Executable Formats

ELF, PE, MACHO

In this class

- Brief overview of what executable formats do. (15 minutes)
- We will break down the ELF file format. (1 hour)
- Students will implement a program to parse ELF files. (1 hour)

- Bonus Points: Students can programmatically rearrange, change, or resize portions of an ELF file on disk while maintaining runtime behavior.

If you're going for Bonus Points

You may want to write code as we go.

I have conveniently placed `/usr/include/elf.h` on everyone's linux machines. This file includes all of the structs and defines we will cover.

Executable Formats

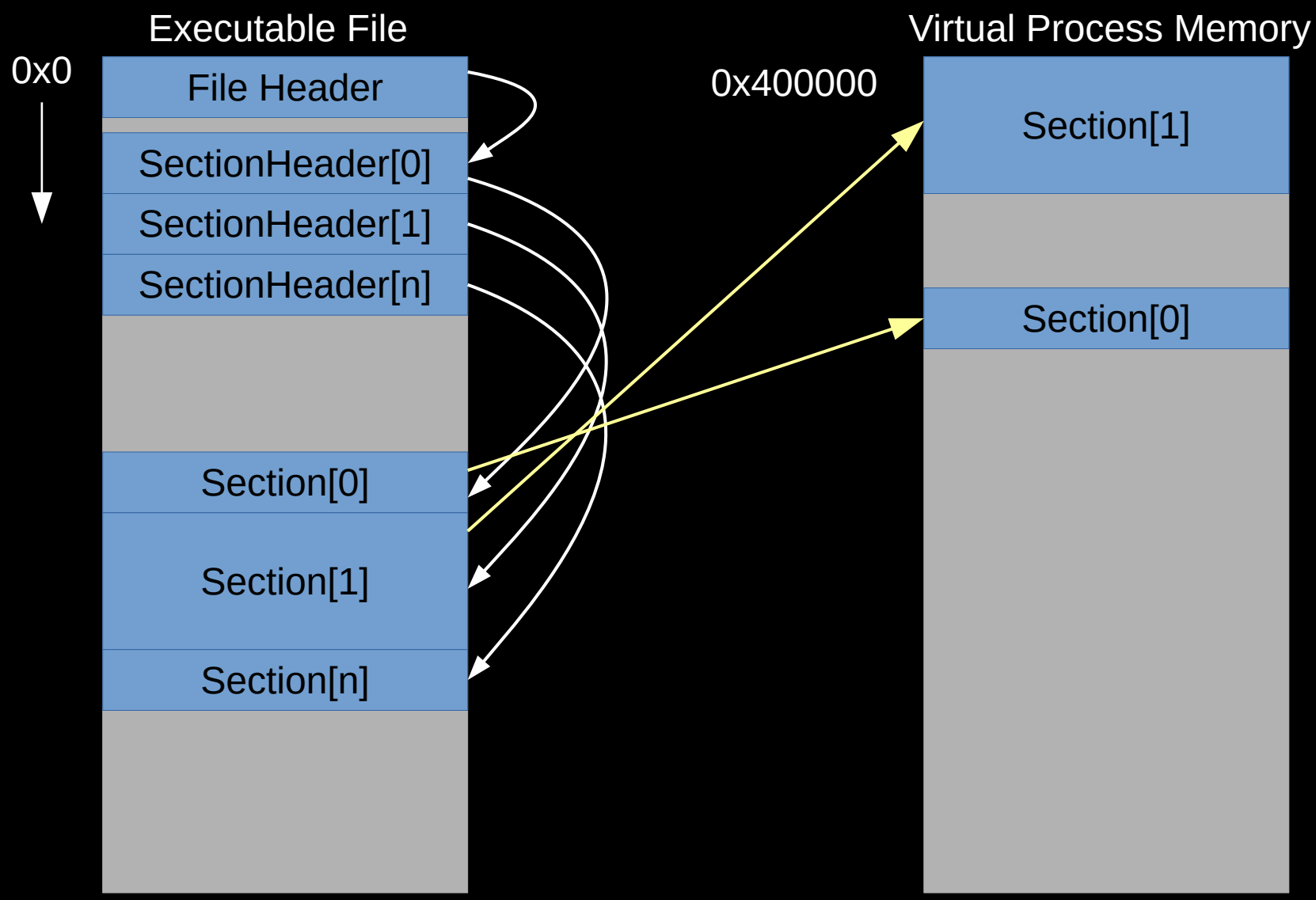
Executable formats contain the information required by the operating system to load a program into memory.

The big three are PE (Windows), MACHO (Mac OS X/iOS), and ELF (Everything Else, including Linux)

