenbaum vs Linus Torvalds

“Not portable.”

“LINUX is obsolete!”

“Linux is more portable than MINIX.”

The debate raged on through the decades

In many variations, e.g:

- compartmentalization vs. monolithic
- ‘distributed’ system (multiserver) vs function calls (monolithic)

Interestingly *both* have become popular

- Linux + Android → dominant in mobile and server market (+ now on Windows!)
- Minix → installed in every Intel processor

...and virtualisation?

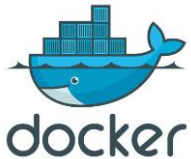
VM forms another security/isolation domain

Sometimes VMs used to partition OS (e.g., CubesOS)
not unlike multi-server
Sometimes microkernel very similar to hypervisor

Virtual machines vs. containers

VM images use independent storage, config, testing, maintenance, …
Containers:

- isolated from each other as much as possible
- have own kernel name spaces, resource limits, etc.,
- share underlying OS kernel, often binaries and libraries
- more lightweight



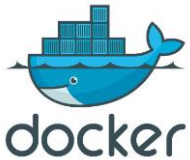
What about security?

Ignoring management, VMs often perceived as more secure than containers

- Strict partition
- Share only thin hypervisor

On other hand, containers *may be* more secure than virtual machines

- Lightweight, so we can break applications into ‘microservices’ in containers.
- Moreover, having fewer things to keep secure reduces attack surface overall.
- Management is simpler



3. Principles and Models

and Models

OS/VM foundation for all higher-level security

As a result ⇒ often discussed in terms of principles and models

Security principles serve as

- guidelines
- checklist
- ...

Models guarantee certain properties, such as the flow of information

and models: Saltzer & Schroeder

- Best known
- But others have added to list, e.g.:

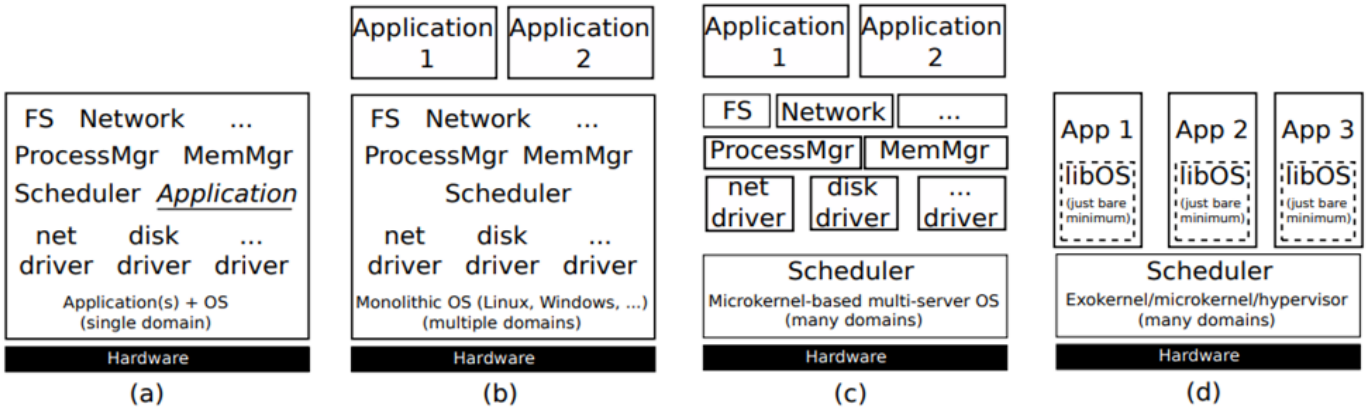
  Principle of Minimising Trusted Code (TCB)

  Principle of Intentional Use

  ...

Principle of…
Economy of mechanism
Fail-safe defaults
Complete mediation
Open design
Separation of privilege
Least privilege / least authority
Least common mechanism
Psychological acceptability

We can now apply these principles to OS designs



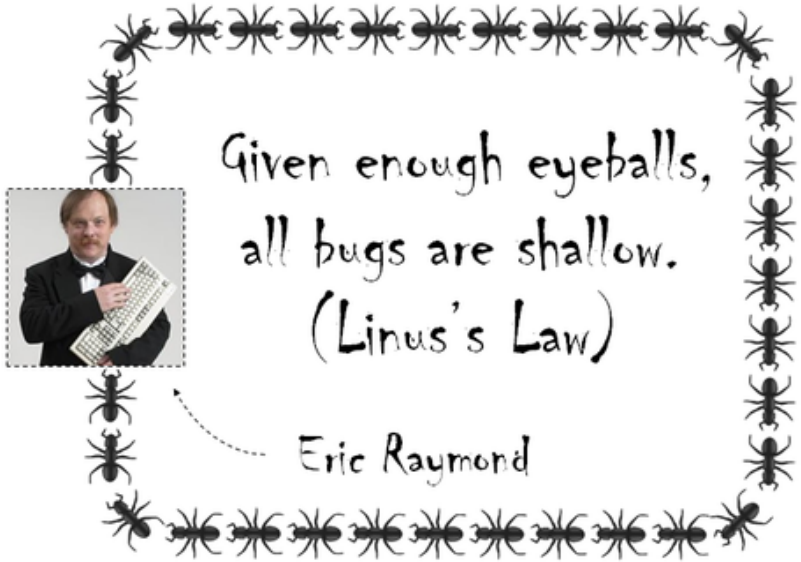
TCB: all software. All mechanisms ‘common’. Little notion of fail-safe defaults or mediation, etc.

OS shielded from processes, but itself still single security domain. Little privilege separation, POLA, etc.

Decomposition allows mediation, POLA, fail-safe defaults. Also: small TCB. Etc.

Single domain, but still small TCB, economy of mechanism, least common mechanism, mediation, etc.

