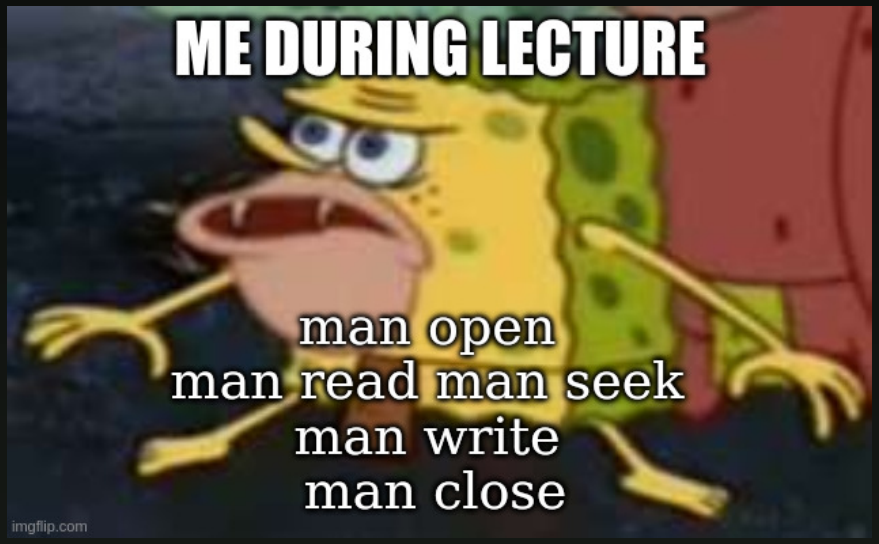
CY-4973/7790

Kernel Security: how2rootkit

Linux Filesystem Operations

- Focus: filesystem syscalls on AArch64
- I.e. "How do we list, traverse, and manage files"
- Explore kernel internals & structs
- Understand Unix permission model
- Build directory walker and file protector
- In-class assignment: recursive file scanner



Reminder: Finding Syscall values

```
cat /usr/include/asm-generic/unistd.h | grep getdents
#define __NR3_getdents64 61
__SYSCALL(__NR3_getdents64, sys_getdents64)
```

```
cat /usr/include/asm-generic/unistd.h | grep faccessat
#define __NR_faccessat 48
__SYSCALL(__NR_faccessat, sys_faccessat)
#define __NR_faccessat2 439
__SYSCALL(__NR_faccessat2, sys_faccessat2)
```

```
grep -ho "__NR_[a-zA-Z0-9_]\+\s\+[0-9]\+" /usr/include/asm-generic/unistd.h | \
  sed 's/__NR_//' | column -t
```

Syscall Interface (AArch64)

- Syscalls invoked with `svc #0`
- Registers:
  - `x8` = syscall number
  - `x0`–`x5` = up to 6 args
  - return value in `x0`
- Example:

```
int fd = syscall(SYS_openat, AT_FDCWD, "/tmp", O_RDONLY | O_DIRECTORY);
```

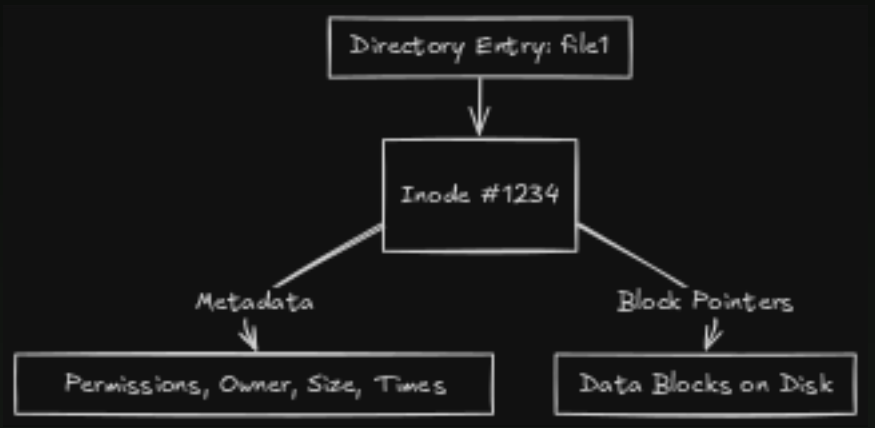
Kernel Objects Overview

```
task_struct
 ├─ files → files_struct
 │     └─ fd table → struct file *
 ├─ fs → fs_struct
 │     ├─ pwd (cwd dentry)
 │     └─ root (root dentry)
 └─ mm → mm_struct
```

Inode

Definition:

- An **inode** (index node) is a data structure on a filesystem that stores metadata about a file.
- Every file/directory has an inode (except special pseudo-filesystems like procfs).



Inode continued

- File type (regular, dir, symlink, etc.)
- Permissions and ownership (UID, GID)
- File size
- Timestamps (created, modified, accessed, changed)
- Link count (number of directory entries pointing to it)
- Pointers to data blocks on disk

Not stored in an inode:

- **Filename** (stored in the directory entry (dirent) instead!)

Linux File Types

File types shown by `ls -l` first character:

- `-` Regular file: standard data file (text, binary, executable, etc.)
- `d` Directory: contains file entries (like a folder)
- `l` Symbolic link: pointer to another file/directory (can cross filesystems)
- `b` Block device: buffered device (e.g., disk)
- `c` Character device: unbuffered device (e.g., terminal, serial port)
- `p` Named pipe (FIFO): interprocess communication
- `s` Socket: endpoint for IPC/network communication

