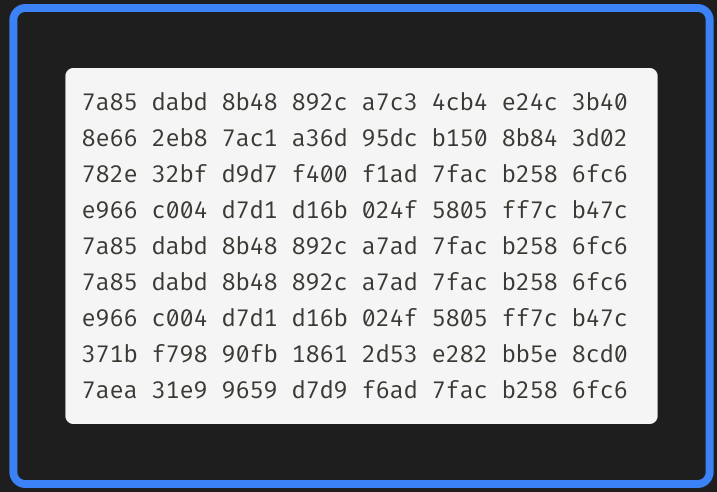
Binary Analysis I

Identifying system components and properties

Start Week 3 →



Review

Any questions on what we covered last week?

Building an Executable

Executables are usually built using a compiler like `gcc`, `clang`, `msvc`, `rustc`, etc., which generate machine code from source code.

C program source.

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

Compile the source code.

```
gcc main.c -o my_program
```

Hex output

```
$ xxd main | head
00000000: cafe babe 0000 0002 0100 0007 0000 0003  ................
00000010: 0000 4000 0000 bbf0 0000 000e 0100 000c  ..@.............
00000020: 8000 0002 0001 0000 0001 5c00 0000 000e  ..........\.....
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Executable Format Header

Different platforms support different executable file formats. These formats begin with a header that specifies the entry point, function locations, and other metadata.

Linux (ELF)

Magic: `7F 45 4C 46`

```c
#include <stdint.h>
#define EI_NIDENT 16

typedef struct {
  unsigned char   e_ident[EI_NIDENT];
  uint16_t        e_type;
  uint16_t        e_machine;
  uint32_t        e_version;
  uint64_t        e_entry;
  uint64_t        e_phoff;
  uint64_t        e_shoff;
  uint32_t        e_flags;
  uint16_t        e_ehsize;
```

macOS (Mach-O)

Magic: `FE ED FA CF`

```c
#include <stdint.h>

typedef struct {
  uint32_t magic;
  uint32_t cputype;
  uint32_t cpusubtype;
  uint32_t filetype;
  uint32_t ncmds;
  uint32_t sizeofcmds;
  uint32_t flags;
  uint32_t reserved;
} mach_header_64;
```

Windows (PE)

Magic: `4D 5A` + `50 45 00 00`

```c
typedef struct {
  WORD  e_magic;
  WORD  e_cblp;
  WORD  e_cp;
  WORD  e_crlc;
  WORD  e_cparhdr;
  WORD  e_minalloc;
  WORD  e_maxalloc;
  WORD  e_ss;
  WORD  e_sp;
  WORD  e_csum;
  WORD  e_ip;
  WORD  e_cs;
```

Components of an Executable

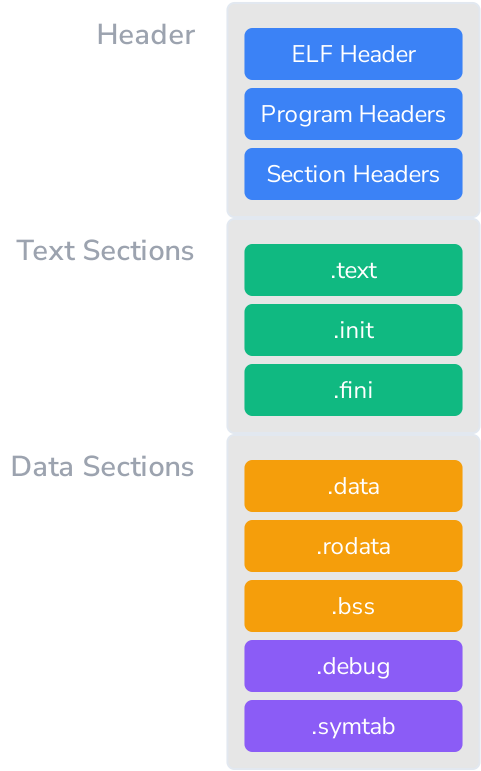
Executable files are organized into distinct sections that serve different purposes during execution.

Headers:

- **ELF Header** - Entry point and other metadata
- **Section Headers** - Table describing all sections
- **Program Headers** - Memory layout and loading

Key Sections:

- **.text** - Contains the actual executable code
- **.data** - Initialized global and static variables
- **.rodata** - Read-only data (strings, constants)
- **.bss** - Uninitialized global and static variables
- **.debug** - Debugging information for developers
- **.symtab** - Symbol table for linking and debugging



Executable Format Header

Which platform and format is this? What's the entry point?