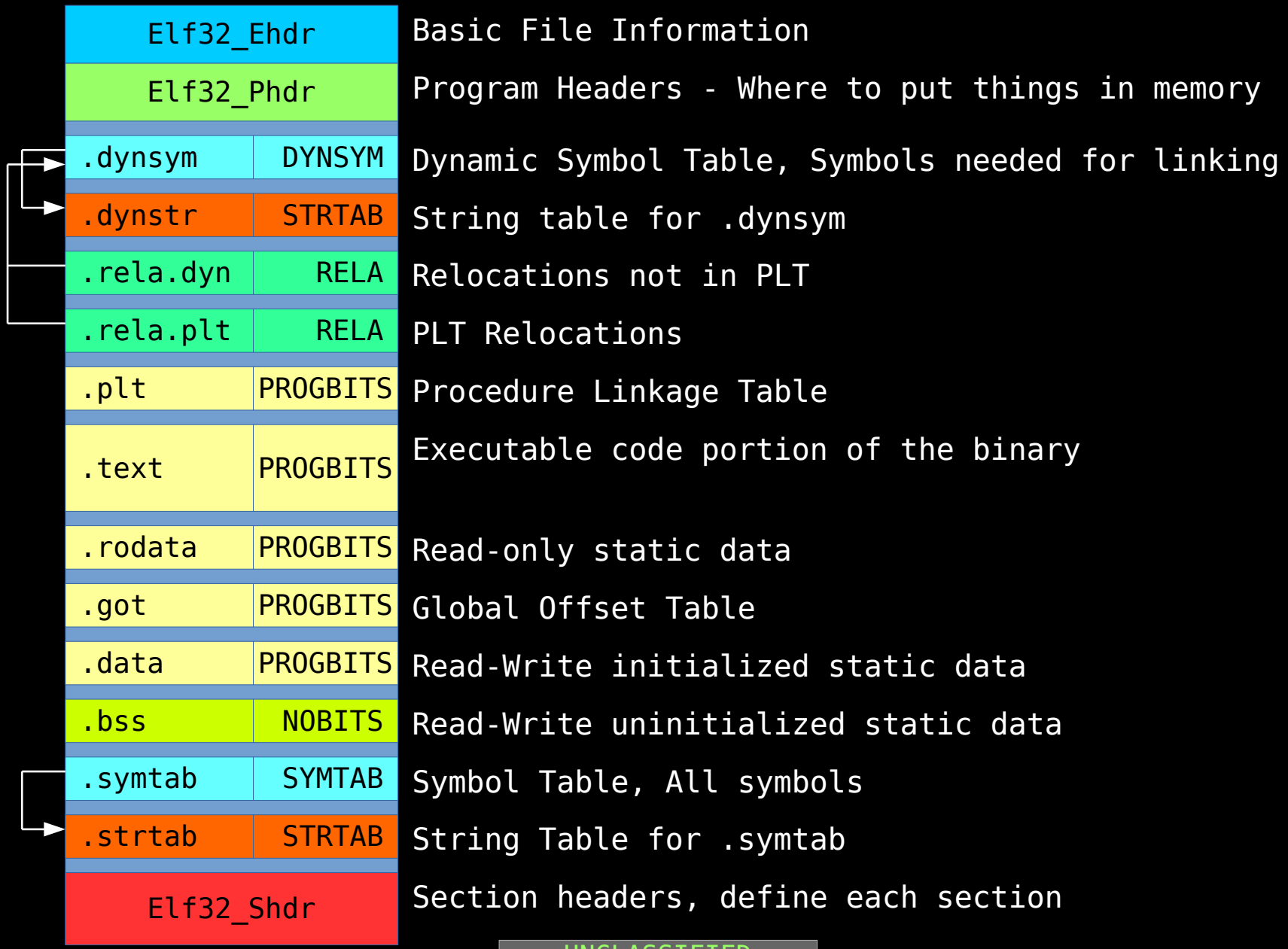
Basic Information Executable Formats Must Answer

- Where different sections of the file should go in memory.
- What permissions these sections of memory should have.
- Relocations (how we link libraries together).
- The address of the first instruction to execute, or the, “Entry Point.”

