Software Security I

CSE 565: Fall 2024
Computer Security

Xiangyu Guo (xiangyug@buffalo.edu)

University at Buffalo

Disclaimer

- We don’t claim any originality of the slides. The content is developed heavily based on
  - Slides from ASU security team’s pwn.college
  - Slides from Prof Ziming Zhao’s past offering of CSE565 (https://zzm7000.github.io/teaching/2023springcse410565/index.html)
  - Slides from Prof Hongxin Hu’s past offering of CSE565

Announcement

- HW3 and Project3 **due Tue, Nov 12, 23:59 pm**.
- Bonus in-class quizzes starting from next week

Review of Last Lecture

- Privacy
  - 3PD tracking
    - 3PD Cookies
    - Fingerprinting
- Anonymity
  - Tor network
- Secure messaging app
  - E2E Encryption
  - Deniability

Today's topic

- Background
  - ELF
  - Linux process Loading & Execution

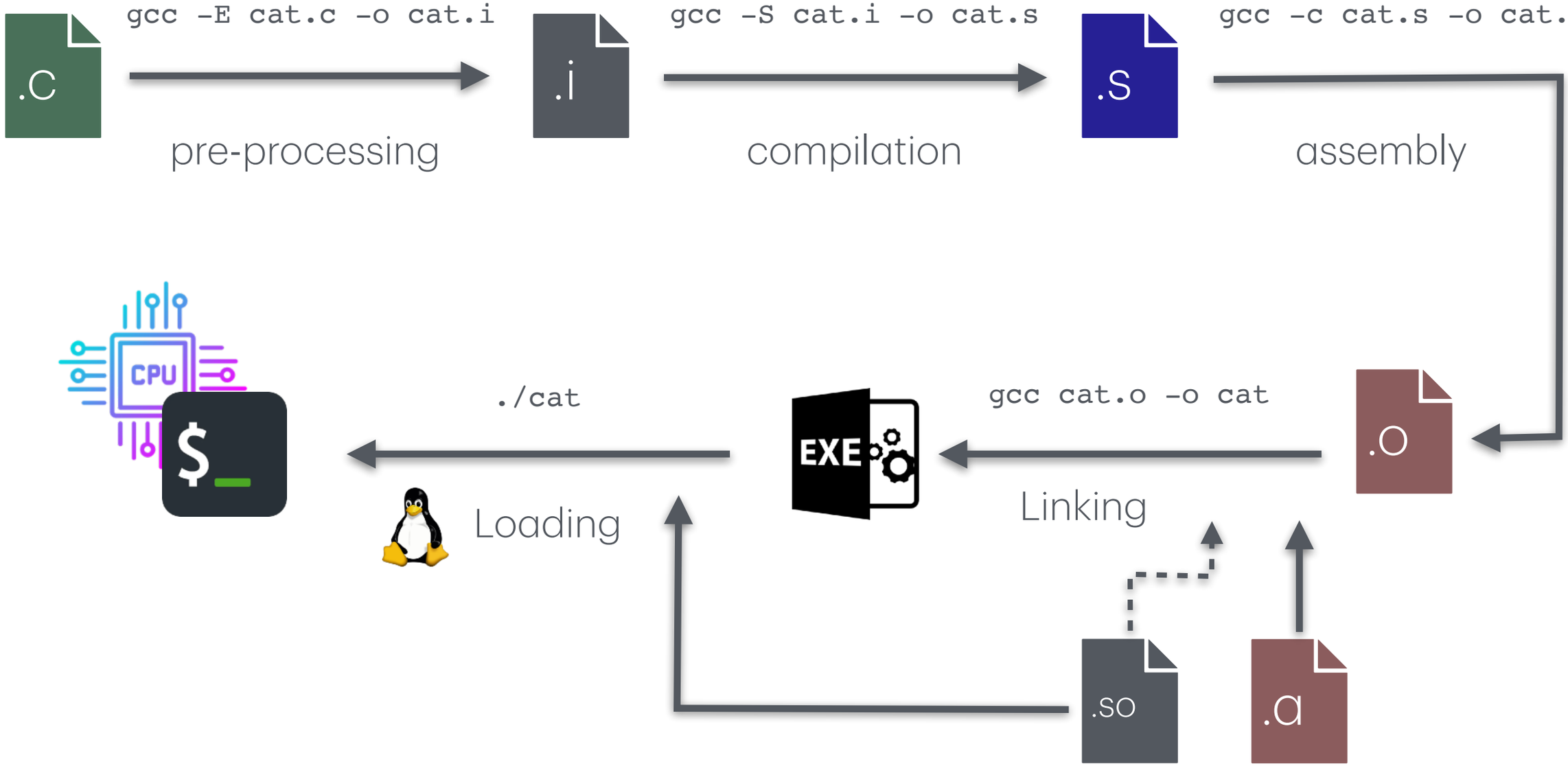
From a C program to a process

cat.c


```
// implements a cat!
int main(int argc, char **argv) {
  char buf[2048];
  int n;
  int fd = argc == 1 ? 0: open(argv[1], 0);
  while ((n = read(fd, buf, 2048)) > 0 && write(1, buf, n) > 0);
}
```

```
  $ cat cat.c
// implements a cat!
int main(int argc, char **argv) {
      char buf[2048];
      int n;
      int fd = argc == 1 ? 0: open(argv[1], 0);
      while ((n = read(fd, buf, 2048)) > 0 && write(1, buf, n) > 0);
}
```

From a C program to a process



Pre-processing

```
seed@seed-vm ~/Programs> gcc -E cat.c -o cat.i
seed@seed-vm ~/Programs> cat cat.i
# 0 "cat.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "cat.c"

int main(int argc, char **argv) {
 char buf[2048];
 int n;
 int fd = argc == 1 ? 0: open(argv[1], 0);

 while ((n = read(fd, buf, 2048)) > 0 && write(1, buf, n) > 0);
}
```

- No header/macro expansion because there was none in the source code.
- Comment is removed.