Principle of Open Design



Hotly debated. We still find bugs that are decades-old in the Linux kernel and in security products such as OpenSSL.

Access models

Principles give us something to adhere to (or at least to target)

Access models give us exact rules

- For instance about how information may flow though the system

Access to data

Who should be allowed what access to what data?

Restrict access and flow of information

MAC, BAC and RBAC

MAC vs DAC

Mandatory Access Control (MAC): *system-wide* policy determines which users have clearance level to read or write which documents, and *users are not able* to make information available to other users without the appropriate clearance level.

Discretionary Access Control (DAC): users with access to object have some say over who else has access to it ⇒ they may transfer those rights to other users. Having only this group-based DAC makes it hard to control the flow of information in the system in a structured way.

However: may combine with MAC, so users have DAC withing constraints of MAC

MAC: different models

1. **Bell-LaPadula**

Security access model to preserve the confidentiality of information

initially created for US government in the 1970s

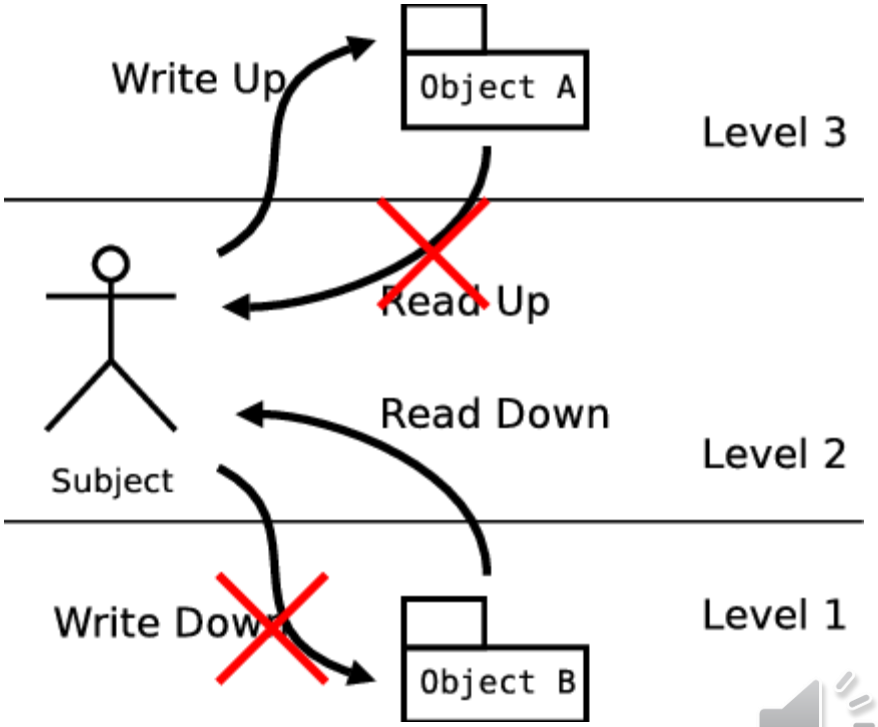
US military wanted users with different clearance levels on same mainframes

This requires “Multi-Level Security”

MAC: Bell-LaPadula

Confidentiality is everything.

Principal with medium clearance cannot read from higher level objects and cannot write (leak) data to lower level.



MAC: different models

2. **Biba**

   Confidentiality is not the only thing ⇒ what about integrity?

   We don’t want soldier to edit the documents of the general

   Biba model is opposite of Bell-LaPadula: read up, write down

MAC, DAC, and RBAC

Role-Based Access Control (RBAC): restricts access to objects on the basis of roles which may be based on, say, job functions.

RBAC can implement both DAC and MAC access control policies.

DAC: users have control over access

For instance UNIX file permissions

```
herbertb@coogee:~/doc$ ls -l tmp/
total 24
-rw-rw-r-- 1 herbertb staff 4096 Apr  1  2018 mydoc
-rwxr-x--- 1 herbertb staff 4096 Apr  2  2018 HelloWorld
-rw-r--r-- 1 herbertb staff  397 Jan 26  2018 id_rsa.pub
```

User herbertb has **r**ead/**w**rite/e**x**ecute rights on file HelloWorld.
Members of group “staff” have **r**ead/e**x**ecute rights.
Everyone else (“others”) cannot access it at all.

However, herbertb can make it available to others: `chmod o+rx HelloWorld`

4. Primitives for Isolation and Mediation

before we begin

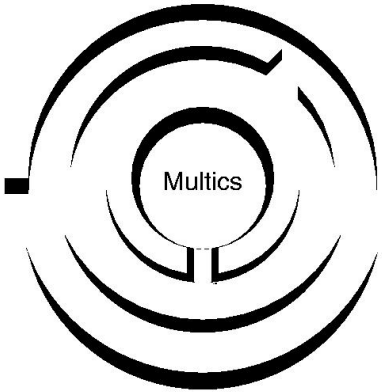
“Multics was here first”

not surprising: Jerome Saltzer (yes, he of the security principles) was one the lead engineers

In 1960s, Multics became first major OS designed with security in mind

Principles and access models such as Bell-LaPadula deeply ingrained

Many of its security innovations can still be found in the most popular OSs today



Multics

- rings of protection
- hierarchical file system with support for DAC and MAC
- MAC: direct implementation of Bell-LaPadula
- its many small software components were strongly encapsulated, accessible only via their published interfaces where mediation took place