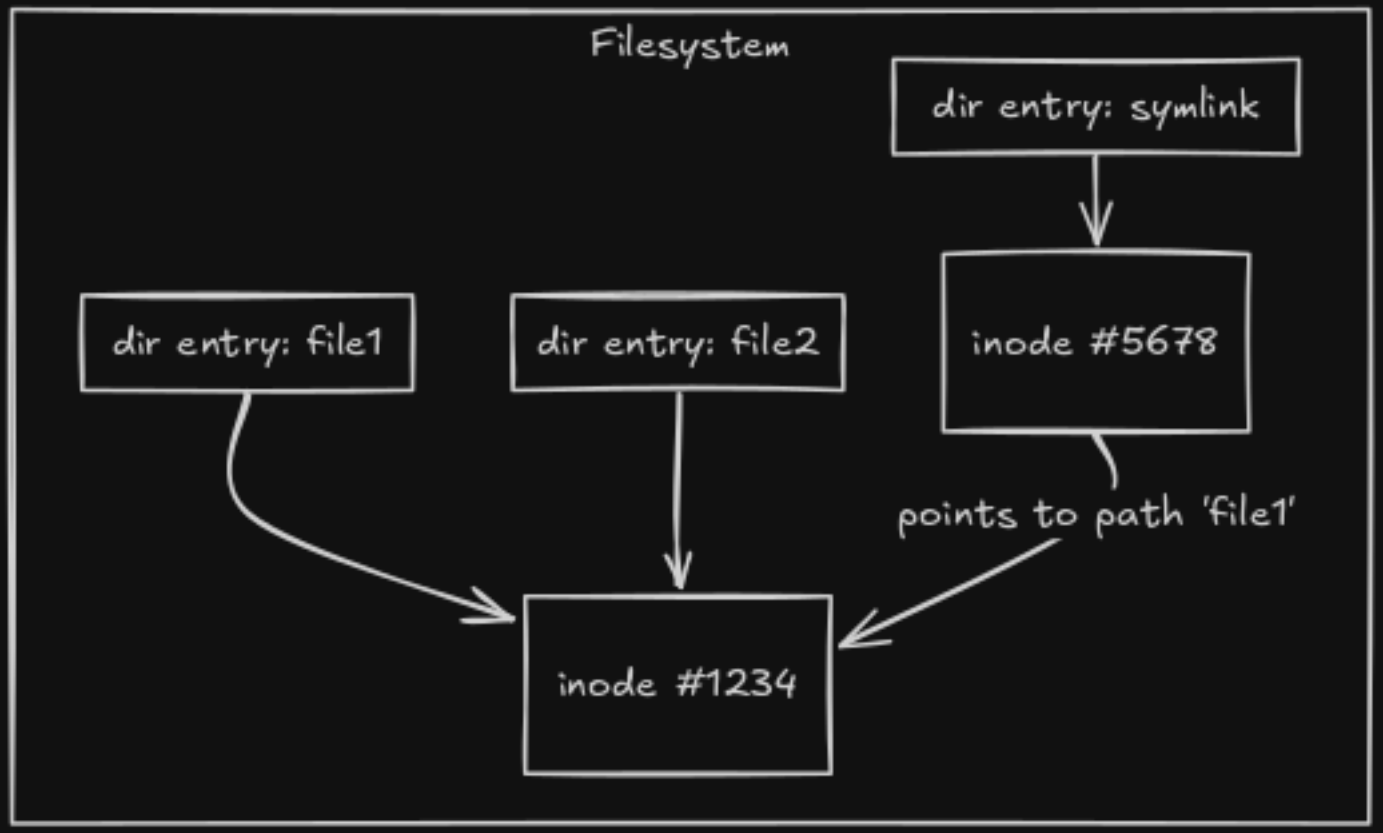
Links:

- **Hard link:** another directory entry for the same inode.
  - Same inode number, data shared.
  - Cannot span filesystems or link to directories.
- **Symbolic link (symlink):** special file that points to a pathname.
  - Can cross filesystems, can point to directories, can dangle if target removed.

Hard Link vs. Symbolic Link

Conceptual model:

- Hard link → multiple directory entries (filenames) pointing to the **same inode**.
- Symbolic link → a separate inode that stores the **path** to another file.


Filesystem
dir entry: symlink
dir entry: file1
dir entry: file2
inode #5678
points to path 'file1'
inode #1234

Unix Permission Model

File mode bits (16 bits total): (see with `ls -la`)

```
    15  14  13  12 | 11  10   9 |  8   7   6 |  5   4   3 |  2   1   0
[file type bits]   |  [special]  | [user rwx] | [group rwx]| [other rwx]
                   | suid sgid T |   r   w   x |  r   w   x  |  r   w   x
```

Common octal modes:

- `0755` = rwxr-xr-x (user: rwx, group: rx, other: rx)
- `0644` = rw-r--r-- (user: rw, group: r, other: r)
- `0700` = rwx------ (user: rwx, group: none, other: none)
- `0600` = rw------- (user: rw, group: none, other: none)

Special bits:

- `04000` = setuid runs as file owner (user id)
- `02000` = setgid runs as file group (group id)
- `01000` = sticky bit only owner can delete

man `fstatat` / `newfstatat`

- Get file metadata without opening file
- Use AT_FDCWD for relative to cwd
- AT_SYMLINK_NOFOLLOW flag to not follow symlinks
- Returns struct stat with inode metadata

Userland:

```c
#include <sys/stat.h>
#include <fcntl.h>

struct stat st;
int ret = syscall(SYS_newfstatat, AT_FDCWD, "/etc/passwd", &st, 0);
if (ret == 0) {
    printf("Size: %ld\n", st.st_size);
    printf("Mode: %o\n", st.st_mode & 0777);
    printf("UID: %d, GID: %d\n", st.st_uid, st.st_gid);
}
```

man fchmodat

- Changes file permission bits
- Use AT_FDCWD for relative to cwd
- Does not follow symlinks by default on most implementations
- Updates inode->i_mode

Userland:

```c
#include <sys/stat.h>
#include <fcntl.h>

// Make file readable/writable only by owner
int ret = syscall(SYS_fchmodat, AT_FDCWD, "/tmp/secret.txt", 0600);

// Make directory accessible only by owner
ret = syscall(SYS_fchmodat, AT_FDCWD, "/home/user/.ssh", 0700);
```

man fchownat

- Changes file ownership (UID/GID)
- Requires CAP_CHOWN capability
- AT_SYMLINK_NOFOLLOW to not follow symlinks
- Updates inode owner/group

Userland:

```c
#include <unistd.h>
#include <fcntl.h>

// Change owner to UID 1000, GID 1000
int ret = syscall(SYS_fchownat, AT_FDCWD, "/tmp/file.txt",
                  1000, 1000, 0);

// Change owner of symlink itself
ret = syscall(SYS_fchownat, AT_FDCWD, "/tmp/link",
              1000, 1000, AT_SYMLINK_NOFOLLOW);
```

man getdents64

- Reads directory entries into buffer
- Returns variable-length `struct linux_dirent64`
- Kernel iterates through dentry cache
- Much more efficient than readdir() loop