Compilation

```
seed@seed-vm ~/Programs> gcc -S cat.i -o cat.s -Wno-implicit-function-declaration
seed@seed-vm ~/Programs> head -40 cat.s
        .arch armv8-a
        .file   "cat.c"
        .text
        .align  2
        .global main
        .type   main, %function
main:
.LFB0:
        .cfi_startproc
        sub     sp, sp, #2096
        .cfi_def_cfa_offset 2096
        stp     x29, x30, [sp]
        .cfi_offset 29, -2096
        .cfi_offset 30, -2088
        mov     x29, sp
        str     w0, [sp, 28]
        str     x1, [sp, 16]
        adrp    x0, :got:__stack_chk_guard
        ldr     x0, [x0, #:got_lo12:__stack_chk_guard]
        ldr     x1, [x0]
        str     x1, [sp, 2088]
```

Assembly

```
seed@seed-vm ~/Programs> gcc -c cat.s -o cat.o
seed@seed-vm ~/Programs> strings cat.o
GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
cat.c
main
__stack_chk_guard
open
read
write
__stack_chk_fail
.symtab
.strtab
.shstrtab
.rela.text
.data
.bss
.comment
.note.GNU-stack
.rela.eh_frame
seed@seed-vm ~/Programs> cat cat.o
ELFx@@

       {
        @@@qT
T'@ Rq,TR*@D@c@T*{@ _GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0zRx
                                                              AAq
$).4cat.c$x$dmain__stack_chk_guardopenreadwrite__stack_chk_fail7
                                                             8
                                                              D


8
 ymtab.strtab.shstrtab.rela.text.data.bss.comment.note.GNU-stack.rela.eh_frame @@      &10,:<O@J@      x

        EY⏎
```

Linking

```
seed@seed-vm ~/Programs> gcc -o cat cat.o
seed@seed-vm ~/Programs> file cat.o
cat.o: ELF 64-bit LSB relocatable, ARM aarch64, version 1 (SYSV), not stripped
seed@seed-vm ~/Programs> file cat
cat: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux-aarch64.so.1, BuildID[sha1]=e1dabc44b4317a3354cf92e32db4a442f
68b20f6, for GNU/Linux 3.7.0, not stripped
```

ELF Binary Format

What is a Cat

```
> file ./cat

./cat: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/
Linux 3.7.0, BuildID[sha1]=e1dabc44b4317a3354cf92e32db4a442f68b20f6,
not stripped
```

What is a Cat

```
> file ./cat

./cat: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/
Linux 3.7.0, BuildID[sha1]=e1dabc44b4317a3354cf92e32db4a442f68b20f6,
not stripped
```

What is an ELF?

- **ELF** (Executable and Linkable Format) is a binary file format.
  - Contains the program and its data.
  - Describes how the program should be loaded (*program/segment headers*).
  - Contains metadata describing program components (*section headers*).

ELF Headers

readelf -a ./cat

```
seed@seed-vm ~/Programs> readelf -a ./cat
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 (Position-Independent Executable file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x7c0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          7240 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 20

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
```

ELF Program Headers

- Program headers specify *segments* that will be loaded to memory for executing the program. Most important entry types:
  - **INTERP:** defines the library that should be used to load this ELF into memory.
  - **LOAD:** defines a part of the file that should be loaded into memory.
- PH are *the* source of information used when loading a file. It tells the OS how to create a process image
  - Always exists for any valid ELFs.