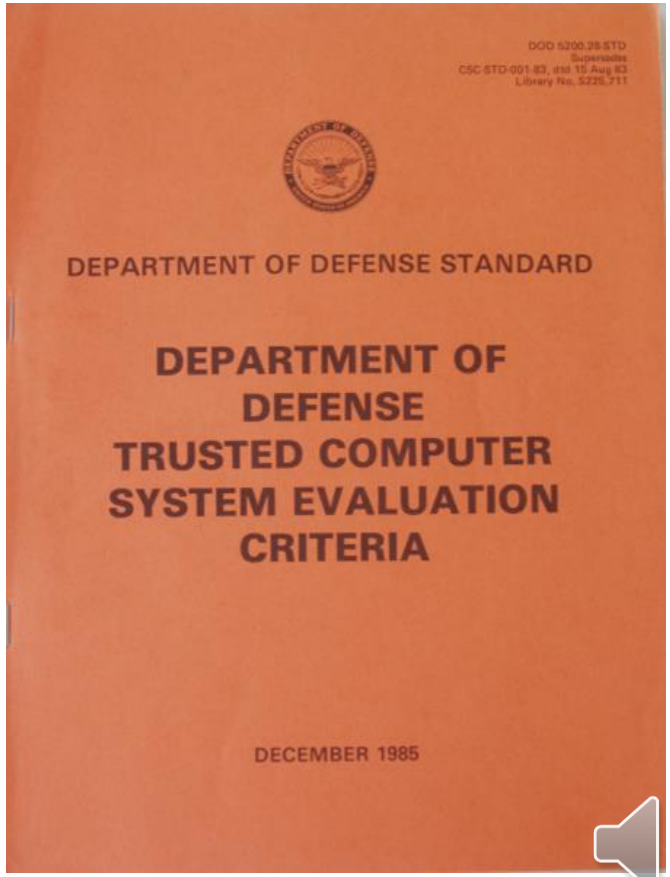
Famous “Orange Book”

Describes requirements for evaluating the security of a computer system

Strongly based on Multics

Multics was clearly very advanced, but also unwieldy ⇒ Unics ⇒ UNIX

Like all OSs, relies on small number of primitives



Anyway… primitives!

OS must authenticate users to decide whether they may access certain resources

OS also needs support for access control to objects such as files

OS needs memory protection to prevent domain from accessing another's memory

OS needs to distinguish between privileged code and non-privileged code, so that only the privileged code can configure the desired isolation at the lowest level and guarantee mediation for all operations

Authentication

See Authentication, Authorisation & Accountability (AAA) Knowledge Area

Many ways to authenticate: username/passwords, smart cards, biometrics, …

Multi-factor authentication makes it harder to bypass authentication

For each user, the OS maintains a user id for each user (and a process id for each process running on behalf of user) -- and tracks ownership and access rights

Storing the credentials is crucial

- may use TPM or VM to ensure credential store is cryptographically sealed

Access Control: Access Control Lists

Multics file system: Access Control List (ACL) for every block of data in the system

ACL: table specifying for each data block which users have what access rights

Most modern OSs have adopted ACLs, typically for the file system

ACL for a file called myscript on Linux:

```
herbertb@nordkapp:~/tmp$ getfacl myscript
# file: home/herbertb/tmp/myscript
# owner: herbertb
# group: cybok
user::rwx group::rwx other::r-x
```

Basic UNIX file permissions simple, but modern systems (such as Linux and Windows) also allow for more extensive ACLs

What about MAC today?

Took a while, but nowadays systems such Windows, BSD and Linux support it

E.g., SELinux, a set of security patches originally developed by NSA

gives users/processes a context of three strings: (username, role, domain).

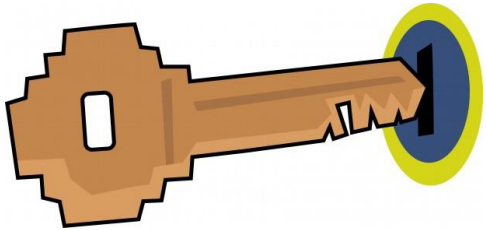
(in case you are wondering: yes, it supports RBAC)

resources also have such contexts

admins define system-wide policies for access control. E.g.:

- which contexts must processes have to access certain resources in specific ways, or
- Bell-LaPadula, or
- Biba

Capabilities (1)



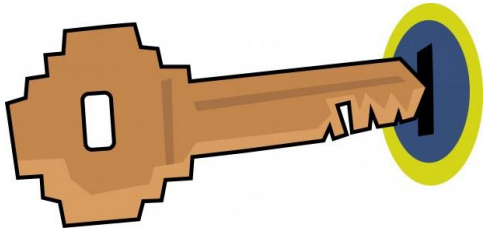
Where ACL keeps all info on whether process P can perform operation O on resource R, capabilities (Dennis, Van Horn, 1966) do not require administration

Capability itself proves that the requested access is permitted.

Not unlike a file descriptor: if you have valid fd, you can access file

A capability is a ‘token, ticket, or key that gives the possessor permission to access an entity or object in a computer system’ (contains the access rights provided)

Capabilities (2)



Possession of capability grants all rights specified in it and whenever a process wants to perform an operation on an object, it presents appropriate capability

Should be impossible to *forge* capabilities

But can be passed to other processes (maybe with subset of rights)

What if we want to restrict capabilities from spreading arbitrarily?

common solution: add some flags to indicate if copying is permitted, etc.

ACLs vs Capabilities

ACL typically based on users (“user with id *x* can read and write”),
Capability can be more fine-grained (e.g., separate cap for sending and receiving)

ACL : processes typically run with full power of user
Capability: process runs only with the caps it needs (better for POLA)

ACL: may grow very large

BUT

ACL: easy to revoke access rights
Capability: more complex to revoke (esp. if only for a subset of processes)

Many systems with capabilities built

Hardware

- MIT PDP-1 Time Sharing
- Chicago Magic Number Machine
- Cambridge CAP
- CMU Hydra
- CHERI
- …

Software

- Amoeba
- L4 and seL4 → today
- Capsicum / FreeBSD → today
- ...

Cambridge CAP computer

Note that capabilities in Linux

...are not really capabilities

(in the Dennis/Van Horn sense)

Physical access and secure deletion

At OS we may control access all we want, but will this stop physical access?