Userland:

```c
#include <syscall.h>
#include <dirent.h>

struct linux_dirent64 {
    ino64_t        d_ino;    // Inode number
    off64_t        d_off;    // Offset to next dirent
    unsigned short d_reclen; // Length of this dirent
    unsigned char  d_type;   // File type
    char           d_name[]; // Filename (null-terminated)
};

char buf[4096];
int fd = syscall(SYS_openat, AT_FDCWD, "/tmp", O_RDONLY | O_DIRECTORY);
long nread = syscall(SYS_getdents64, fd, buf, sizeof(buf));
```

man mkdirat

- Creates new directory
- Specify mode (permissions)
- AT_FDCWD for relative to cwd
- Fails if directory exists

Userland:

```c
#include <sys/stat.h>
#include <fcntl.h>

// Create directory with mode 0755
int ret = syscall(SYS_mkdirat, AT_FDCWD, "/tmp/mydir", 0755);
if (ret < 0) {
    perror("mkdirat");
}
```

man unlinkat

- Removes file or empty directory
- AT_REMOVEDIR flag required for directories
- Decrements inode link count
- File deleted when link count reaches 0 and no open fds

Userland:

```c
#include <fcntl.h>
#include <unistd.h>

// Remove file
syscall(SYS_unlinkat, AT_FDCWD, "/tmp/file.txt", 0);

// Remove directory
syscall(SYS_unlinkat, AT_FDCWD, "/tmp/mydir", AT_REMOVEDIR);
```

man getcwd

- Gets current working directory
- Kernel traverses dentry path to root
- Returns absolute path

Userland:

```
#include <unistd.h>

char buf[PATH_MAX];
long ret = syscall(SYS_getcwd, buf, sizeof(buf));
if (ret > 0) {
    printf("CWD: %s\n", buf);
}
```

Practical Example: Protecting SSH Keys

Goal: Create .ssh directory and protect private key

Steps:

1. Create `.ssh` with mode 0700
2. Create private key file
3. Set private key to 0600
4. Create public key with 0644
5. Verify with fstatat

Why?

- SSH refuses to use keys with wrong permissions
- Prevents other users from reading private keys
- Common operational security practice

Code

```c
#include <stdio.h>
#include <syscall.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define SSH_DIR ".ssh"
#define PRIV_KEY ".ssh/id_rsa"
#define PUB_KEY ".ssh/id_rsa.pub"

int protect_ssh_keys() {
    struct stat st;
    int ret;

    // Create .ssh directory with 0700
    ret = syscall(SYS_mkdirat, AT_FDCWD,
                  SSH_DIR, 0700);
    if (ret < 0 && errno != EEXIST) {
        perror("mkdirat");
        return -1;
    }
```

SSH Key Protection (cont.)

```
    // Verify directory permissions
    ret = syscall(SYS_newfstatat, AT_FDCWD, SSH_DIR, &st, 0);
    if (ret < 0) {
        perror("fstatat");
        return -1;
    }
    // Sanity check: directory should be 0700
    if ((st.st_mode & 0777) != 0700) {
        printf("Warning: %s has mode %o, fixing...\n",
               SSH_DIR, st.st_mode & 0777);
        syscall(SYS_fchmodat, AT_FDCWD, SSH_DIR, 0700);
    }
    // Create private key (or fix existing)
    int fd = syscall(SYS_openat, AT_FDCWD, PRIV_KEY,
                     O_CREAT | O_WRONLY | O_TRUNC, 0600);
    if (fd < 0) {
        perror("openat");
        return -1;
    }
    // Write dummy key data
    const char *key_data = "-----BEGIN PRIVATE KEY-----\n...\n";
    syscall(SYS_write, fd, key_data, strlen(key_data));
    syscall(SYS_close, fd);
```

SSH Key Protection (cont.)

```
    // Verify private key permissions
    ret = syscall(SYS_newfstatat, AT_FDCWD, PRIV_KEY, &st, 0);
    if (ret < 0) {
        perror("fstatat");
        return -1;
    }
    // Sanity check: private key must be 0600
    if ((st.st_mode & 0777) != 0600) {
        printf("ERROR: %s has mode %o (expected 0600)\n",
               PRIV_KEY, st.st_mode & 0777);
        syscall(SYS_fchmodat, AT_FDCWD, PRIV_KEY, 0600);
    }
    // Create public key with 0644 (world-readable)
    fd = syscall(SYS_openat, AT_FDCWD, PUB_KEY,
                 O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd < 0) {
        perror("openat");
        return -1;
    }
    const char *pub_data = "ssh-rsa AAAAB3Nza...\n";
    syscall(SYS_write, fd, pub_data, strlen(pub_data));
    syscall(SYS_close, fd);
    printf("SSH keys protected successfully!\n");
    return 0;
}
```

Directory Traversal with getdents64

Goal: Walk directory tree recursively

```c
#include <stdio.h>
#include <syscall.h>
#include <fcntl.h>
#include <dirent.h>
#include <string.h>

void list_directory(const char *path, int depth) {
    char buf[4096];
    int fd, nread, bpos;
    struct linux_dirent64 *d;

    // Open directory
    fd = syscall(SYS_openat, AT_FDCWD, path,
                 O_RDONLY | O_DIRECTORY);
    if (fd < 0) {
        perror("openat");
        return;
    }

    // Read directory entries
    while ((nread = syscall(SYS_getdents64, fd, buf, sizeof(buf))) > 0) {
        for (bpos = 0; bpos < nread;) {
            d = (struct linux_dirent64 *)(buf + bpos);
            ...
```

Directory Traversal (cont.)

```
            // Skip . and ..
            if (strcmp(d->d_name, ".") == 0 ||strcmp(d->d_name, "..") == 0) {
                bpos += d->d_reclen;
                continue;
            }
            // Print with indentation
            for (int i = 0; i < depth; i++) printf("  ");
            // Print type indicator
            char type = '?';
            if (d->d_type == DT_REG) type = 'F';
            else if (d->d_type == DT_DIR) type = 'D';
            else if (d->d_type == DT_LNK) type = 'L';
            printf("[%c] %s\n", type, d->d_name);
            // Recurse into subdirectories
            if (d->d_type == DT_DIR) {
                char subpath[4096];
                snprintf(subpath, sizeof(subpath),
                        "%s/%s", path, d->d_name);
                list_directory(subpath, depth + 1);
            }
            bpos += d->d_reclen;
        }
    }
    syscall(SYS_close, fd);
}
```