ELF Program Headers

readelf -a ./cat

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000aa8 0x0000000000000aa8  R E    0x10000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_EH_FRAME   0x00000000000009bc 0x00000000000009bc 0x00000000000009bc
                 0x000000000000003c 0x000000000000003c  R      0x4
LOAD             0x0000000000000d60 0x0000000000010d60 0x0000000000010d60
                 0x00000000000002b0 0x00000000000002b8  RW     0x10000
GNU_RELRO        0x0000000000000d60 0x0000000000010d60 0x0000000000010d60
                 0x00000000000002a0 0x00000000000002a0  R      0x1
LOAD             0x0000000000010000 0x0000000000020000 0x0000000000020000
                 0x00000000000001c8 0x00000000000001c8  RW     0x10000
NOTE             0x0000000000010180 0x0000000000020180 0x0000000000020180
                 0x0000000000000020 0x0000000000000020  R      0x4
NOTE             0x00000000000101a0 0x00000000000201a0 0x00000000000201a0
                 0x0000000000000024 0x0000000000000024  R      0x4
INTERP           0x0000000000020000 0x0000000000030000 0x0000000000030000
                 0x000000000000001b 0x000000000000001b  R      0x1
    [Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
LOAD             0x0000000000020000 0x0000000000030000 0x0000000000030000
                 0x0000000000000020 0x0000000000000210  RW     0x10000
DYNAMIC          0x0000000000030000 0x0000000000040000 0x0000000000040000
                 0x0000000000000210 0x0000000000000210  RW     0x8
LOAD             0x0000000000030000 0x0000000000040000 0x0000000000040000
                 0x00000000000002f8 0x00000000000002f8  RW     0x10000
```

0x1b=27

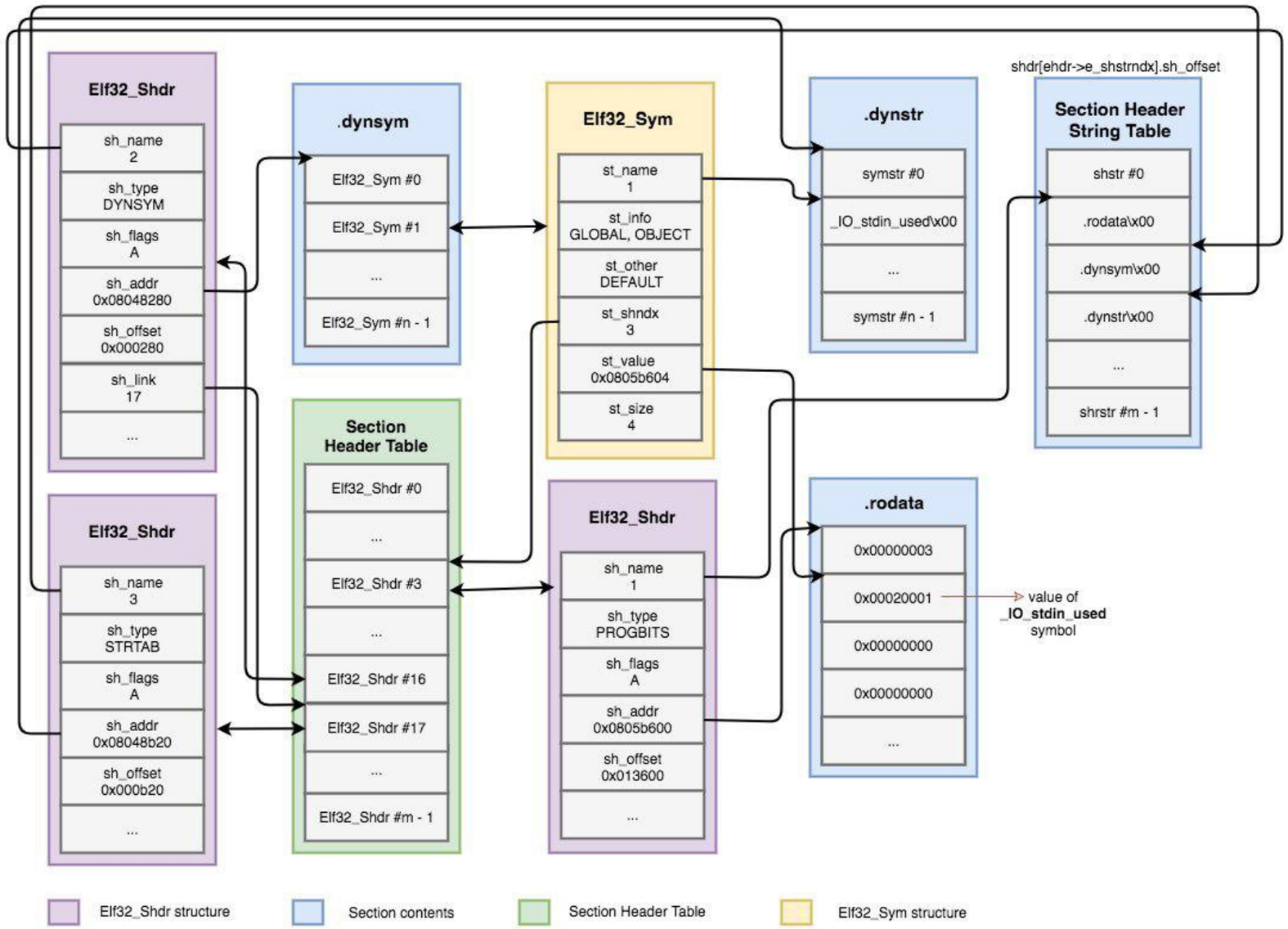
27 bytes (including the end '\0')

ELF Section Headers

- A different view of the ELF with useful information for introspection, debugging, etc.
- Important sections:
  - `.text`: the executable code of your program.
  - `.plt` and `.got`: used to resolve and dispatch library calls.
  - `.data`: used for pre-initialized *global* writable data (such as global arrays with initial values)
  - `.rodata`: used for *global* read-only data (such as string constants)
  - `.bss`: used for uninitialized *global* writable data (such as global arrays without initial values)
- Section headers are not a necessary part of the ELF: only *segments* (defined via program headers) are needed for loading and operation! Section headers are just metadata.

Symbols

Binaries (and libraries) that use dynamically loaded libraries rely on symbols (names) to find libraries, resolve function calls into those libraries, etc.



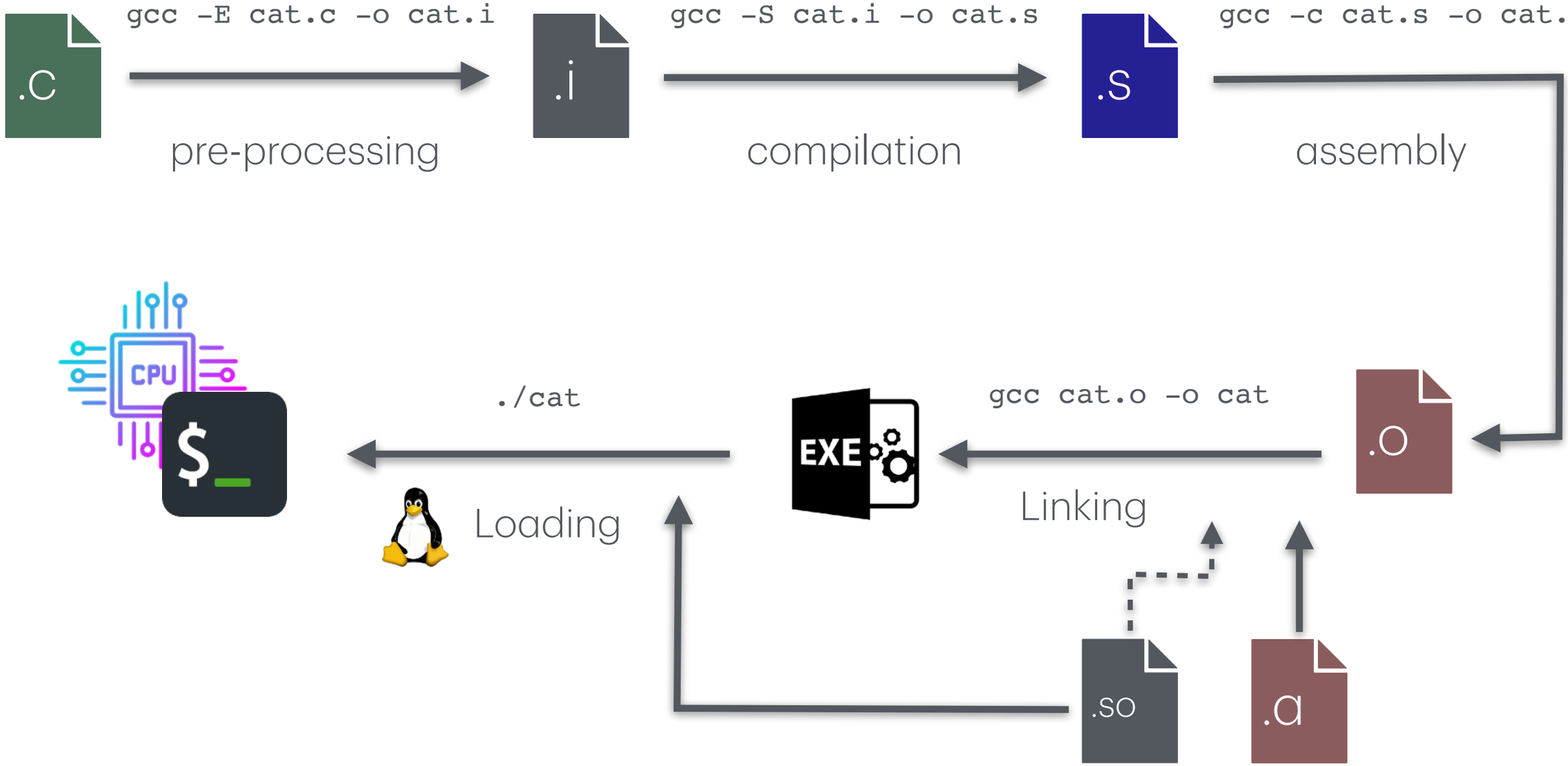
Ref: https://www.intezer.com/blog/elf/executable-linkable-format-101-part-2-symbols/

Interacting with ELF

- Several ways to dig in:
  - **gcc** to make your ELF.
  - **readelf** to parse the ELF header.
  - **objdump** to parse the ELF header and disassemble the source code.
  - **nm** to view your ELF's symbols.
  - **patchelf** to change some ELF properties.
  - **objcopy** to swap out ELF sections.
  - **strip** to remove otherwise-helpful information (such as symbols).
  - **kaitai** struct (https://ide.kaitai.io/) to look through your ELF interactively.

Linux Process Loading and Execution

From a C program to a process



The life of a cat