Not always. E.g., if OS deletes files on magnetic disk by removing metadata ⇒ data still on disk
Or: even when it overwrites the blocks, magnetic traces close to tracks may remain ⇒ may be read
Or: SSD ⇒ in the end, firmware decides what to (over)write and when → beyond control of OS.

Also, secure deletion may be more involved

Maybe the OS made backups → these may need secure deletion also

Securing physical access clearly requires more work (e.g., encryption on disk)

Memory protection

One security domain should not be able access memory of other domain

- Either remove all data from one domain before switching to other domain, or
- Add memory protection

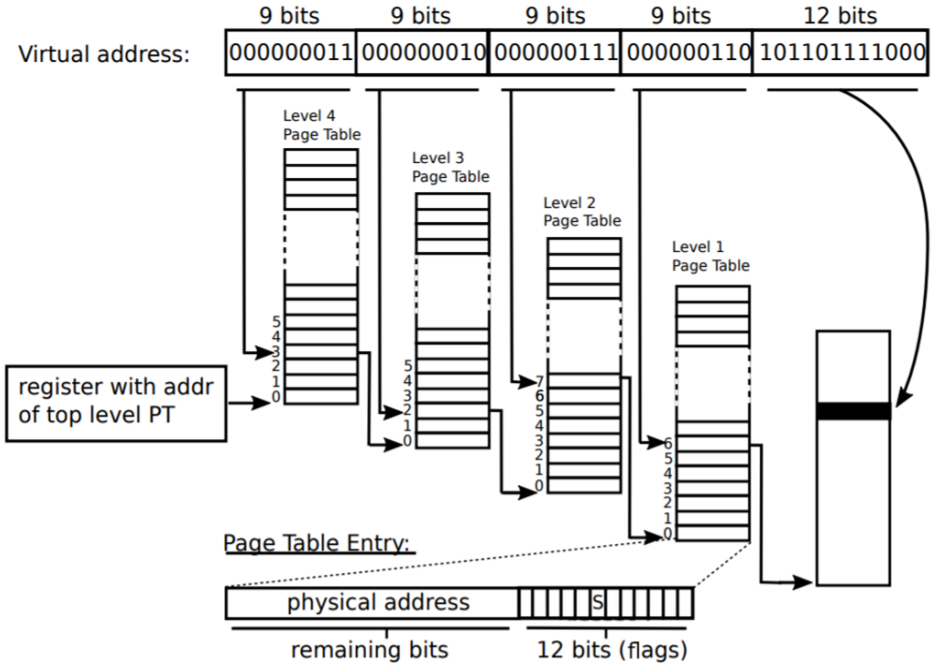
Already from Multics onward: give every process

- own processor state (e.g., registers) and own subset of memory

On context switch from P1 to P2:

- save P1 state, load P2 state
  - adjust bookkeeping determining what memory is available

What bookkeeping? Nowadays: page tables*



Address translation in modern CPUs: MMU ‘walks’ page tables to find physical address of the page. Only if page is ‘mapped’ on a process’ page tables can the process address it, provided it is present and process has correct access rights. Specifically, a user process cannot access the page for which the supervisor (S) bit is set in the page table entry.

* in general purpose computing, not necessarily present in embedded systems / IoT

Page tables

Are managed by the OS

- although the MMU may walk them
  - and translation may be sped up by the Translation Look-aside Buffer (TLB)

Give every process its own virtual address space

Besides paging, Multics supported segmentation. And so did the x86.

- no longer the hip way of doing things
  - no longer really there in x86-64, although vestiges remain
- OS/2 and AS/400 use segmentation

Paging is not the only trick up the hardware sleeve

For memory protection, modern hardware may support things like

- Memory Protection Extensions
  - to prevent buffer boundary violations
  - failure(?) [so we will not discuss any further]
- Memory Protection Keys (MPK). Old idea, but on Intel:
  - allows one to set four previously unused bits in the PTE to one of 16 *key* values
  - adds new 32b register with 2 bits for each key indicating R / W allowed for corresponding pages
  - only OS can tag PTEs with keys, but unprivileged user process may set the register
- Memory Tagging Extensions (MTE)
  - every chunk of memory gets a tag (say 4 bits)
  - pointers also get a tag (stored in unused upper bits of the pointer)
  - pointers can only access memory with same tag
  - stops most attacks targetting spatial and/or temporal memory errors

Not all processors offer such elaborate protection

For instance: IoT

Some don’t even have an MMU

Sometimes offer a (much simpler) Memory Protection Unit (MPU)

e.g., defines regions with specific access permissions and attributes.

Others protect from even more threats

For instance, a Trusted Execution Environment (TEE)

- protects when even the OS or hypervisor cannot be trusted
- E.g., Intel's SGX
  - offers TEE known as enclave
  - protects *part of application*
  - all data stored in memory is encrypted (only in unencypted form in the CPU)
  - offers hardware to do atttestation (allowing 3rd parties to verify it is running right code)
- ARM TrustZone
  - different mode: offers secure world and normal world
  - normal word cannot access secure world
  - secure world runs its own OS and applications
  - code in normal world can make calls to secure world (not unlike system calls)

Or yet others

Malicious DMA (you can’t trust anyone these days!)

- malicious hardware may use DMA to read/write sensitive data in memory that should be inaccessible
- Thunderbolt over USB-C even exposes PCIe bus to external devices

Partial remedy: IOMMU

- just like regular MMU
- translates addresses used by devices to real physical addresses
  - offers page tables
  - offers page table protection

Are these all hardware features?

No. And others are sure to emerge.