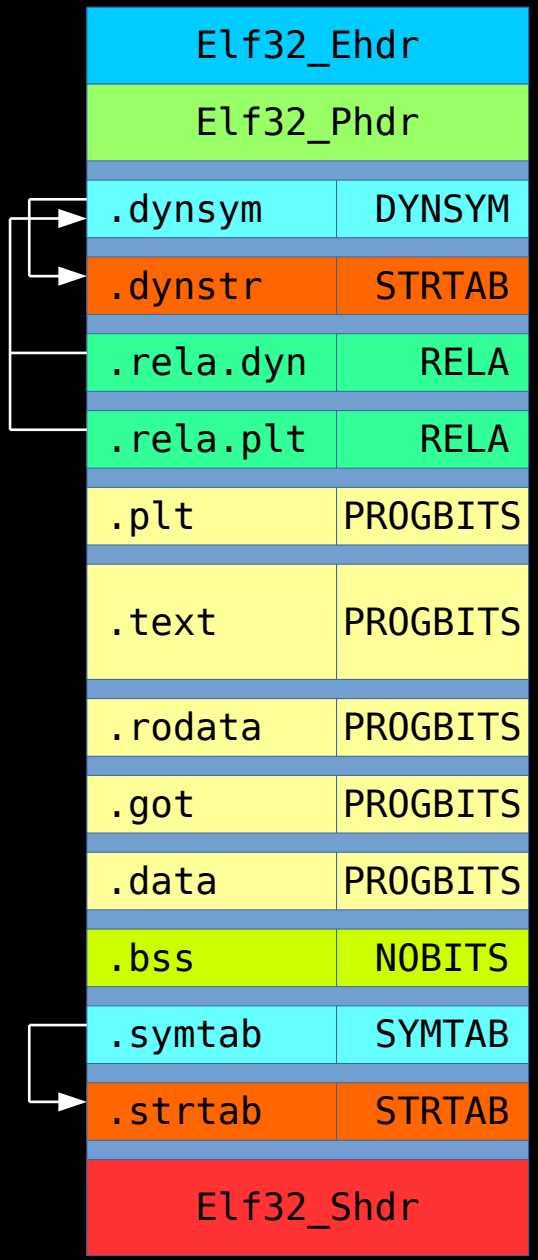
Basic Format Agnostic Example



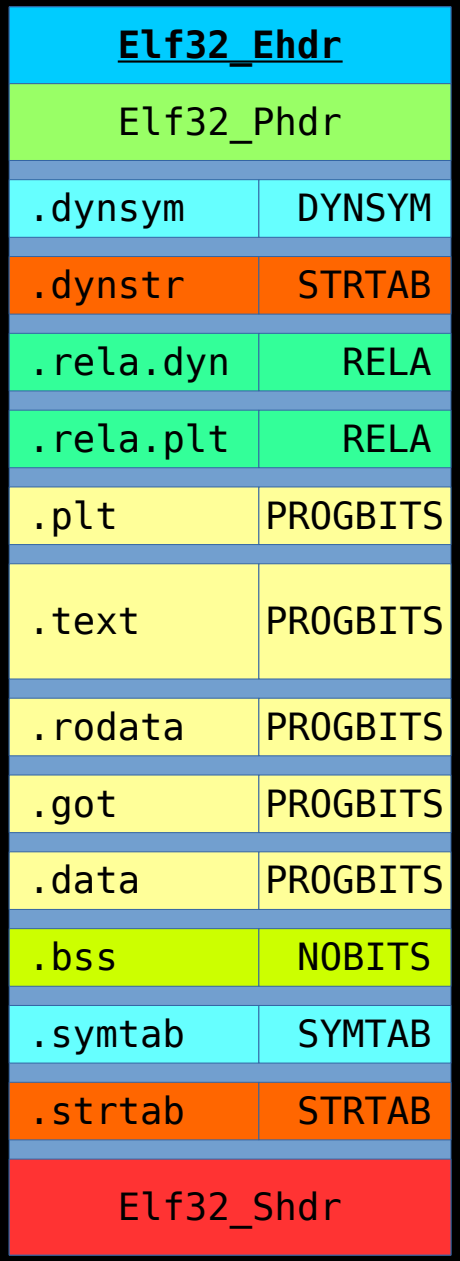
Executable Linkable Format



Executable Linkable Format

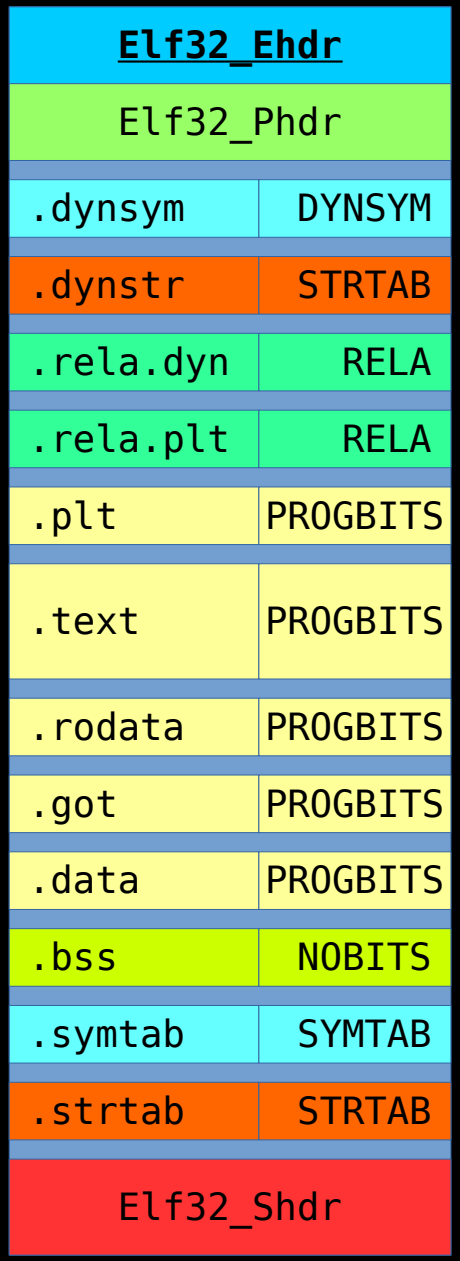


Executable Linkable Format



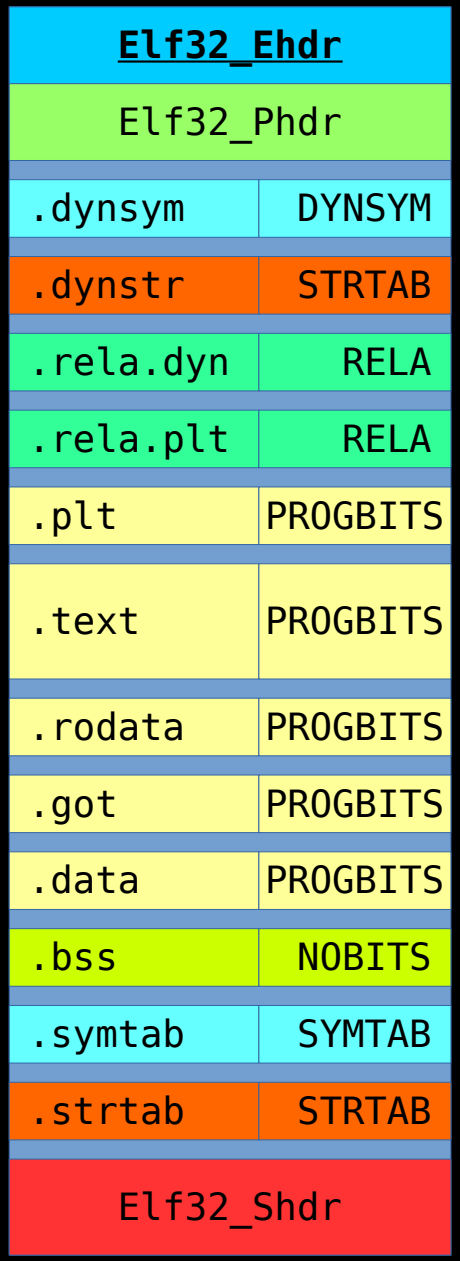

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half    e_type;       /* Object file type */
  Elf32_Half    e_machine;    /* Architecture */
  Elf32_Word    e_version;    /* Object file version */
  Elf32_Addr    e_entry;      /* Entry point virtual address */
  Elf32_Off     e_phoff;      /* Program header table file offset */
  Elf32_Off     e_shoff;      /* Section header table file offset */
  Elf32_Word    e_flags;      /* Processor-specific flags */
  Elf32_Half    e_ehsize;     /* ELF header size in bytes */
  Elf32_Half    e_phentsize;  /* Program header table entry size */
  Elf32_Half    e_phnum;      /* Program header table entry count */
  Elf32_Half    e_shentsize;  /* Section header table entry size */
  Elf32_Half    e_shnum;      /* Section header table entry count */
  Elf32_Half    e_shstrndx;   /* Section header string table index */
} Elf32_Ehdr;