```
  $ cat cat.c
// implements a cat!
int main(int argc, char **argv) {
      char buf[2048];
      int n;
      int fd = argc == 1 ? 0: open(argv[1], 0);
      while ((n = read(fd, buf, 2048)) > 0 && write(1, buf, n) > 0);
}
```

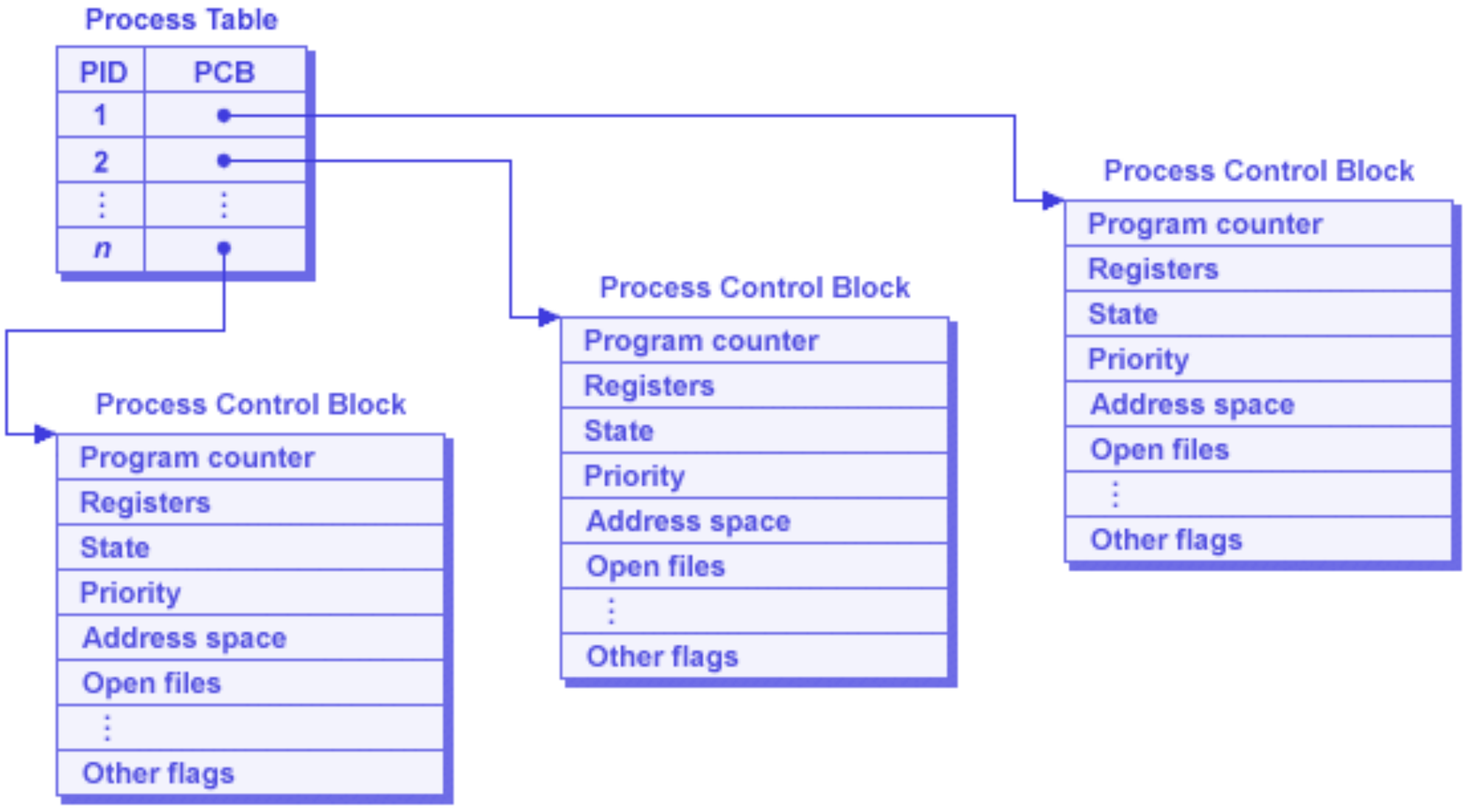
1. A process is created
2. cat is loaded
3. cat is initialized
4. cat launched
5. cat reads args and envs
6. cat does its thing
7. cat terminates

A process is created

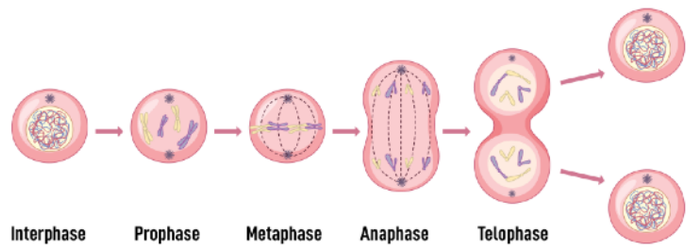
- Every Linux process has
  - Process ID (PID)
  - Current state (running, ready, waiting, etc.)
  - Program counter (indicating the next instruction to execute)
  - Memory allocation details: virtual memory space
  - I/O status and other resources (files, pipes, sockets)
  - **Security context**: effective UID/GID, saved UID/GID, capabilities

A process is created

Process Control Block (PCB)



A process is created



- In Linux, processes propagate by mitosis
  - **`fork`** and (more recently) **`clone`** are system calls that create a nearly exact copy of the calling process: a parent and a child
    - The child then uses **`execve`** syscall to replace itself with another process
- Example
  - Type **`./cat`** in bash
  - bash **`fork`**s itself into the old parent and child process
  - the child **`execves ./cat`**, becoming **`./cat`**

(Before) Loading Cat

- Before anything is loaded, the kernel does some validations
  - permission, memory requirements, etc
    - If failed, then **`execve`** fails
- Then the OS sets up a new process for the program to run in, including a virtual address space.
- Then the OS tries to find the interpreter for the program

Loading Cat

- If file starts with `#!`,
  - kernel extracts the interpreter from the rest of that line and executes this interpreter with the original file as an argument
    - E.g. shell script
- If file matches a format (magic number) in `/proc/sys/fs/binfmt_misc`
  - kernel executes the interpreter specified for that format with the original file as an argument