Sanity Checks for Filesystem Operations

1. **Path lengths:** Check against PATH_MAX (4096)

```
if (strlen(path) >= PATH_MAX) {
 fprintf(stderr, "Path too long\n");
 return -1;
}
```

2. **NULL pointers:** Validate all pointer arguments

```
if (!path || !buf) return -EINVAL;
```

3. **Syscall return values:** Always check for errors

```
if (fd < 0) {
 perror("syscall failed");
 return -1;
}
```

4. **Buffer bounds:** Ensure sufficient space for getdents64

```
if (d->d_reclen > sizeof(buf) - bpos) break;
```

File Type Detection with d_type

From `getdents64`, `d_type` field provides quick type check:

```
#define DT_UNKNOWN  0   // Unknown type
#define DT_FIFO     1   // Named pipe (FIFO)
#define DT_CHR      2   // Character device
#define DT_DIR      4   // Directory
#define DT_BLK      6   // Block device
#define DT_REG      8   // Regular file
#define DT_LNK      10  // Symbolic link
#define DT_SOCK     12  // Unix domain socket

// Example: filter for regular files only
if (d->d_type == DT_REG) {
    printf("Regular file: %s\n", d->d_name);
}
```

Complete Directory Walker Example

```c
//... headers as above
int walk_directory(const char *path, int max_depth, int current_depth) {
    char buf[8192];
    int fd, nread, bpos;
    struct linux_dirent64 *d;
    // Sanity check: path length
    if (strlen(path) >= PATH_MAX) {
        fprintf(stderr, "Path too long: %s\n", path);
        return -1;
    }
    // Sanity check: recursion depth
    if (current_depth > max_depth) {
        return 0;
    }

    fd = syscall(SYS_openat, AT_FDCWD, path, O_RDONLY | O_DIRECTORY);
    if (fd < 0) {
        if (errno == EACCES) {
            fprintf(stderr, "Permission denied: %s\n", path);
            return 0;
        }
        perror("openat");
        return -1;
    }
```

Complete Directory Walker (cont.)

```
    while ((nread = syscall(SYS_getdents64, fd, buf, sizeof(buf))) > 0) {
        for (bpos = 0; bpos < nread;) {
            d = (struct linux_dirent64 *)(buf + bpos);
            if (strcmp(d->d_name, ".") != 0 && strcmp(d->d_name, "..") != 0) {
                for (int i = 0; i < current_depth; i++) printf("  ");
                printf("%s", d->d_name);
                if (d->d_type == DT_DIR) {
                    printf("/\n");
                    char subpath[PATH_MAX];
                    int len = snprintf(subpath, sizeof(subpath),
                                       "%s/%s", path, d->d_name);
                    // Sanity check: path construction
                    if (len >= PATH_MAX) {
                        fprintf(stderr, "Path too long\n");
                    } else {
                        walk_directory(subpath, max_depth, current_depth + 1);
                    }
                } else { printf("\n"); }
            }
            bpos += d->d_reclen;
        }
    }
    syscall(SYS_close, fd);
    return 0;
}
```

Discussion:

- **Reconnaissance:** How might malware enumerate the filesystem?
  - Looking for SSH keys (`~/.ssh/`)
  - Finding configuration files (`/etc/`, `~/.config/`)
  - Discovering user documents
- **Privilege escalation:** How do permission bits matter?
  - SUID binaries (mode & 04000)
  - World-writable directories
  - Misconfigured sensitive files
- **Persistence:** Where do attackers hide?
  - Hidden directories (names starting with `.`)
  - Unusual permissions that prevent inspection
  - Disguising as system directories
- **Detection:** What syscalls should we monitor?
  - Unexpected `getdents64` on sensitive directories
  - `fchmodat` changing security-critical files
  - Mass file access patterns

man elf

Agenda: ELF

- Basic process creation on Linux
    - " How programs get run"
    - https://lwn.net/Articles/630727/
    - https://lwn.net/Articles/631631/
- Reading the docs
- elf file format
- Parsing
- Basic loading: Static Elf
    - Goal by end of class is to build a loader that can handle statically linked elf executable

man execve

- " execve() executes the program referred to by pathname. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments."

```
long ret = sys_execve(argv, argv, envp);
```

execve **Kernel Perspective high Level**

- Create process `task_struct`
  - allocate associated resources
    - `vm_area_struct`, `mm_struct` ...etc
- Determine how to run the program
  - Search for a handler to perform actions required to prepare binary for execution
  - `int search_binary_handler(struct linux_binprm *bprm)`
  - https://elixir.bootlin.com/linux/v3.18/source/fs/binfmt_elf.c#L2200
    - `load_elf/map_elf`
  - https://elixir.bootlin.com/linux/v3.18/source/fs/binfmt_script.c#L108

Structure of an ELF: Header

- Overview:
    - header: where is my stuff
    - Segments: the stuff in memory
    - Sections: mostly used for linking and not suuuuper relevant to loading
        - see ElfKickers sstrip: it can be removed entirely
- Elf header
- Collection of file offsets
- indicates the type of elf, where the loader can find information needed to run the elf

Terminology

- `elf file` the file as it exists on disk
- `elf buffer/elf_file_bytes` the file as it exists on disk read into memory.
    - Note I made this phrase up :)
- `elf image` the region of memory that houses the memory mapped segments of an elf

Sample Program: Static Elf

```
//cat src/static_write.c
#include "syscall_wrapper.h"
#include <unistd.h>

/*
 *
// compile
aarch64-linux-gnu-gcc src/static_write.c -static -nostdlib  -Iinc/ -o
bin/static_write
// run
qemu-aarch64 -L /usr/aarch64-linux-gnu/ bin/static_write
 * */

int _start() {
  const char msg[] = "Hello world!\n";
  sys_write(0, msg, sizeof(msg));
  sys_exit(0);
  return 0;
}
```