Protection rings

Among most revolutionary ideas introduced by Multics

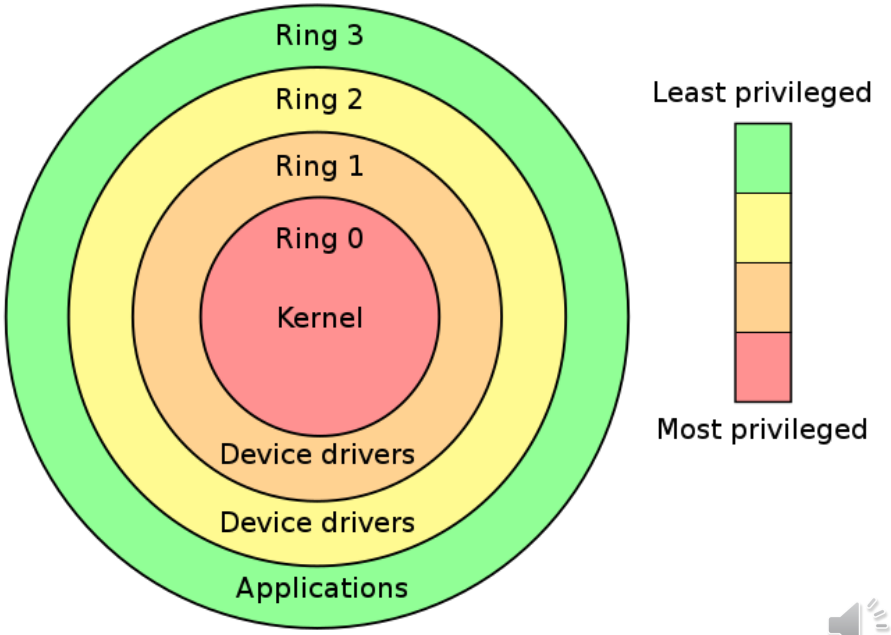
Hierarchical layering of privilege

Inner ring (ring 0) most privileged

Outer ring least privileged

Kernel executes in ring 0

User processes in outer ring (e.g., ring 3)



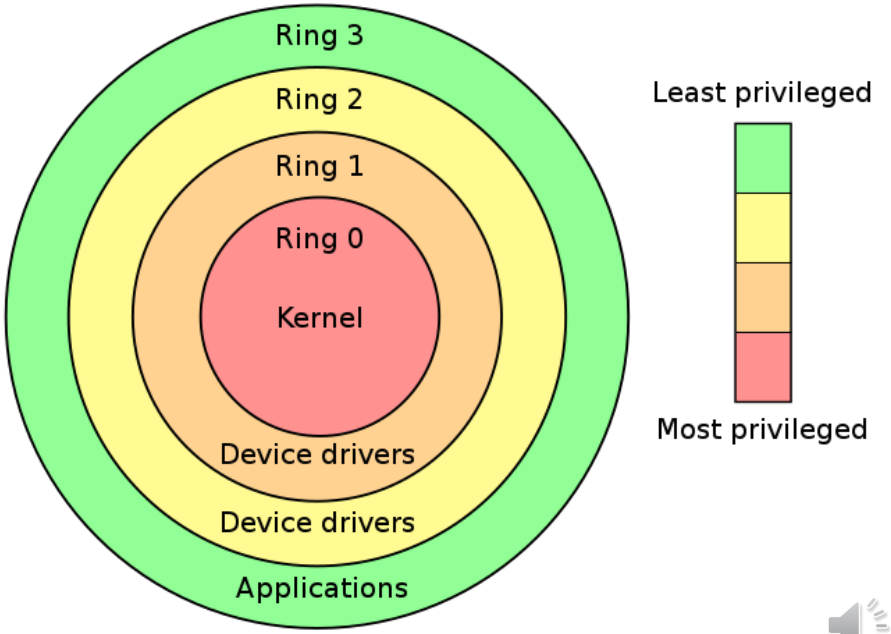
Protection rings

If less privileged code requires function by privileged code:

calls into privileged code

handled by specific code in more privileged ring

only privileged code may execute sensitive instructions (e.g., change PTE)



[BTW: we may sometimes require calls in opposite direction also: signals]

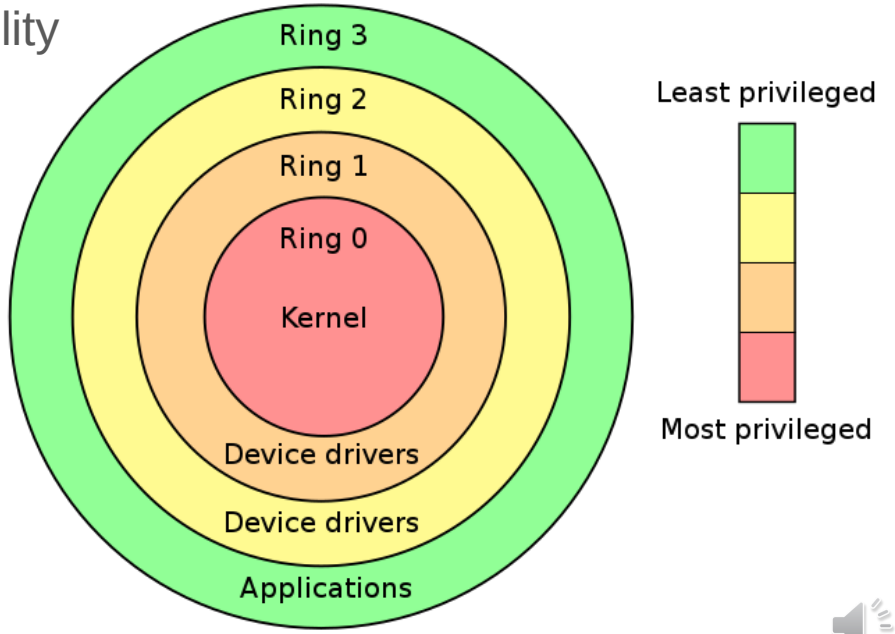
Protection ring boundaries

Interface between rings: good place to check system calls and their arguments

In capability-based systems: check capability

In other systems (e.g., MINIX 3): check if process can even make this system call