```
seed@seed-vm ~/Programs> cat /proc/sys/fs/binfmt_misc/python3.10
enabled
interpreter /usr/bin/python3.10
flags:
offset 0
magic 6f0d0d0a
```

Loading Cat

- If file is a **dynamically-linked ELF** (Executable and Linkable Format)
  - kernel reads the interpreter/loader from the ELF,
  - kernel loads the interpreter and the original file
    - The interpreter takes control
- If file is a **statically-linked ELF**
  - kernel loads it
- This can be recursive

Loading a Dynamically-linked Cat

The interpreter

Process loading is done by the ELF interpreter specified in the binary.

```
seed@seed-vm ~/Programs> readelf -a ./cat | grep interpreter
      [Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
```

Colloquially known as "the loader".

Can be overridden:

```
seed@seed-vm ~/Programs> /lib/ld-linux-aarch64.so.1 ./cat cat.c
int main(int argc, char **argv) {
        char buf[2048];
        int n;
        int fd = argc == 1 ? 0: open(argv[1], 0);

        while ((n = read(fd, buf, 2048)) > 0 && write(1, buf, n) > 0);
}
```

Or changed permanently:

```
seed@seed-vm ~/Programs> patchelf --set-interpreter /some/interp ./cat
seed@seed-vm ~/Programs> readelf -a ./cat | grep interpreter
      [Requesting program interpreter: /some/interp]
```

Loading a Dynamically-linked Cat

The Loading Process

- The program and its interpreter are loaded by the kernel.
- The interpreter locates the libraries.
  - **`LD_PRELOAD`** environment variable, and anything in **`/etc/ld.so.preload`**
  - **`LD_LIBRARY_PATH`** environment variable (can be set in the shell)
  - **`DT_RUNPATH`** or **`DT_RPATH`** specified in the binary file (both can be modified with **`patchelf`**)
  - system-wide configuration (**`/etc/ld.so.conf`**)
  - **`/lib`** and **`/usr/lib`**
- The interpreter loads the libraries.
  - These libraries can depend on other libraries, causing more to be loaded
  - Relocations updated

Loading a Dynamically-linked Cat

The Loading Process

```
seed@seed-vm ~/Programs> strace ./cat cat.c
execve("./cat", ["./cat", "cat.c"], 0xffffffa344c68 /* 43 vars */) = 0
brk(NULL)                               = 0xafad3790e000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7d3e7ac8000
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=58719, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 58719, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7d3e7a84000
close(3)                                = 0
openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\340u\2\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1637400, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 1805928, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7d3e78cb000
mmap(0xf7d3e78d0000, 1740392, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0xf7d3e78d0000
munmap(0xf7d3e78cb000, 20480)           = 0
munmap(0xf7d3e7a79000, 44648)           = 0
mprotect(0xf7d3e7a58000, 61440, PROT_NONE) = 0
mmap(0xf7d3e7a67000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x187000) = 0xf7d3e7a67000
mmap(0xf7d3e7a6d000, 48744, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xf7d3e7a6d000
close(3)                                = 0
```

Loading a Dynamically-linked Cat

The Loading Process

Change the `LD_LIBRARY_PATH`

```
seed@seed-vm ~/Programs> strace -E LD_LIBRARY_PATH=/some/library/path ./cat cat.c
execve("./cat", ["./cat", "cat.c"], 0xbed0ad4f46c0 /* 44 vars */) = 0
brk(NULL)                               = 0xbd6747bd2000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xfd3b6852d000
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/tls/aarch64/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such
newfstatat(AT_FDCWD, "/some/library/path/tls/aarch64/atomics", 0xfffffb52a5b0, 0) = -1 ENOENT (No such file or
openat(AT_FDCWD, "/some/library/path/tls/aarch64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or
newfstatat(AT_FDCWD, "/some/library/path/tls/aarch64", 0xfffffb52a5b0, 0) = -1 ENOENT (No such file or directo
openat(AT_FDCWD, "/some/library/path/tls/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or
newfstatat(AT_FDCWD, "/some/library/path/tls/atomics", 0xfffffb52a5b0, 0) = -1 ENOENT (No such file or directo
openat(AT_FDCWD, "/some/library/path/tls/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory
newfstatat(AT_FDCWD, "/some/library/path/tls", 0xfffffb52a5b0, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/aarch64/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file
newfstatat(AT_FDCWD, "/some/library/path/aarch64/atomics", 0xfffffb52a5b0, 0) = -1 ENOENT (No such file or dire
openat(AT_FDCWD, "/some/library/path/aarch64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or direc
newfstatat(AT_FDCWD, "/some/library/path/aarch64", 0xfffffb52a5b0, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or direc
newfstatat(AT_FDCWD, "/some/library/path/atomics", 0xfffffb52a5b0, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/library/path", 0xfffffb52a5b0, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=58719, ...}, AT_EMPTY_PATH) = 0
```

Loading a Dynamically-linked Cat

The Loading Process

Change both the **LD_LIBRARY_PATH** and **DT_RPATH**