#define EI_NIDENT (16)

#define EI_CLASS        4       /* File class byte index */
#define ELFCLASS32      1       /* 32-bit objects */
#define ELFCLASS64      2       /* 64-bit objects */

#define EI_DATA         5       /* Data encoding byte index */
#define ELFDATA2LSB     1       /* 2's complement, little endian */
#define ELFDATA2MSB     2       /* 2's complement, big endian */
#define EI_VERSION      6       /* File version byte index */
                                /* Value must be EV_CURRENT */
```

Executable Linkable Format



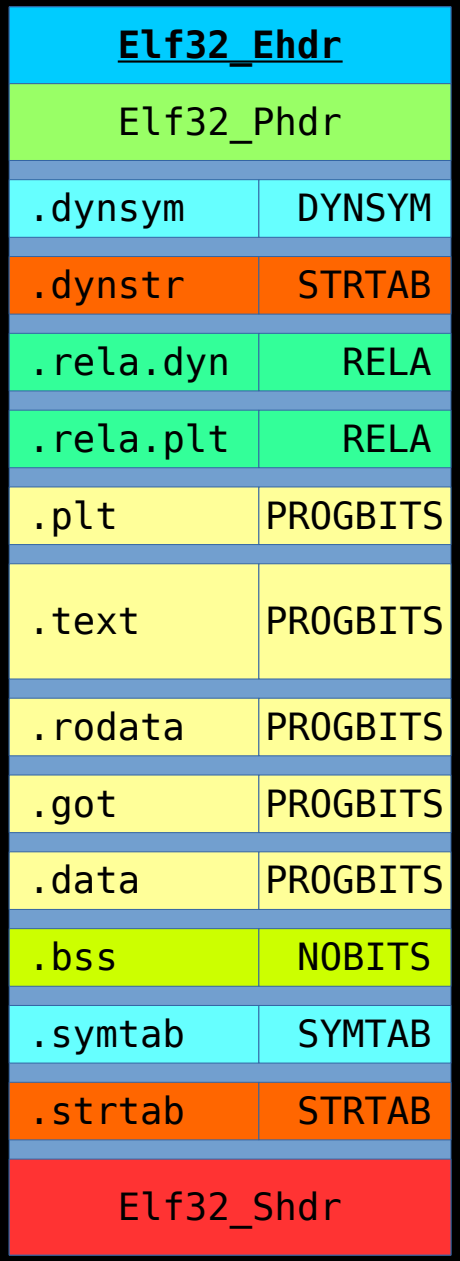

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half    e_type;       /* Object file type */
  Elf32_Half    e_machine;    /* Architecture */
  Elf32_Word    e_version;    /* Object file version */
  Elf32_Addr    e_entry;      /* Entry point virtual address */
  Elf32_Off     e_phoff;      /* Program header table file offset */
  Elf32_Off     e_shoff;      /* Section header table file offset */
  Elf32_Word    e_flags;      /* Processor-specific flags */
  Elf32_Half    e_ehsize;     /* ELF header size in bytes */
  Elf32_Half    e_phentsize;  /* Program header table entry size */
  Elf32_Half    e_phnum;      /* Program header table entry count */
  Elf32_Half    e_shentsize;  /* Section header table entry size */
  Elf32_Half    e_shnum;      /* Section header table entry count */
  Elf32_Half    e_shstrndx;   /* Section header string table index */
} Elf32_Ehdr;

#define ET_NONE         0               /* No file type */
#define ET_REL          1               /* Relocatable file */
#define ET_EXEC         2               /* Executable file */
#define ET_DYN          3               /* Shared object file */
#define ET_CORE         4               /* Core file */
```