In Linux/Windows: check arguments (and maybe more)



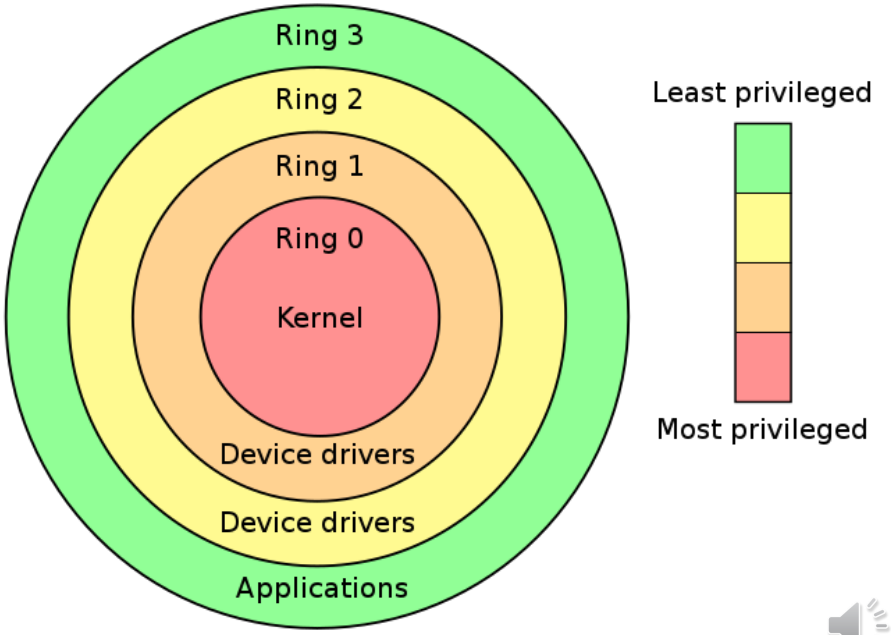
One ring to rule them all

Is Ring 0 really king? Let us consider Intel...

Nowadays hypervisors: Ring -1

System Management Mode: Ring -2

Even special processor with own OS (Management Engine): Ring -3 ?



Again, such processors are very complex

IoT processors may not have any of this

5. Operating System Hardening

Hardening

As we cannot formally verify all of the OS *yet,* we need additional hardening

1. Information hiding
2. Control flow restrictions
3. Partitioning
4. Code and data integrity checks
5. Anomaly detection

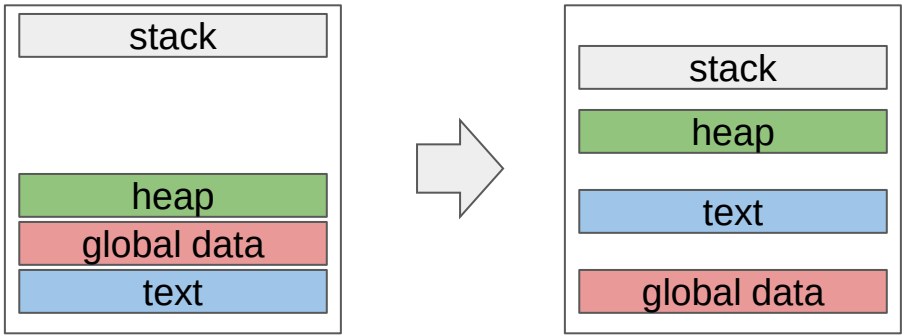
Hardening 1: Information hiding

A main line of defense: hide whatever the attackers wants to have

Address Space Layout Randomization (ASLR)

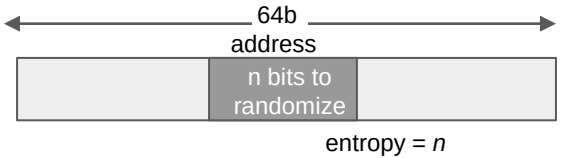
Randomize the location of relevant memory regions (text, globaldata, stack, heap)

⇒ Attackers do not know where to divert control flow or which addresses contain sensitive data, etc.
⇒ In all major OSs. First in PaX Security Patch for Linux in 2001. Other OSs followed soon(ish) after.
⇒ Initially only for applications. Now also for kernel (KASLR).



ASLR

Important questions for ASLR



⇒ How random (i.e., how many bits of entropy)

⇒ Granularity of randomization

- Only base address of region? → currently most common
- Page-level?
- Function-level?
- Basic block?
- Instructions?
- Registers?

⇒ When to randomize

- At boot time? → common (Windows)
- Load time? → common (Linux)
- Constant/periodic rerandomization?

(K)ASLR - final words

Not considered a very strong defense by itself (especially coarse-grained)

- pointers stored in memory ⇒ may leak ⇒ break randomization
- entropy sometimes too limited ⇒ brute forceable
- side channels may help attacker

Sometimes randomization is used as building block for stronger defenses

For instance, hide sensitive data at random location in huge virtual address space

- no pointers in memory point to the sensitive data (only a dedicated register)
- attackers cannot easily obtain this location

Hardening 2: Control flow restrictions

Ensure attackers cannot divert control flow “to just anywhere”

“Control-Flow Integrity: Principles, Implementations, and Applications”

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti




CFI

Simple idea:

Allow only “legitimate branches and calls”

(i.e., that follow the control flow graph)

CFI: consider following code

```
bool less_than(int x, int y);
```

```
bool greater_than(int x, int y);
```

```
bool sort(int a[], int len, comp_func_t fptr)
{
    ...
        if (fptr(a[i], a[i+i]))
            ...
    ...
}
```

```
void sort_1(int a[], int len)
{
    ...
    sort(a, len, less_than);
    ...
}
```

```
void sort_2(int a[], int len)
{
    ...
    sort(a, len, greater_than);
    ...
}
```