```
seed@seed-vm ~/Programs> patchelf --set-rpath /some/rpath ./cat
seed@seed-vm ~/Programs> strace -E LD_LIBRARY_PATH=/some/library/path ./cat cat.c
execve("./cat", ["./cat", "cat.c"], 0xc21d1982a6c0 /* 44 vars */) = 0
brk(NULL)                               = 0xc244ec889000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xe9b8cb92f000
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/tls/aarch64/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/library/path/tls/aarch64/atomics", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/tls/aarch64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/library/path/tls/aarch64", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/tls/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/library/path/tls/atomics", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/tls/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/library/path/tls", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/aarch64/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/library/path/aarch64/atomics", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/aarch64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/library/path/aarch64", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/library/path/atomics", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/library/path/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/library/path", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/rpath/tls/aarch64/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/rpath/tls/aarch64/atomics", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/rpath/tls/aarch64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/rpath/tls/aarch64", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/rpath/tls/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/rpath/tls/atomics", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/rpath/tls/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/rpath/tls", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/rpath/aarch64/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/rpath/aarch64/atomics", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/rpath/aarch64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/rpath/aarch64", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/rpath/atomics/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/some/rpath/atomics", 0xffffeef87330, 0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/some/rpath/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
```

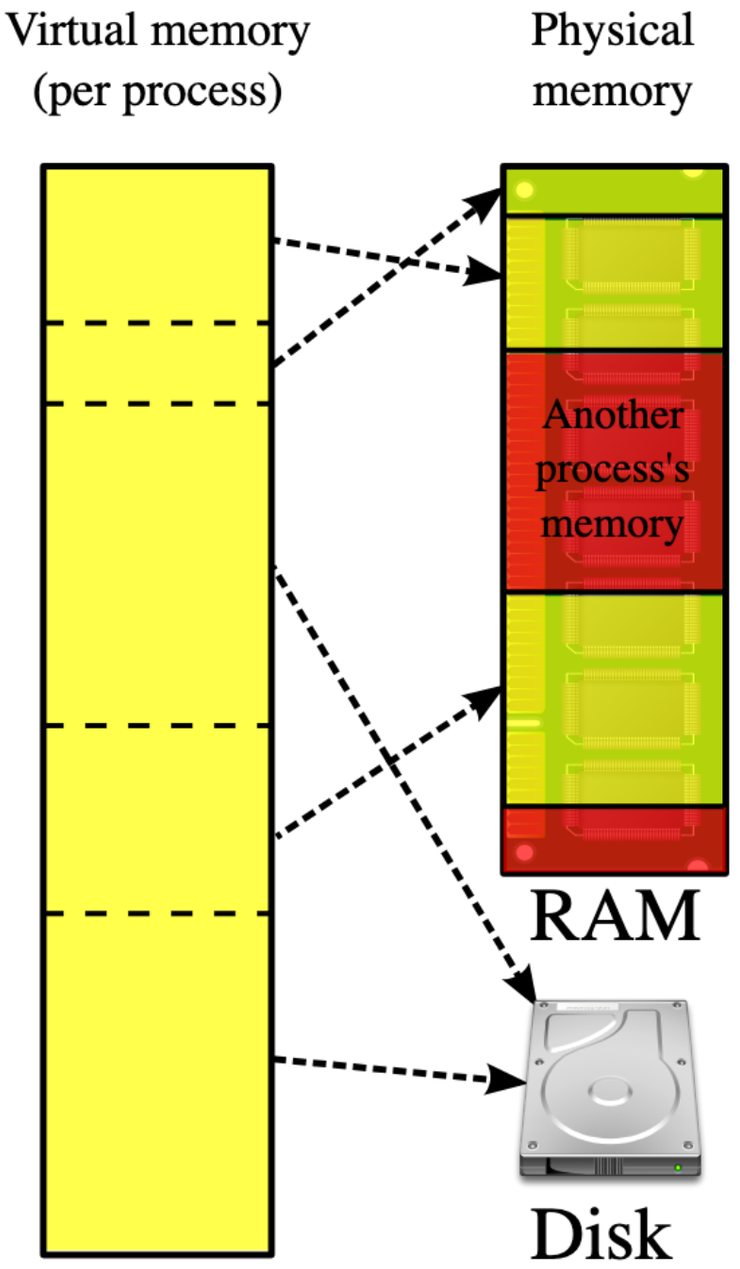
search **LD_LIBRARY_PATH**

search **RPATH**

Loading a Dynamically-linked Cat

Where is cat loaded to?

- Each Linux process has a virtual memory space. It contains:
  - The binary
  - The libraries
  - The "heap" (for dynamically allocated memory)
  - The "stack" (for function local variables)
  - Any memory specifically mapped by the program
  - Some helper regions
  - Kernel code in the "upper half" of memory (above `0x8000000000000000` on 64-bit architectures), inaccessible to the process
- Virtual memory is dedicated to your process.
  Physical memory is shared among the whole system.
- You can see this whole space by looking at `/proc/self/maps`



Loading a Dynamically-linked Cat

Where is cat loaded to?