Executable Linkable Format



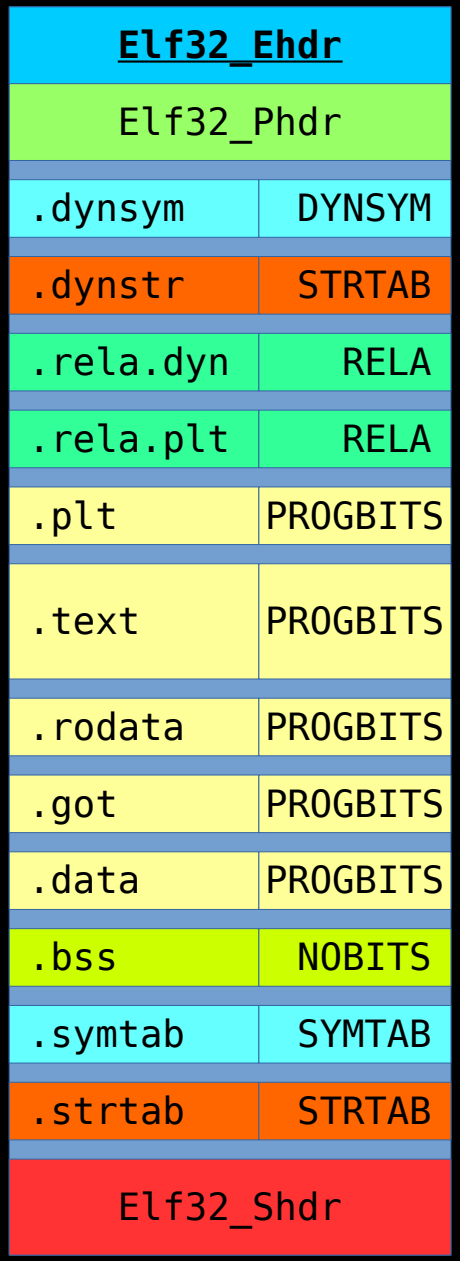

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half    e_type;       /* Object file type */
  Elf32_Half    e_machine;    /* Architecture */
  Elf32_Word    e_version;    /* Object file version */
  Elf32_Addr    e_entry;      /* Entry point virtual address */
  Elf32_Off     e_phoff;      /* Program header table file offset */
  Elf32_Off     e_shoff;      /* Section header table file offset */
  Elf32_Word    e_flags;      /* Processor-specific flags */
  Elf32_Half    e_ehsize;     /* ELF header size in bytes */
  Elf32_Half    e_phentsize;  /* Program header table entry size */
  Elf32_Half    e_phnum;      /* Program header table entry count */
  Elf32_Half    e_shentsize;  /* Section header table entry size */
  Elf32_Half    e_shnum;      /* Section header table entry count */
  Elf32_Half    e_shstrndx;   /* Section header string table index */
} Elf32_Ehdr;

#define EM_NONE          0              /* No machine */
#define EM_SPARC         2              /* SUN SPARC */
#define EM_386           3              /* Intel 80386 */
#define EM_MIPS          8              /* MIPS R3000 big-endian */
#define EM_ARM          40              /* ARM */
#define EM_X86_64       62              /* AMD x86-64 architecture */
```

Executable Linkable Format




```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half    e_type;       /* Object file type */
  Elf32_Half    e_machine;    /* Architecture */
  Elf32_Word    e_version;    /* Object file version */
  Elf32_Addr    e_entry;      /* Entry point virtual address */
  Elf32_Off     e_phoff;      /* Program header table file offset */
  Elf32_Off     e_shoff;      /* Section header table file offset */
  Elf32_Word    e_flags;      /* Processor-specific flags */
  Elf32_Half    e_ehsize;     /* ELF header size in bytes */
  Elf32_Half    e_phentsize;  /* Program header table entry size */
  Elf32_Half    e_phnum;      /* Program header table entry count */
  Elf32_Half    e_shentsize;  /* Section header table entry size */
  Elf32_Half    e_shnum;      /* Section header table entry count */
  Elf32_Half    e_shstrndx;   /* Section header string table index */
} Elf32_Ehdr;

#define EV_NONE         0               /* Invalid ELF version */
#define EV_CURRENT      1               /* Current version */
#define EV_NUM          2
```

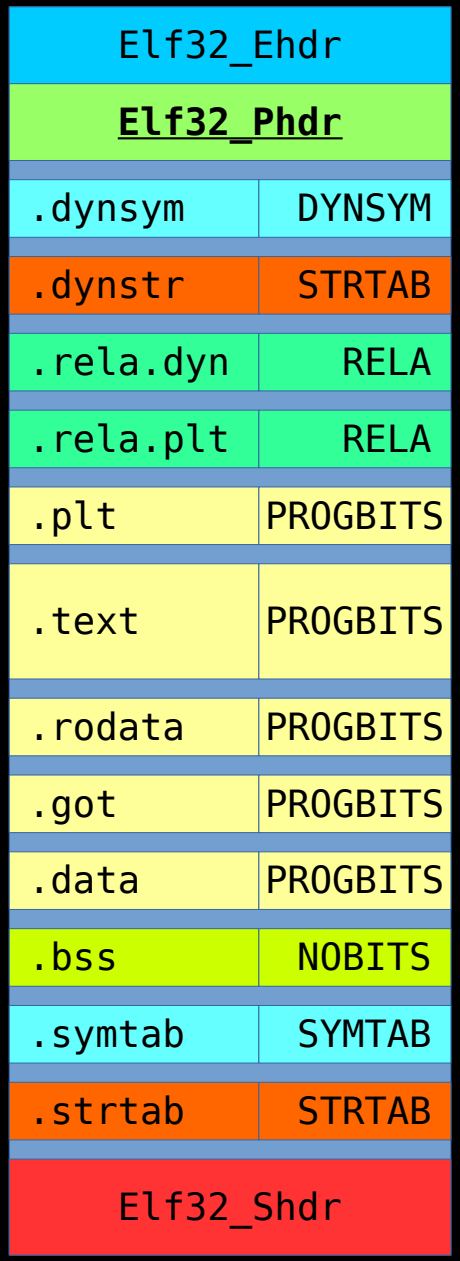
Executable Linkable Format