CFI: consider following code ⇒ restrict control flow

```
bool less_than(int x, int y);
```

```
bool greater_than(int x, int y);
```

```
bool sort(int a[], int len, comp_func_t fptr)
{
    ...
        if (fptr(a[i], a[i+i]))
            ...
    ...
}
```

CFI example: we want to insert a check that *sort* can only return to just after the call in *sort_1* and just after the call to *sort_2*

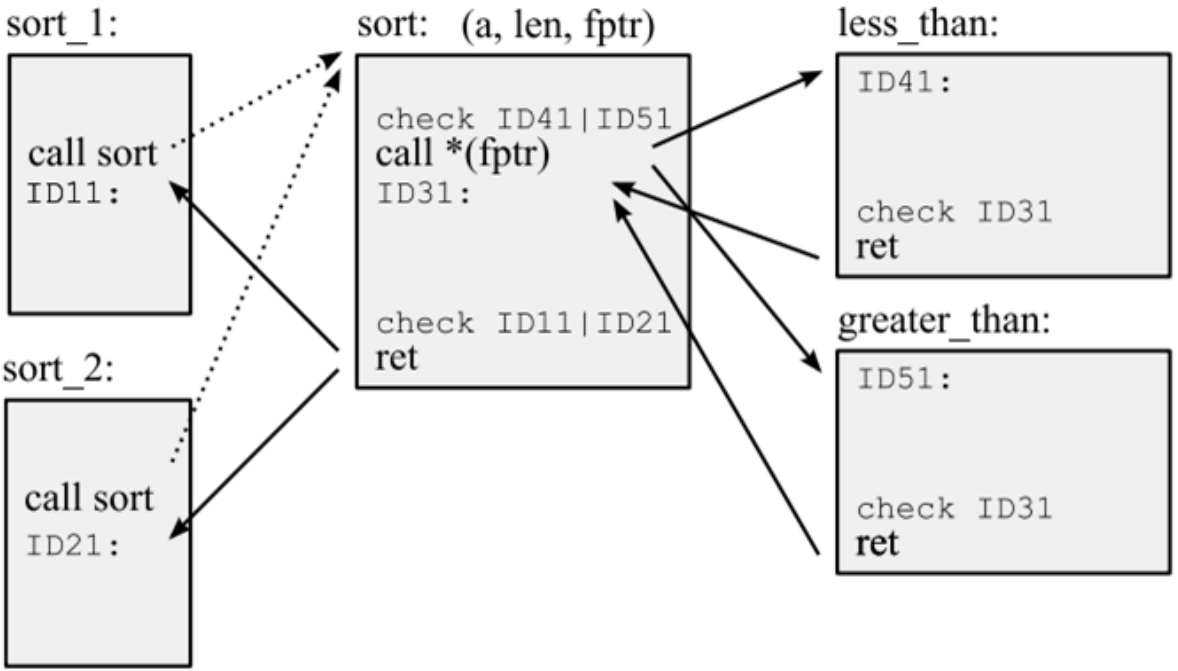
Similarly for other indirect branches (e.g., *fptr* can only land at *less_than* or *greater_than*

```
void sort_1(int a[], int len)
{
    ...
    sort(a, len, less_than);
    ...
}
```

```
void sort_2(int a[], int len)
{
    ...
    sort(a, len, greater_than);
    ...
}
```

So we give the valid targets a "label" (or ID) and check that we do not branch anywhere else

CFI


sort_1:
call sort
ID11:
sort_2:
call sort
ID21:
sort: (a, len, fptr)
check ID41|ID51
call *(fptr)
ID31:
check ID11|ID21
ret
less_than:
ID41:
check ID31
ret
greater_than:
ID51:
check ID31
ret

CFI : many variations

Granularity

E.g., option 1: allow indirect call to target small set of specific functions (expensive, but fine-grained)
option 2: allow indirect call to target any address-taken function (cheaper, but coarse-grained)

Related: context sensitivity

E.g., option 1: allow returns to return only to call site whence it came (expensive, but stronger),
option 2: allow returns to return to any call site (cheaper, but weaker)

(Option 1 can be implemented with shadow stack)

Current implementations in practice offer coarse-grained CFI

(sometimes combined with fine-grained, context sensitive shadow stack)

Data Flow Integrity (DFI)

CFI protects only control flow

With DFI, we can also restrict data flow:

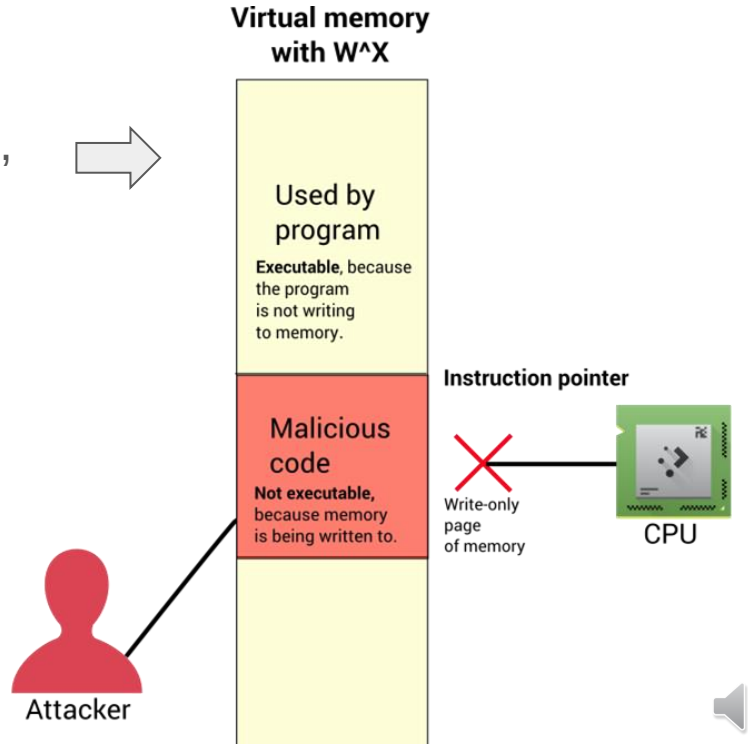
- Statically for each load instruction which store instructions may legitimately have produced the data
- Label these instructions and save these labels in a set.
- At runtime remember, for each byte in memory, the label of the last store to that location.
- When we encounter a load, check if last store to the address is in the set of legitimate stores.