```
./cat /proc/self/maps
```

```
seed@seed-vm ~/Programs> ./cat /proc/self/maps
b0b6c4950000-b0b6c4951000 r-xp 00000000 103:02 428496                    /home/seed/Programs/cat
b0b6c4960000-b0b6c4961000 r--p 00000000 103:02 428496                    /home/seed/Programs/cat
b0b6c4961000-b0b6c4962000 rw-p 00001000 103:02 428496                    /home/seed/Programs/cat
b0b6c4970000-b0b6c4971000 rw-p 00010000 103:02 428496                    /home/seed/Programs/cat
b0b6c4980000-b0b6c4981000 rw-p 00020000 103:02 428496                    /home/seed/Programs/cat
b0b6c4990000-b0b6c4991000 rw-p 00030000 103:02 428496                    /home/seed/Programs/cat
e5881e1f0000-e5881e378000 r-xp 00000000 103:02 1053664                   /usr/lib/aarch64-linux-gnu/libc.so.6
e5881e378000-e5881e387000 ---p 00188000 103:02 1053664                   /usr/lib/aarch64-linux-gnu/libc.so.6
e5881e387000-e5881e38b000 r--p 00187000 103:02 1053664                   /usr/lib/aarch64-linux-gnu/libc.so.6
e5881e38b000-e5881e38d000 rw-p 0018b000 103:02 1053664                   /usr/lib/aarch64-linux-gnu/libc.so.6
e5881e38d000-e5881e399000 rw-p 00000000 00:00 0
e5881e3ac000-e5881e3d7000 r-xp 00000000 103:02 1053335                   /usr/lib/aarch64-linux-gnu/ld-linux-aarch64.so.1
e5881e3df000-e5881e3e3000 rw-p 00000000 00:00 0
e5881e3e3000-e5881e3e5000 r--p 00000000 00:00 0                          [vvar]
e5881e3e5000-e5881e3e6000 r-xp 00000000 00:00 0                          [vdso]
e5881e3e6000-e5881e3e8000 r--p 0002a000 103:02 1053335                   /usr/lib/aarch64-linux-gnu/ld-linux-aarch64.so.1
e5881e3e8000-e5881e3ea000 rw-p 0002c000 103:02 1053335                   /usr/lib/aarch64-linux-gnu/ld-linux-aarch64.so.1
ffffff630d000-ffffff632e000 rw-p 00000000 00:00 0                        [stack]
```

cat's binary

libc

linker/loader

The Standard C Library

- **`libc.so`** is linked by almost every process.
- Provides functionality you take for granted:
  - **`printf()`**
  - `scanf()`
  - `socket()`
  - `atoi()`
  - `malloc()`
  - `free()`
  - …

Loading a Statically-linked Cat

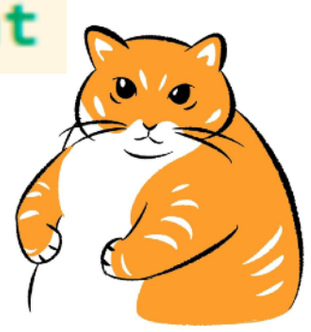
The binary is loaded by the kernel. DONE.

We can force linking a library statically:

- `gcc -static -o cat-stat cat.c`

...and we end up with a much bigger cat:

```
seed@seed-vm ~/Programs> ll cat cat-stat
-rwxrwxr-x 1 seed seed 8.9K Nov  7 23:09 cat
-rwxrwxr-x 1 seed seed 632K Nov  7 23:09 cat-stat
```



Loading a Statically-linked Cat

The binary is loaded by the kernel. DONE.

We can force linking a library statically:

- `gcc -static -o cat-stat cat.c`

...and we end up with a much bigger cat, with no **`libc.so`** linked

```
seed@seed-vm ~/Programs> ./cat-stat /proc/self/maps
00400000-0047e000 r-xp 00000000 103:02 404267                           /home/seed/Programs/cat-stat
0048d000-00491000 r--p 0007d000 103:02 404267                           /home/seed/Programs/cat-stat
00491000-00494000 rw-p 00081000 103:02 404267                           /home/seed/Programs/cat-stat
00494000-00499000 rw-p 00000000 00:00 0
282bb000-282dd000 rw-p 00000000 00:00 0                                 [heap]
fbc6b9e8e000-fbc6b9e90000 r--p 00000000 00:00 0                         [vvar]
fbc6b9e90000-fbc6b9e91000 r-xp 00000000 00:00 0                         [vdso]
ffffc166a000-ffffc168b000 rw-p 00000000 00:00 0                         [stack]
```

Launch Cat

- A normal ELF automatically calls **`__libc_start_main()`** in libc, which in turn calls the program's **`main()`** function.

```
seed@seed-vm ~/Programs> readelf -a ./cat | grep Entry
  Entry point address:               0x7c0
```

```
seed@seed-vm ~/Programs> objdump -d -j .text ./cat | head -n 40

./cat:     file format elf64-littleaarch64


Disassembly of section .text:

00000000000007c0 <.text>:
 7c0:   d503201f        nop
 7c4:   d280001d        mov     x29, #0x0                       // #0
 7c8:   d280001e        mov     x30, #0x0                       // #0
 7cc:   aa0003e5        mov     x5, x0
 7d0:   f94003e1        ldr     x1, [sp]
 7d4:   910023e2        add     x2, sp, #0x8
 7d8:   910003e6        mov     x6, sp
 7dc:   90000080        adrp    x0, 10000 <read@plt+0xf870>
 7e0:   f947f800        ldr     x0, [x0, #4080]
 7e4:   d2800003        mov     x3, #0x0                        // #0
 7e8:   d2800004        mov     x4, #0x0                        // #0
 7ec:   97ffffcd        bl      720 <__libc_start_main@plt>
 7f0:   97ffffe4        bl      780 <abort@plt>
```

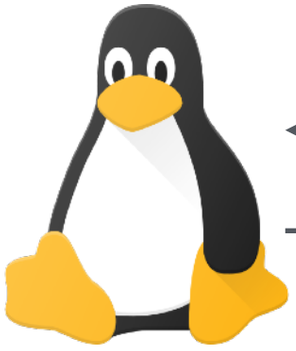
- Your code is running! Now what?

Cat reads its args and envs

- `int main(int argc, void **argv, void **envp);`
- Your process's entire input from the outside world, at launch, comprises of:
  - The loaded objects (binaries and libraries)
  - command-line arguments in **`argv`**
  - "environment" in **`envp`**
- Of course, processes need to keep interacting with the outside world.

Cat does its thing

Kernel



syscalls

signals

IPC

Other processes

Using library functions

- The binary's import symbols have to be resolved using the libraries' export symbols.
- In the past, this was an *on-demand* process and carried great peril.
- In modern times, this is all done when the binary is *loaded*, and is much safer.
- We'll explore this further in the future.

Using library functions