```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half    e_type;      /* Object file type */
  Elf32_Half    e_machine;   /* Architecture */
  Elf32_Word    e_version;   /* Object file version */
  Elf32_Addr    e_entry;     /* Entry point virtual address */
  Elf32_Off     e_phoff;     /* Program header table file offset */
  Elf32_Off     e_shoff;     /* Section header table file offset */
  Elf32_Word    e_flags;     /* Processor-specific flags */
  Elf32_Half    e_ehsize;    /* ELF header size in bytes */
  Elf32_Half    e_phentsize; /* Program header table entry size */
  Elf32_Half    e_phnum;     /* Program header table entry count */
  Elf32_Half    e_shentsize; /* Section header table entry size */
  Elf32_Half    e_shnum;     /* Section header table entry count */
  Elf32_Half    e_shstrndx;  /* Section header string table index */
} Elf32_Ehdr;
```

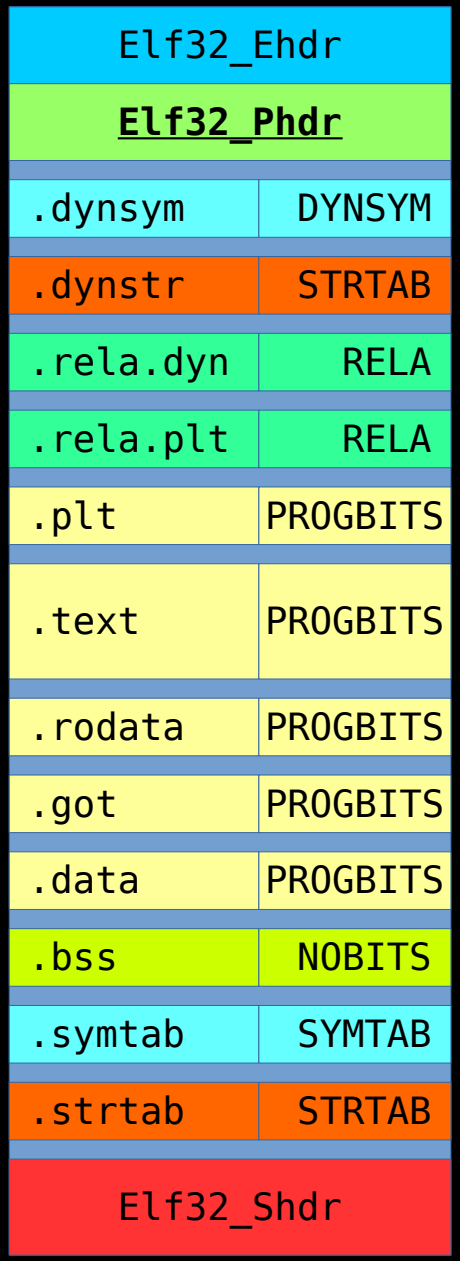
```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4048c5
  Start of program headers:          64 (bytes into file)
  Start of section headers:          116552 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
```

Executable Linkable Format



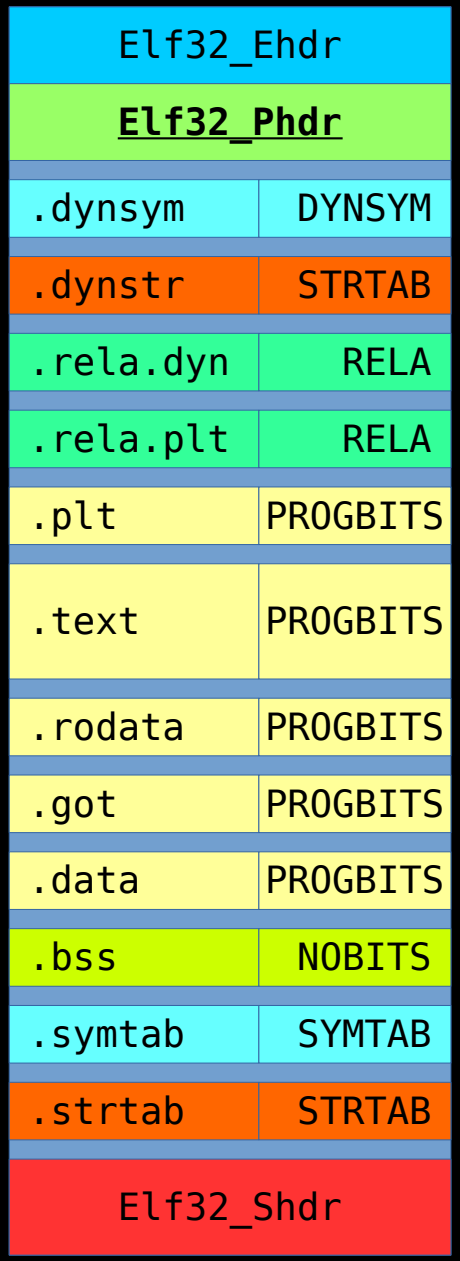

```
typedef struct
{
  Elf32_Word    p_type;                 /* Segment type */
  Elf32_Off     p_offset;               /* Segment file offset */
  Elf32_Addr    p_vaddr;                /* Segment virtual address */
  Elf32_Addr    p_paddr;                /* Segment physical address */
  Elf32_Word    p_filesz;               /* Segment size in file */
  Elf32_Word    p_memsz;                /* Segment size in memory */
  Elf32_Word    p_flags;                /* Segment flags */
  Elf32_Word    p_align;                /* Segment alignment */
} Elf32_Phdr;

#define PT_NULL         0    /* Program header table entry unused */
#define PT_LOAD         1    /* Loadable program segment */
#define PT_DYNAMIC      2    /* Dynamic linking information */
#define PT_INTERP       3    /* Program interpreter */
#define PT_NOTE         4    /* Auxiliary information */
#define PT_SHLIB        5    /* Reserved */
#define PT_PHDR         6    /* Entry for header table itself */
#define PT_TLS          7    /* Thread-local storage segment */
#define PT_NUM          8    /* Number of defined types */
```

Executable Linkable Format




```
typedef struct
{
  Elf32_Word    p_type;             /* Segment type */
  Elf32_Off     p_offset;           /* Segment file offset */
  Elf32_Addr    p_vaddr;            /* Segment virtual address */
  Elf32_Addr    p_paddr;            /* Segment physical address */
  Elf32_Word    p_filesz;           /* Segment size in file */
  Elf32_Word    p_memsz;            /* Segment size in memory */
  Elf32_Word    p_flags;            /* Segment flags */
  Elf32_Word    p_align;            /* Segment alignment */
} Elf32_Phdr;