Sample Program: Dynamic

```c
#include <stdio.h>

extern void _start(void); // Declare _start as an external symbol

void prepend() {
  printf("woah I ran first. wonder what this means :)\n");
  _start();
}
void unused_func() {
  printf("Woah how did I get here?\n");
  return;
}

int main() {
  printf("Hello world!\n");
  return 0;
}
```

Converting Text Programs into Executables

- What is a C/C++ compiler toolchain responsible for? Converting text (code) into an application that a CPU can run!
- This is accomplished in two main steps.
- **Compiling**
- **Linking**

Compiling: Seriously Oversimplified

- Converts source code (`.c`/`.cpp`) into object files (`.o`) containing machine code.
- The compiler performs various tasks:
    - Preprocessing (macros, includes)
    - Parsing and building an Abstract Syntax Tree
    - Generating machine code using its backend for AArch64
- The result of this stage: **Object files**

Linking: Seriously Oversimplified

- Once we have compiled object files, we need to **link** them together into an ELF executable.
- The linker resolves symbols (variables/functions) and stitches everything together:
    - References to undefined symbols are replaced with their correct addresses.
- We can share code in **libraries** to avoid duplication.
- Multiple ways to link against external code:

Linking

- **Static Linking**: External code gets included directly into your final executable.
    - Useful if you are unsure a needed library will be present on the target system.
    - Results in bigger binaries.
- **Dynamic Linking**: References to external libraries are stored symbolically in the binary.
    - At runtime, the loader (`ld.so`) loads these shared libraries.
    - Reduces binary size and promotes code reuse.

Shared Libraries (.so)

- In Linux, shared libraries are typically `.so` (shared object) files.
- They contain exported, callable functions loaded at runtime.
- Examples:
    - `libc.so`: Core C library for syscalls, memory management, etc.
    - `libm.so`: Math library.
    - Other specialized libraries: `libssl.so`, `libcrypto.so`, etc.

Dynamic Linking

- **Implicit Linking**
  - Your executable's ELF headers declare which `.so` libraries it depends on.
  - At load time, if the loader can't find them, the program can't start.
- **Explicit Linking**
  - Programs can manually load libraries at runtime with something like `dlopen()`.
  - If loading fails, the program can decide how to handle that gracefully.

Static vs Dynamic ELF

- Static Elf starts out as bytes on filesystem
- The elf is arranged in its "file configuration"
    - file block size vs Page size
- The kernel "maps" the elf into memory. The layout of data is not different than it was on disk
- Execution is passed to the entry point
- Dynamic ELFs requires a dynamic loader to resolve symbols
    - i.e `printf`, handle relocations, ...etc
- `ldd` to see required libraries

`elf.h`

- Skim through elf.h
- make sure to read `man elf`

ELF Header

- Used in part for verifying the file format type
    - I.e., does the arch match, what kind of elf do we have, how is it linked
- Used to find the entrypoint
- Used to find the `Program Header` (if it exists--which usually it does)

ELF header (Ehdr)

The ELF header is described by the type Elf32_Ehdr or Elf64_Ehdr:

```
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

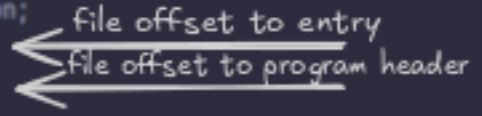
ELF Header

ELF header (Ehdr)

The ELF header is described by the type Elf32_Ehdr or Elf64_Ehdr:

```
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```



Example program

Where is the entry point?

```
readelf -h bin/ex_elf_entry
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:               0x680
  Start of program headers:          64 (bytes into file)
  Start of section headers:          7184 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 27
```

Continued

```
$ objdump -D bin/ex_elf_entry
...
0000000000000680 <_start>:
 680:   d503201f        nop
 684:   d280001d        mov     x29, #0x0                       // #0
 688:   d280001e        mov     x30, #0x0                       // #0
 68c:   aa0003e5        mov     x5, x0
 690:   f94003e1        ldr     x1, [sp]
 694:   910023e2        add     x2, sp, #0x8
 698:   910003e6        mov     x6, sp
 69c:   90000080        adrp    x0, 10000 <__FRAME_END__+0xf694>
 6a0:   f947f800        ldr     x0, [x0, #4080]
 6a4:   d2800003        mov     x3, #0x0                        // #0
 6a8:   d2800004        mov     x4, #0x0                        // #0
 6ac:   97ffffd1        bl      5f0 <__libc_start_main@plt>
 6b0:   97ffffdc        bl      620 <abort@plt>
```

Sanity check

- Build this program and use `readelf` to determine the entrypoint
- write a small c program called `patch_entry` that takes the binary name, the new entry point file offset, and creates a new binary called `<binary>.patched` that changes the entry point
    - Optionally, call the main function again after
- use `objdump -D` to find an offset to the new start. try it out with both
    - `prepend` and `unused_func`
- For parsing:
    - `unsigned long new_entry = strtoul(argv[2], NULL, 16);`

Patching the entrypoint

```
//$ aarch64-linux-gnu-objdump -D bin/ex_elf_entry | grep unused
// Update the entry point
ehdr.e_entry = new_entry;

 * $ qemu-aarch64 -L /usr/aarch64-linux-gnu/ bin/patch_entry bin/ex_elf_entry
 * `$ qemu-aarch64 -L /usr/aarch64-linux-gnu/ bin/ex_elf_entry.patched $`
 * $ chmod +x bin/ex_elf_entry.patched
 * $ qemu-aarch64 -L /usr/aarch64-linux-gnu/
 * bin/ex_elf_entry.patched
```

- Question: What do we think will happen with each one?