```
                 U write@GLIBC_2.17
seed@seed-vm ~/Programs> nm ./cat
0000000000000278 r __abi_tag
                 U abort@GLIBC_2.17
0000000000011018 B __bss_end__
0000000000011018 B _bss_end__
0000000000011010 B __bss_start
0000000000011010 B __bss_start__
00000000000007f4 t call_weak_fn
0000000000011010 b completed.0
                 w __cxa_finalize@GLIBC_2.17
0000000000011000 D __data_start
0000000000011000 W data_start
0000000000000810 t deregister_tm_clones
0000000000000880 t __do_global_dtors_aux
0000000000010d68 d __do_global_dtors_aux_fini_array_entry
0000000000011008 D __dso_handle
0000000000010d70 a _DYNAMIC
0000000000011010 D _edata
0000000000011018 B __end__
0000000000011018 B _end
00000000000009a4 T _fini
00000000000008d0 t frame_dummy
0000000000010d60 d __frame_dummy_init_array_entry
0000000000000aa4 r __FRAME_END__
0000000000010fc8 a _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
00000000000009bc r __GNU_EH_FRAME_HDR
00000000000006e8 T _init
00000000000009b8 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 U __libc_start_main@GLIBC_2.34
00000000000008d4 T main
                 U open@GLIBC_2.17
                 U read@GLIBC_2.17
0000000000000840 t register_tm_clones
                 U __stack_chk_fail@GLIBC_2.17
                 U __stack_chk_guard@GLIBC_2.17
00000000000007c0 T _start
0000000000011010 D __TMC_END__
                 U write@GLIBC_2.17
```

U: undefined symbols

will be validated by the linker and loaded by the loader

Interacting with the environment

- Almost all programs have to interact with the outside world!
- This is primarily done via system calls (**`man syscalls`**). Each system call is well-documented in section 2 of the man pages (i.e., **`man 2 open`**).
  - https://x86.syscall.sh/
- We can trace process system calls using **`strace`**.

System Calls

- System calls have very well-defined interfaces that very rarely change.
- There are over 300 system calls in Linux. Here are some examples:
  - `int open(const char *pathname, int flags)` – returns a file new file descriptor of the open file (also shows up in `/proc/self/fd!`)
  - `ssize_t read(int fd, void *buf, size_t count)` – reads data from the file descriptor
  - `ssize_t write(int fd, void *buf, size_t count)` – writes data to the file descriptor
  - `pid_t fork()` – forks off an identical child process. Returns 0 if you're the child and the PID of the child if you're the parent.
  - `int execve(const char *filename, char **argv, char **envp)` – replaces your process.
  - `pid_t wait(int *wstatus)` – wait child termination, return its PID, write its status into `*wstatus.`
  - `long syscall(long syscall, ...)` – invoke specified `syscall.`
- Typical signal combinations:
  - `fork, execve, wait` (think: a shell)
  - `open, read, write (cat)`

Signals

- System calls are a way for a process to call into the OS. What about the other way around?
- Enter: signals. Relevant system calls:
  - `sighandler_t signal(int signum, sighandler_t handler)` - register a signal handler
  - `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)` - more modern way of registering a signal handler
  - `int kill(pid_t pid, int sig)` - send a signal to a process.
- Signals pause process execution and invoke the handler.
- Handlers are functions that take one argument: the signal number.
- Without a handler for a signal, the default action is used (often, kill).
- `SIGKILL` (signal 9) and `SIGSTOP` (signal 19) cannot be handled.

Signals

Full list in section 7 of signal manual: **`man 7 signal`**

```
Signal      Standard   Action   Comment
──────────────────────────────────────────────────────────────────────
SIGABRT      P1990      Core    Abort signal from abort(3)
SIGALRM      P1990      Term    Timer signal from alarm(2)
SIGBUS       P2001      Core    Bus error (bad memory access)
SIGCHLD      P1990      Ign     Child stopped or terminated
SIGCLD         -        Ign     A synonym for SIGCHLD
SIGCONT      P1990      Cont    Continue if stopped
SIGEMT         -        Term    Emulator trap
SIGFPE       P1990      Core    Floating-point exception
SIGHUP       P1990      Term    Hangup detected on controlling terminal
                                or death of controlling process
SIGILL       P1990      Core    Illegal Instruction
SIGINFO        -                A synonym for SIGPWR
SIGINT       P1990      Term    Interrupt from keyboard
SIGIO          -        Term    I/O now possible (4.2BSD)
SIGIOT         -        Core    IOT trap. A synonym for SIGABRT
SIGKILL      P1990      Term    Kill signal
SIGLOST        -        Term    File lock lost (unused)
SIGPIPE      P1990      Term    Broken pipe: write to pipe with no
                                readers; see pipe(7)
SIGPOLL      P2001      Term    Pollable event (Sys V);
                                synonym for SIGIO
SIGPROF      P2001      Term    Profiling timer expired
SIGPWR         -        Term    Power failure (System V)
```

Shared memory

- Another way of interacting with the outside world is by sharing memory with other processes.

- Requires system calls to establish, but once established, communication happens without system calls.
  - `int shm_fd = shm_open("/myshm", O_CREAT | O_RDWR, 0666);`
  - `char *ptr = (char*) mmap(0, 1024, PROT_WRITE, MAP_SHARED, shm_fd, 0);` // map shared memory to process' own mem address
  - `strncpy(ptr, 1024, "hello");` // write data to shared memory

- Easy way: use a shared memory-mapped file in **`/dev/shm`**.
  - `FILE *fp = fopen("/dev/shm/my_shm", "r+");`
  - `fprintf(fp, "Writing to the shared mem\n");`
  - `fgets(buf, sizeof(buf), fp)`

Cat terminates

- Processes terminate by one of two ways:
  1. Receiving an unhandled signal.
  2. Calling the **`exit()`** system call: **`int exit(int status)`**
- All processes must be "reaped":
  - After termination, they will remain in a zombie state until they are **`wait()`**ed on by their parent.
  - When this happens, their exit code will be returned to the parent, and the process will be freed.
  - If their parent dies without **`wait()`**ing on them, they are re-parented to PID 1 and will stay there until they're cleaned up.

Questions?