#define PF_X            (1 << 0)        /* Segment is executable */
#define PF_W            (1 << 1)        /* Segment is writable */
#define PF_R            (1 << 2)        /* Segment is readable */
#define PF_MASKOS       0x0ff00000      /* OS-specific */
#define PF_MASKPROC     0xf0000000      /* Processor-specific */
```

Executable Linkable Format

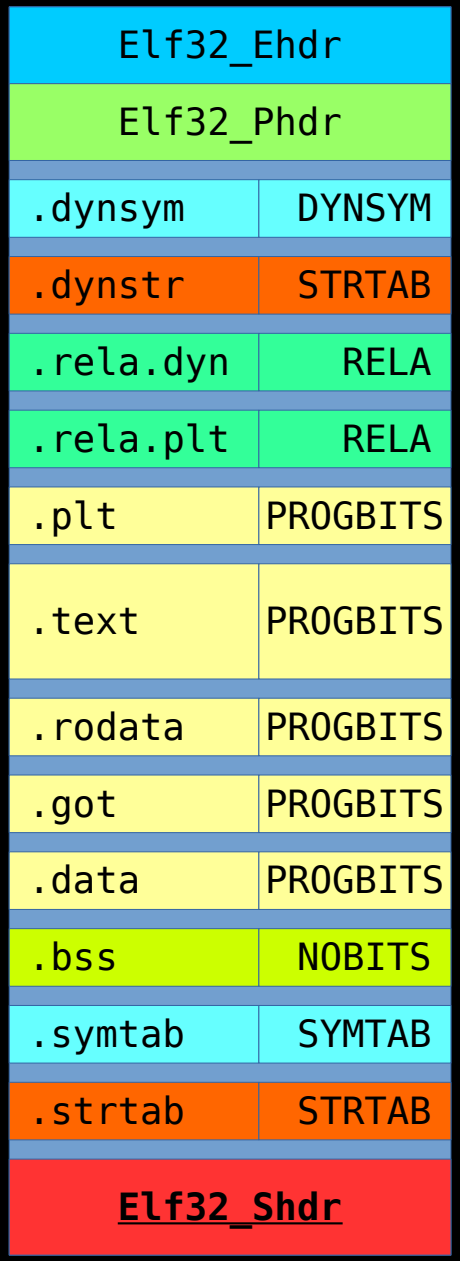



```
typedef struct
{
  Elf32_Word    p_type;                 /* Segment type */
  Elf32_Off     p_offset;               /* Segment file offset */
  Elf32_Addr    p_vaddr;                /* Segment virtual address */
  Elf32_Addr    p_paddr;                /* Segment physical address */
  Elf32_Word    p_filesz;               /* Segment size in file */
  Elf32_Word    p_memsz;                /* Segment size in memory */
  Elf32_Word    p_flags;                /* Segment flags */
  Elf32_Word    p_align;                /* Segment alignment */
} Elf32_Phdr;

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
...
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000001bad4 0x000000000001bad4  R E    200000
  LOAD           0x000000000001bdf0 0x000000000061bdf0 0x000000000061bdf0
                 0x0000000000000864 0x0000000000001690  RW     200000
```

p_paddr = no one cares

"On systems for which physical addressing is relevant, this member is reserved for the segment's physical address. Because System V ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects."

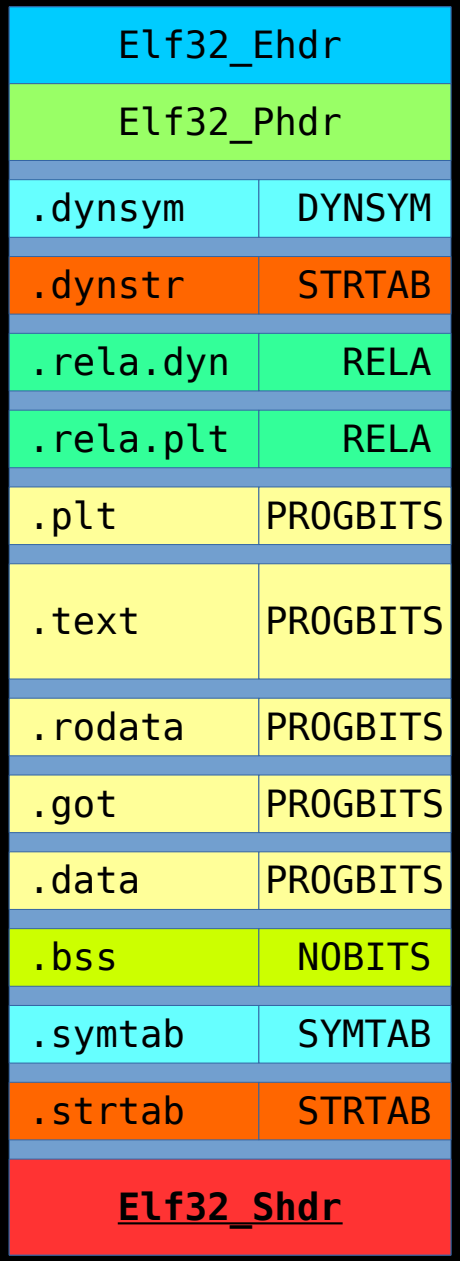
Executable Linkable Format