Program Header

- man elf

```
Program header (Phdr)
      - An executable or shared object file's program header table is an array of structures,
      - each describing a segment or other information the system  needs  to
      prepare  the  program for execution.
      - An object file segment contains one or more sections.
      - Program headers are meaningful only for executable and shared
      object files.  A file specifies its own program header size with the ELF header's e_phents
          typedef struct {
              uint32_t   p_type;
              uint32_t   p_flags;
              Elf64_Off  p_offset;
              Elf64_Addr p_vaddr;
              Elf64_Addr p_paddr;
              uint64_t   p_filesz;
              uint64_t   p_memsz;
              uint64_t   p_align;
          } Elf64_Phdr;
```

ELF Header → Program Header

- Base address: (*ELF Header)
- `e_phoff` "program header offset"
- *Program Header = Base address + elfhdr->e_phoff
- `elfhdr->e_phentsize`: byte size of program header entries
- `elfhdr->e_phnum` : number of program header entries

Program Header Types

- **PT_NULL**: Unused entry; values undefined.
- **PT_LOAD**: Loadable segment; file bytes mapped to memory.
- **PT_DYNAMIC**: Contains dynamic linking information.
- **PT_INTERP**: Specifies interpreter pathname (for executables).
- **PT_NOTE**: Holds auxiliary note information.
- **PT_SHLIB**: Reserved; semantics unspecified.
- **PT_PHDR**: Specifies program header table location & size.
- **PT_LOPROC - PT_HIPROC**: Reserved for processor-specific use.
- **PT_GNU_STACK**: GNU extension to control stack state.

Relevant Types For Static

- PT_LOAD: information about a segment
- provides information about segment start and end
    - both Physical and Virtual
    - This means we have the size of the segment
- Here: physical address is a file offset
- Provides information about memory protections
- Question: is virtual size = physical size?

readelf -l

- statically linked binary

```
readelf -l bin/static_write

Elf file type is EXEC (Executable file)
Entry point 0x4001b4
There are 3 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz              MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000002ac 0x00000000000002ac  R E    0x10000
  NOTE           0x00000000000000e8 0x00000000004000e8 0x00000000004000e8
                 0x0000000000000024 0x0000000000000024  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .text .rodata .eh_frame
   01     .note.gnu.build-id
   02
```

What if we change it?

```c
#include "syscall_wrapper.h"
#include <unistd.h>

const char msg[] = "Hello world!\n";
int _start() {

  sys_write(0, msg, sizeof(msg));
  sys_exit(0);
  return 0;
}
```

- How many segments do we expect of type PT_LOAD?
- What permissions do we expect?

readelf -l: ro

- Wait what? why is there only 1 segment?

```
readelf -l  bin/static_write_ro

Elf file type is EXEC (Executable file)
Entry point 0x4001b4
There are 3 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000294 0x0000000000000294  R E    0x10000
  NOTE           0x00000000000000e8 0x00000000004000e8 0x00000000004000e8
                 0x0000000000000024 0x0000000000000024  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .text .rodata .eh_frame
   01     .note.gnu.build-id
   02
```

Sections? Segments?

- oh --- It is making a separate section but it can store it in the same segment

```
readelf -S bin/static_write_ro
There are 9 section headers, starting at offset 0x5b8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .note.gnu.bu[...] NOTE             00000000004000e8  000000e8
       0000000000000024  0000000000000000   A       0     0     4
  [ 2] .text             PROGBITS         000000000040010c  0000010c
       00000000000000d8  0000000000000000  AX       0     0     4
  [ 3] .rodata           PROGBITS         00000000004001e8  000001e8
       000000000000000e  0000000000000000   A       0     0     8
  [ 4] .eh_frame         PROGBITS         00000000004001f8  000001f8
       000000000000009c  0000000000000000   A       0     0     8
  [ 5] .comment          PROGBITS         0000000000000000  00000294
       000000000000002b  0000000000000001  MS       0     0     1
  [ 6] .symtab           SYMTAB           0000000000000000  000002c0
       0000000000000228  0000000000000018           7    14     8
  [ 7] .strtab           STRTAB           0000000000000000  000004e8
       000000000000007c  0000000000000000           0     0     1
  [ 8] .shstrtab         STRTAB           0000000000000000  00000564
       000000000000004f  0000000000000000           0     0     1
...
```

Example 2: rw

- What about this?

```
#include "syscall_wrapper.h"
#include <unistd.h>


 char msg[] = "Hello world!\n";
int _start() {

  sys_write(0, msg, sizeof(msg));
  sys_exit(0);
  return 0;
}
```

readelf -l: rw

- because the data is not marked as read only, we need a separate segment to store it
- we don't want the segment containing our executable code to be writable

```
 readelf -l  bin/static_write_rw

Elf file type is EXEC (Executable file)
Entry point 0x400224
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000304 0x0000000000000304  R E    0x10000
  LOAD           0x0000000000000fd8 0x0000000000410fd8 0x0000000000410fd8
                 0x0000000000000028 0x0000000000000028  RW     0x10000
  NOTE           0x0000000000000158 0x0000000000400158 0x0000000000400158
                 0x0000000000000024 0x0000000000000024  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000fd8 0x0000000000410fd8 0x0000000000410fd8
                 0x0000000000000028 0x0000000000000028  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .text .rodata .eh_frame
   01     .got .got.plt
   02     .note.gnu.build-id
   03
```

Exercise

- Consider the following:

```
readelf -l bin/static_write

Elf file type is EXEC (Executable file)
Entry point 0x4001b4
There are 3 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz              MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000002ac 0x00000000000002ac  R E    0x10000
  NOTE           0x00000000000000e8 0x00000000004000e8 0x00000000004000e8
                 0x0000000000000024 0x0000000000000024  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
```

