Discussion 4

Harishankar Vishwanathan

Overview

- Statically linked programs on linux
- HW3

Statically linked programs

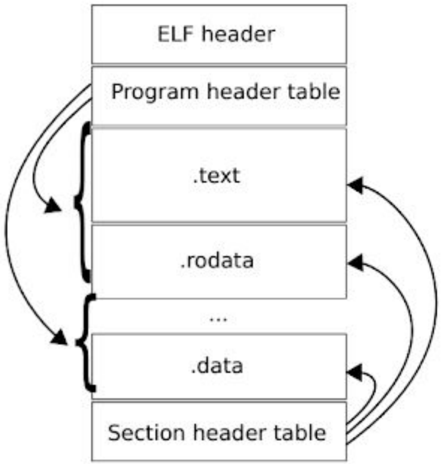
- Program doesn't require any shared objects to run (not even `libc`)
  - In reality, this isn't true, programs almost always will require shared objects

Program execution

- Always begins in the kernel
- A process will call `exec`, which ends up issuing `sys_execve` system call
- The kernel supports different binary formats for an executable
  - It will try every format one-by-one until it succeeds.
- We will focus on `ELF`
  - `load_elf_binary`

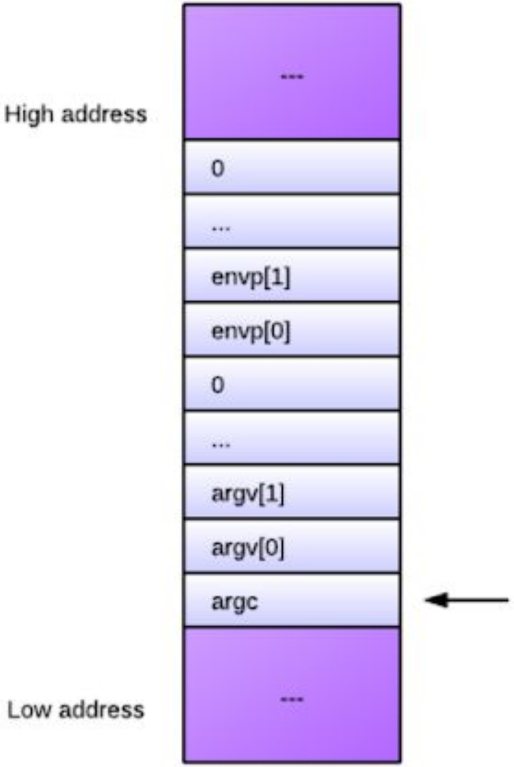
Reading ELF

- The ELF binary is composed of:
  - ELF header
  - Program Header Table
  - Section Header Table
- ELF is mainly composed of segments and sections
- Segments:
  - Portions of the binary that are actually loaded into memory at runtime (composed of one or more sections)
- Sections:
  - Actual program code and data that is available in memory when a program runs
  - Metadata about other sections used only in the linking process



Reading ELF

- Kernel reads ELF
  - Maps programs segments into memory according to the PHT.
- Passes execution
  - Directly modifying EIP register, to the entry address read from ELF header of the program (`e_entry`)
  - Arguments are passed to the program on the stack



Program entry point

How is the ELF binary created?

- Several object files are linked into an executable ELF binary by using the linker `ld`
- **The linker looks for a special symbol called `_start` in one of the object files**
- Sets the entry point to the address of that symbol

Demo : Program entry point in .asm

```
; file: nasm_rc.asm
section    .text
    ; The _start symbol must be declared for the linker (ld)
    global _start

_start:
    ; Execute sys_exit call. Argument: status -> ebx
    mov     eax, 1  ; system call 1: sys_exit
    mov     ebx, 42 ; pass arguments to sys_exit
    int     0x80    ; call into kernel
```

- This simple program simply returns 42.

Demo : Program entry point in .asm

- Compile with
```
nasm -f elf32 nasm_rc.asm -o nasm_rc.o
```
- Link with
```
ld -m elf_i386 -o nasm_rc nasm_rc.o
```
- Read the elf header, what entry point do you see?
```
readelf -h nasm_rc
```
- Is it the same as the address of `_start`?
```
objdump -M intel -d nasm_rc
```
- Run the program and check its exit code:
```
$ ./c_rc
$ echo $? # return code of a program
42
```

Demo : Program entry point in .c

```
/* file c_rc.c */

int main() {
    return 42;
}
```

Demo : Program entry point in .c

- Use the -c flag in gcc to compile but not link.

```
gcc -c -m32 -fno-pic -fno-builtin -fno-omit-frame-pointer c_rc.c
```

- When we ask gcc to just compile (but not link), the generated object file object file is minimal:

```
objdump -M intel -d c_rc.o
```

- Now, link with

```
ld -m elf_i386 -o c_rc c_rc.o
```

- Does the linker give you a warning?
- What happens if you try to execute **`c_rc`**?
- How is c_rc different from c_rc.o?
    - `objdump -M intel -d c_rc.o`

Demo : Program entry point in .c

- Since we just compiled (did not link) our minimal C file, the linker cannot find the entry point (it tries to guess).
- The linker clearly needs some additional object files, where it will find the entry point i.e. the `_start` symbol.

- We can specify the additional object files to the linker, but since we don't know what those files exactly are, we will use gcc's help.
- Gcc when invoked without the -c flag, will invoke the linker with the required object files

Demo : Program entry point in .c

- Since this talk is about how statically linked programs work, we will specify the `-static` flag to gcc (the flag is passed on to the linker internally, since we are invoking gcc and the linker together).
  `gcc -o c_rc -static c_rc.o`

- Run the program and check its exit code:
  `$ ./c_rc`

- To see a list of all the libraries the gcc passed on to the linker:
  `gcc Wl,-verbose -m32 -o c_rc -static c_rc.o`

- We see that there are some additional object files needed (the whole static libc, `libc.a`).

Demo : Program entry point in .c

- C code does not live in a vacuum!
- It has several dependant objects, most notably libc.

Exercise

- Our code was clearly linked correctly and it worked: it should have the `_start` symbol.
- Check out if it does in `objdump -d c_rc | less`, (search for `_start`) and if the address matches the entry point in `readelf -h c_rc`
- The code at the symbol `_start` should call a libc related function: `__libc_start_main`.
- What are the arguments to `__libc_start_main`?
    - One of them should be the address of our `main` function!

__libc_start_main

```
int __libc_start_main(
         /* Pointer to the program's main function */
         (int (*main) (int, char**, char**),
         /* argc and argv */
         int argc, char **argv,
         /* Pointers to initialization and finalization functions */
         __typeof (main) init, void (*fini) (void),
         /* Finalization function for the dynamic linker */
         void (*rtld_fini) (void),
         /* End of stack */
         void* stack_end)
```

__libc_start_main

- What does it do?
  - Figure out where the environment variables are on the stack
  - Initialize libc
  - Call the program initialization function through the passed pointer (`init`)
  - Register the program finalization function (`fini`) for execution on exit
  - Call `main(argc, argv, envp)`
  - Call `exit` with the result of main as the exit code

HW3: Reading elf

- `readelf` is your friend
  - Use it to figure out what exactly the binary of an executable contains, and at offset locations in that binary
- You have to, finally, load the ELF binary called `elf` into memory and run it.
- Two structs are provided to you, `read` into these structs and fill them up.
  - `elfhdr`
  - `Proghdr`
- `lseek, open, read, mmap` are the syscall wrappers you would need to work with.

Mmap demo

References

https://eli.thegreenplace.net/2012/08/13/how-statically-linked-programs-run-on-linux