```c
#include <stdint.h>
#define EI_NIDENT 16

typedef struct {
  unsigned char   e_ident[EI_NIDENT];
  uint16_t        e_type;
  uint16_t        e_machine;
  uint32_t        e_version;
  uint64_t        e_entry;
  uint64_t        e_phoff;
  uint64_t        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;
} Elf64_Ehdr;
```

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF....
00000010: 0300 3e00 0100 0000 2015 0000 0000 0000  ..>.....
00000020: 4000 0000 0000 0000 1862 0000 0000 0000  @.......
00000030: 0000 0000 4000 3800 0d00 4000 1f00 1e00  ....@.8...
00000040: 0600 0000 0400 0000 4000 0000 0000 0000  ........
00000050: 4000 0000 0000 0000 4000 0000 0000 0000  @.......
00000060: d802 0000 0000 0000 d802 0000 0000 0000  ........
00000070: 0800 0000 0000 0000 0300 0000 0400 0000  ........
00000080: 1803 0000 0000 0000 1803 0000 0000 0000  ........
00000090: 1803 0000 0000 0000 1c00 0000 0000 0000  ........
000000a0: 1c00 0000 0000 0000 0100 0000 0000 0000  ........
000000b0: 0100 0000 0400 0000 0000 0000 0000 0000  ........
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ........
000000d0: 0010 0000 0000 0000 0010 0000 0000 0000  ........
000000e0: 0010 0000 0000 0000 0100 0000 0500 0000  ........
000000f0: 0010 0000 0000 0000 0010 0000 0000 0000  ........
00000100: 0010 0000 0000 0000 5d1e 0000 0000 0000  ........
00000110: 5d1e 0000 0000 0000 0010 0000 0000 0000  ]...
00000120: 0100 0000 0400 0000 0030 0000 0000 0000  ........
00000130: 0030 0000 0000 0000 0030 0000 0000 0000  .0......
```

Loading an Executable

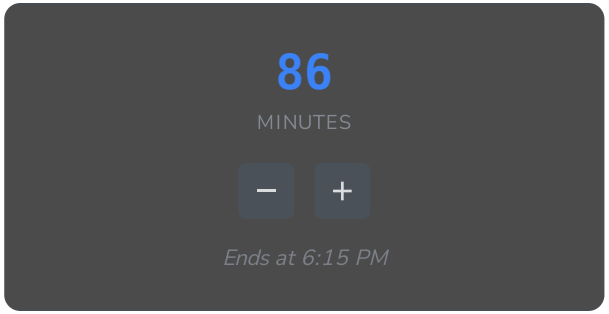
Loading Process:

- **File Validation** - OS checks magic bytes and file format
- **Permission Check** - Verifies user has execute permissions
- **Memory Allocation** - Allocates virtual memory space for the process
- **Header Parsing** - Reads ELF header to find entry point and program headers
- **Segment Mapping** - Maps file segments to memory using program headers
- **Dynamic Linking** - Loads shared libraries if needed
- **Stack Setup** - Creates initial stack frame and environment
- **Process Creation** - Establishes process control block (PCB)
- **Jump to Entry** - Transfers control to the program's entry point

Lab 1

Initial binary triage.

https://hacs408e.umd.edu/labs/week-03/lab-1/



Assembly

Low-level programming language that is translated into the the architecture's byte-code. Here we will use the x86_64 architecture.