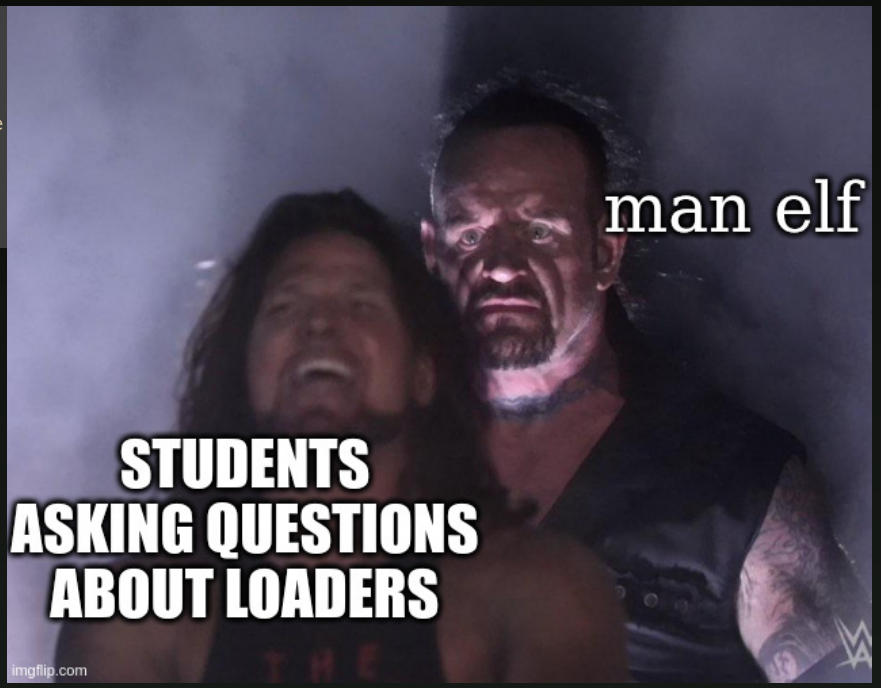
how could we modify the second segment of type NOTE to provide us with RWX scratch space?

Solution

```
typedef struct {
       uint32_t   p_type;
       uint32_t   p_flags;
       Elf64_Off  p_offset;
       Elf64_Addr p_vaddr;
       Elf64_Addr p_paddr;
       uint64_t   p_filesz;
       uint64_t   p_memsz;
       uint64_t   p_align;
   } Elf64_Phdr;
phdr->p_type = PT_LOAD;
phdr->p_memsz = 0x1000;
phdr->p_flags= <read, write, executable> // how??
```

man elf

```
man elf
....
p_flags
 This member holds a bit mask of flags relevant to the
      PF_X    An executable segment.
      PF_W    A writable segment.
      PF_R    A readable segment.
```



Solution

```
typedef struct {
       uint32_t   p_type;
       uint32_t   p_flags;
       Elf64_Off  p_offset;
       Elf64_Addr p_vaddr;
       Elf64_Addr p_paddr;
       uint64_t   p_filesz;
       uint64_t   p_memsz;
       uint64_t   p_align;
   } Elf64_Phdr;
phdr->p_type = PT_LOAD;
phdr->p_memsz = 0x1000; // make it as big as you want i guess
// but becareful if other segments come after it!
phdr->p_flags= PF_X || PF_R || PF_W; // man elf!
```

Ok what about a "normal" binary?

- Here by "normal" we mean dynamically linked
- there is nothing wrong with using statically linked binaries but usually we will want to do at least *some* dynamic linking

```c
#include <stdio.h>
extern void _start(void); // Declare _start as an external symbol

void unused_func() {
  printf("Woah how did I get here?\n");
  return;
}

int main() {
  printf("Hello world!\n");
  return 0;
}

void prepend() {
  printf("woah I ran first. wonder what this means :)\n");
  _start();
}
```

readelf -l

- man readelf

```
Elf file type is DYN (Position-Independent Executable file)
Entry point 0x680
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz              MemSiz               Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001f8 0x00000000000001f8  R      0x8
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
                 0x000000000000001b 0x000000000000001b  R      0x1
      [Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000970 0x0000000000000970  R E    0x10000
  LOAD           0x0000000000000d90 0x0000000000010d90 0x0000000000010d90
                 0x0000000000000280 0x0000000000000288  RW     0x10000
  DYNAMIC        0x0000000000000da0 0x0000000000010da0 0x0000000000010da0
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000254 0x0000000000000254 0x0000000000000254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000000000844 0x0000000000000844 0x0000000000000844
                 0x000000000000004c 0x000000000000004c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000d90 0x0000000000010d90 0x0000000000010d90
```

wtf is all of this

Dynamic linking

- Its pretty complicated and has a lot of moving parts.
- For that reason, we are going to stick with statically linked binaries and work towards loading a statically linked binary first.

Baby's First loader

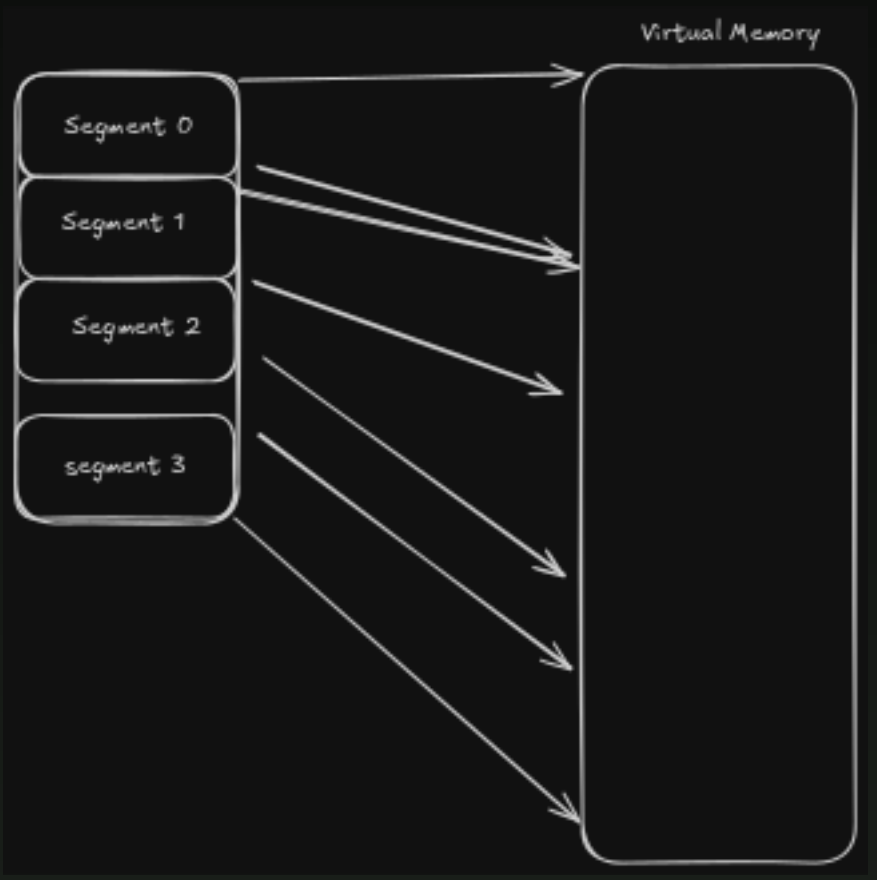
- Rather than trying to implement a fully fledged s loader, we are going to start small
- binary is an elf of type EXEC
- binary is statically linked
- All we need to do is memory map the sections, and then start running at the entry point
- For now, we will use a c program with full access to `libc`
- (no dlopen or dlsym though :)