At this point, DFI is not really used in practice.

Hardening 3: Partitioning

Besides partitioning in security domains (e.g., processes and kernel), we can partition further.

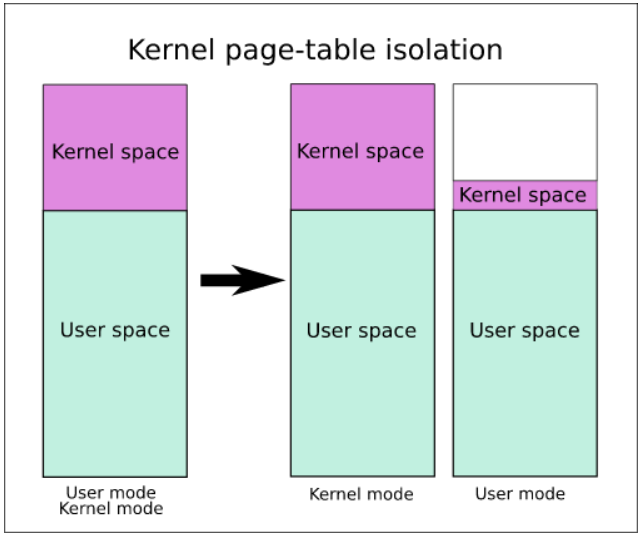
- W ^ X ⇒ pages are writable OR executable, but never both
  - prevents attackers from injecting shellcode and diverting control flow to it
- Prevent kernel from (unintentionally) accessing userspace
  - Supervisor Mode Execution Prevention (SMEP) kernel cannot execute instr in user space
  - Supervisor Mode Access Prevention (SMAP) kernel cannot access data from user space



Hardening 3: Partitioning

Besides partitioning in security domains (e.g., processes and kernel), we can partition further.

- Kernel Page Table Isolation (KPTI)
  - Page tables for user process and for kernel are (almost entirely) separate
  - Really countermeasure against Meltdown vulnerability on certain (older) processors
  - But provides SMEP/SMAP-like protection for free
- Partitioning microarchitectural state
  - e.g., partition cache by cache coloring or hardware
  - defends against variety of side-channel attacks



Hardening 4: Code and data integrity checks

Ensure that code and/or data is unmodified and provided by a trusted vendor.

- For instance, for many years Windows has embraced driver signing
- Can even “lock down” a machine to only allow trusted code to run

To do this properly, we need a secure boot process

- starts with root of trust (e.g., a special processor)
- firmware initiates sequence of stages that ends with fully booted system
  - e.g., firmware loads bootloader which then loads OS kernel which then loads drivers
  - all these steps need protection
    - UEFI Secure Boot verifies if bootloader is signed with valid key (stored in firmware)
    - Bootloader can now verify signature of OS kernel before loading it
    - kernel can verify drivers, etc.

Secure Boot (cont.)

How do we *know* system was securely booted?

Need attestation

- (possibly remote) party can check that system was loaded correctly
- typically uses special hardware such as Trusted Platform Module (TPM)
  - details beyond this presentation, see Section 5.4 of the corresponding CYBOK Document

More integrity measurement

Also possible to check integrity of code and data structures at runtime

For instance, by means of VM introspection

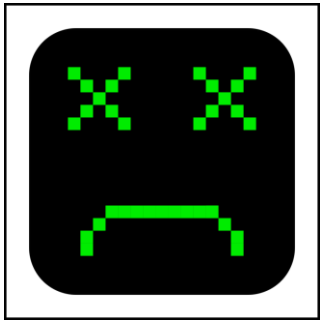
- Hypervisor checks if code still same and kernel data structures make sense
- For instance, check:
  - process list (is any rootkit trying to hide?),
  - system call table (is anybody hijacking specific system calls?),
  - interrupt vector table,
  - etc

Hardening 5: Anomaly detection

Hypervisor or OS monitors system for unusual events

For instance, crashes

Not easy to balance false positives and false negatives



6. Related areas

Many of the issues in OSs resurface in other areas

For instance, security in database systems

- Similar principles as those in operating systems
  - authentication, privileges, access control and so on: prime concerns

- Note: DB-level access control limits access to elements of DB, but not accesses at the OS level to data on disk.
  - many DBs offer transparent data encryption of sensitive table columns ⇒ key outside DB
  - extreme case: all data encrypted in DB, while only client holds keys
  - complex problem with complex solutions (e.g., homomorphic encryption)

This topic is beyond the scope of this presentation

6. Embracing Security

Adopting security solutions is problem

Increasingly advanced attacks are leading to increasingly advanced defenses.

Many of the defenses do not come from OS vendors or open source kernel teams

Surprisingly these solutions are not as eagerly adopted as you might think/want

Possible explanations

- Nothing is free: defenses slow down the system
- Often security is not easily measurable (“how much more secure will we be?”)
- Often the security vs performance trade-off is not very clear

Research focuses on defenses that significantly raise bar, with acceptable overhead

Conclusions

As the most privileged components, OS/hypervisor plays critical role in security.

Attack surface is large

Start from security principles and fundamentals

Security influenced by OS design + available security primitives and mechanisms

https://www.cybok.org/knowledgebase/