Common Architectures

- Intel x86/x86_64: Used by most desktop computers
- ARM: Used by mobile devices, IOT devices, and (increasingly) desktop computers
- MIPS: Used mostly by IOT devices

Moving Data Around

Move immediates and other data between registers and the stack.

```
mov eax, 0×01        ;put 1 into eax
mov [eax], 0×01      ;put 1 into the address in eax
mov eax, [esi]       ;put contents of address (esi)
```

Push data on onto the stack.

```
push eax             ;put contents of eax on top of stack
push 0×01            ;put 1 on top of stack
                     ; and inc the stack pointer

pop eax              ;put contents top of the stack into eax,
                     ; and dec the stack pointer
```

Displacements.

```
; [base + index*size + offset]
; size can only be 1,2,4,8
mov eax, [arr + esi*4 + 0]
```

Moving Data Around

Load effective address does not access memory with the displacement operator! It only does the pointer arithmetic with no dereference!

```
lea eax, ecx   ;invalid
lea eax, [ecx] ;valid, equivalent to mov eax, ecx

lea eax, [ecx + edx]   ;mov eax, ecx + edx*1 (implicit 1)
lea eax, [ecx + edx*3] ;invalid, valid numbers are 1,2,4,8

lea eax, [eax + edx*4] ;can be thought of as
                       ; eax = (DWORD *)eax[edx] why?
```

Directing Execution

Jumping to an address or label.

```
jmp addr     ;addr could be a register
             ; with an address or a label
this_is_a_label:
call addr    ; functions are just labels (addresses), with a calling convention
ret          ; using the correct calling convention,
             ;  ret returns from the called function
syscall      ; more commonly seen as 'int' for interrupt
```

Jump if flags are set.

```
je addr  ; or jz  -- if zero flag is set
jg addr  ; or ja  -- if greater - signed or unsigned
jl addr  ; or jb  -- if less    - signed or unsigned
jge addr ;        -- if greater or equal to
```

Common flags.

```
carry    -- used to indicate carry in arithmetic operation
zero     -- if a value is zero or comparison equals 0
sign     -- if negative
overflow -- if overflow occurred
```

Intel vs AT&T

A given assembly architecture may have multiple syntaxes: Intel and AT&T syntaxes are the moxt common. We'll focus on the former but we want you to know the other exists. The main difference is in the source and destination operand order.

Intel

```
mov edi, esi
```

AT&T

```
mov %esi, %edi
```

Walkthrough

We'll walk through how C code corresponds to the assembly.

```
foo:
    push ebp
    mov ebp, esp

    sub esp, 8            ;make room
    mov ecx, [ebp + 4]    ;get c
    mov edx, [ebp + 8]    ;get d

    mov eax, 4      ;sizeof(int)
    mul edx         ;sizeof(int)*d

    push eax        ;arg to malloc
    call malloc
    add esp, 4      ;clean up arg
    mov [esp], eax ;store in yeet

    add esp, 8      ;clean up locals
    pop ebp
    ret

main:
    push ebp
    mov ebp, esp
```

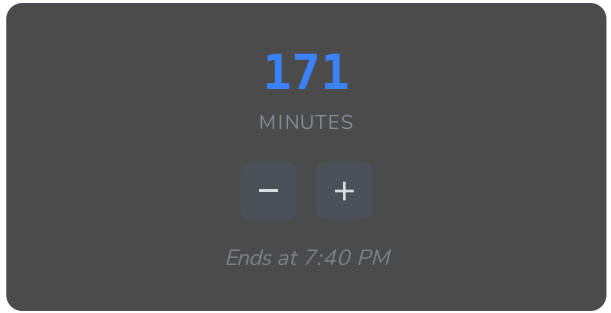
```
int *foo(c,d) {
    char e;
    void *yeet = malloc(sizeof(c)*d);
    /* Stop! */
    return (int *)yeet;
}

int main(int argc, char *argv[]) {
    int a = 5;
    int b = 7;
    char *bar = foo(a,b);

    return 0;
}
```

Lab 2

https://hacs408e.umd.edu/schedule/week-03/lab-2/



Homework

Second homework is due in two week.