Steps

- Open the elf file
- parse the elf header: take note of the entry point, program header size, offset ..etc
    - optionally validate that this is the file is the correct format
- Allocate a block of memory that is large enough to fit the memory mapped elf
- map the sections

Memory Mapping Segments

- Walk the program header and take note of all the entries of type `PT_LOAD`
- Keep track of the position in the block of memory used for mapping the elf
- Copy `physical_size` number of bytes starting at the proper offset to the segment
- pad with zeros if the virtual size is > physical size



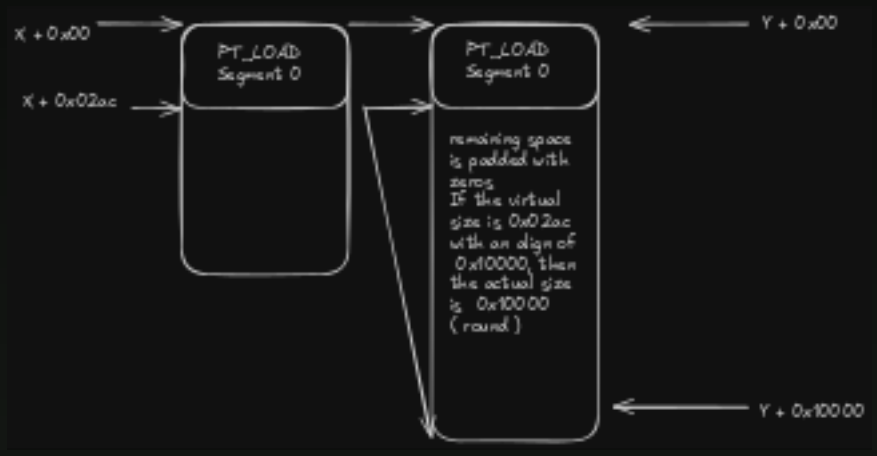
Example

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000002ac 0x00000000000002ac  R E    0x10000
```

- If the file is loaded into a buffer at virtual address X and the newly allocated buffer for the memory mapped elf is at Y
- we copy $X, \ldots, X + 0x2ac - 1$ bytes to Y
- note that the rest of that segment is padded with zeros.
- namely, $Y + 0x2ac, \ldots,$Y + 0x400000 - 1

Title

Field	Value
Type	LOAD
Offset	0x0000000000000000
VirtAddr	0x0000000000400000
PhysAddr	0x0000000000400000
FileSiz	0x00000000000002ac
MemSiz	0x00000000000002ac
Flags	R E
Align	0x10000



What about Multiple PT_LOAD?

```
 readelf -l bin/static_write_rw

Elf file type is EXEC (Executable file)
Entry point 0x400224
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz              MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000304 0x0000000000000304  R E    0x10000
  LOAD           0x0000000000000fd8 0x0000000000410fd8 0x0000000000410fd8
                 0x0000000000000028 0x0000000000000028  RW     0x10000
  NOTE           0x0000000000000158 0x0000000000400158 0x0000000000400158
                 0x0000000000000024 0x0000000000000024  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000fd8 0x0000000000410fd8 0x0000000000410fd8
                 0x0000000000000028 0x0000000000000028  R      0x1
```

Relevant PT_LOAD

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flags	Align
LOAD	0x00	0x400000	0x400000	0x304	0x304	R E	0x10000
LOAD	0xfd8	0x410fd8	0x410fd8	0x28	0x28	RW	0x10000

Calculating Virtual Ismage Size

1. **Find the Lowest Virtual Address**:
    - Look at all **PT_LOAD** segments and find the **smallest** p_vaddr (Virtual Address).
2. **Find the Highest Virtual Address + Size**:
    - For each **PT_LOAD** segment, compute: Segment End=pvaddr+pmemsz Segment End = p_vaddr + p_memszSegment End=pvaddr+pmemsz
    - Take the **largest** of these segment ends.
3. **Compute Total Virtual Memory Required**:
    - **Total Memory = (Highest Segment End - Lowest Virtual Address)**
    - This gives the total amount of memory that must be mapped in the virtual address space.

Step 1

- First Segment (R E) starts at `0x00400000`
- Second Segment (RW) starts at `0x00410fd8`
- Lowest Virtual Address = `0x00400000`

Step 2

- First Segment End: `0x00400000 + 0x00000304 = 0x00400304`
- Second Segment End: `0x00410fd8 + 0x00000028 = 0x00411000`
- Since alignment is `0x10000`, the second segment may actually be mapped at `0x00410000` (page-aligned), so the end extends to `0x00411000`

Step 3

- **Total Virtual Memory Required**:
- 0x00411000 − 0x00400000 = 0x00011000
- Hence we need `0x00011000`
- **Decimal Equivalent:** 69632 bytes (68 KB)

Expected mapping

- Assume we are mapped at address=0x400000 size=0x11000
- segment 0: vaddr=0x400000 offset=0x0 filesz=0x304 memsz=0x304 flags=5
- segment 1: vaddr=0x410fd8 offset=0xfd8 filesz=0x28 memsz=0x28 flags=6

Practical Tips: Page Alignment

```
/* Page size is typically 4KB (0x1000 bytes) on most systems */

#define PAGE_SIZE 0x1000
#define PAGE_MASK (~(PAGE_SIZE - 1))
#define PAGE_ALIGN_DOWN(addr) ((addr) & PAGE_MASK)
#define PAGE_ALIGN_UP(addr) (((addr) + PAGE_SIZE - 1) & PAGE_MASK)
#define PAGE_OFFSET(addr) ((addr) & (PAGE_SIZE - 1))
```

Practical Tips Allocating Memory

- `man mmap`
- map anonymous
- Ideally map as `RW` then fix up permissions later
- To start, it is ok to map as `RWX`
- when setting segment permissions, round the page down.