```
typedef struct
{
  Elf32_Word    sh_name;      /* Section name (string tbl index) */
  Elf32_Word    sh_type;      /* Section type */
  Elf32_Word    sh_flags;     /* Section flags */
  Elf32_Addr    sh_addr;      /* Section virtual addr at execution */
  Elf32_Off     sh_offset;    /* Section file offset */
  Elf32_Word    sh_size;      /* Section size in bytes */
  Elf32_Word    sh_link;      /* Link to another section */
  Elf32_Word    sh_info;      /* Additional section information */
  Elf32_Word    sh_addralign; /* Section alignment */
  Elf32_Word    sh_entsize;   /* Entry size if section holds table */
} Elf32_Shdr;
```

A String Table Index?

Executable Linkable Format

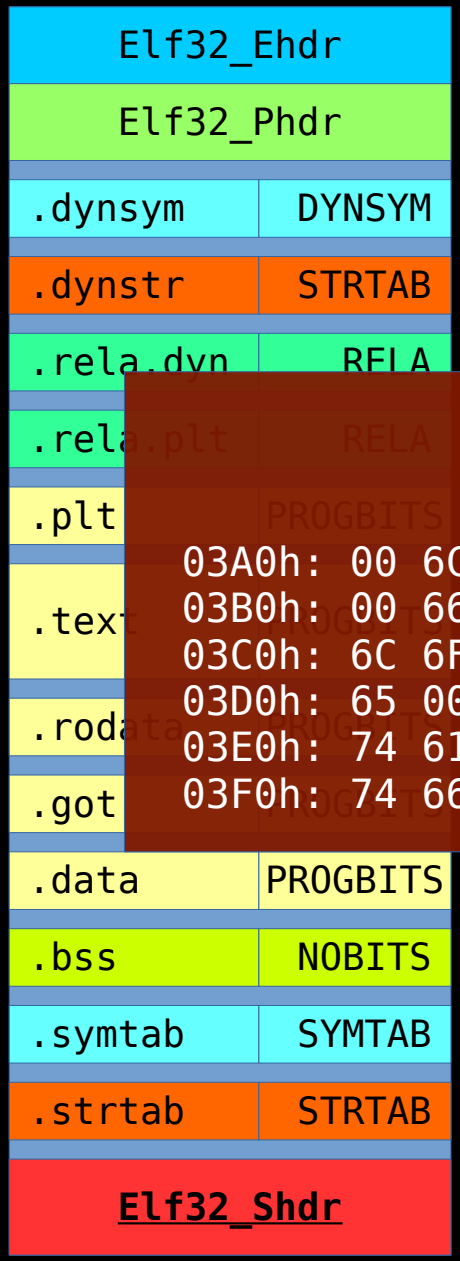



```
typedef struct
{
  Elf32_Word    sh_name;      /* Section name (string tbl index) */
  Elf32_Word    sh_type;      /* Section type */
  Elf32_Word    sh_flags;     /* Section flags */
  Elf32_Addr    sh_addr;      /* Section virtual addr at execution */
  Elf32_Off     sh_offset;    /* Section file offset */
  Elf32_Word    sh_size;      /* Section size in bytes */
  Elf32_Word    sh_link;      /* Link to another section */
  Elf32_Word    sh_info;      /* Additional section information */
  Elf32_Word    sh_addralign; /* Section alignment */
  Elf32_Word    sh_entsize;   /* Entry size if section holds table */
} Elf32_Shdr;
```

A String Table Index?

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section.

Executable Linkable Format



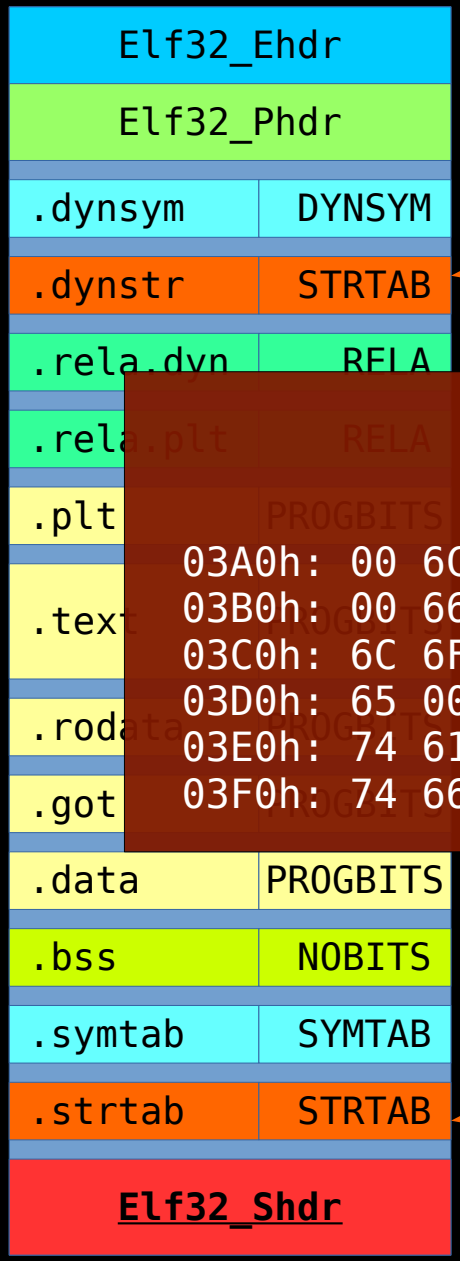

```
typedef struct
{
  Elf32_Word    sh_name;        /* Section name (string tbl index) */
  Elf32_Word    sh_type;        /* Section type */
  Elf32_Word    sh_flags;       /* Section flags */
  Elf32_Addr    sh_addr;        /* Section virtual addr at execution */
  Elf32_Off     sh_offset;      /* Section file offset */
  Elf32_Word    sh_size;        /* Section size in bytes */
  Elf32_Word    sh_link;        /* Link to another section */
  Elf32_Word    sh_info;        /* Additional section information */
  Elf32_Word    sh_addralign;   /* Section alignment */
  Elf32_Word    sh_entsize;     /* Entry size if section holds table */
} Elf32_Shdr;
```

This is what a string table looks like

```
03A0h: 00 6C 69 62 63 2E 73 6F 2E 36 00 66 6F 70 65 6E  .libc.so.6.fopen
03B0h: 00 66 74 65 6C 6C 00 66 73 65 65 6B 00 6D 61 6C  .ftell.fseek.mal
03C0h: 6C 6F 63 00 73 74 64 65 72 72 00 66 77 72 69 74  loc.stderr.fwrit
03D0h: 65 00 66 72 65 61 64 00 5F 5F 6C 69 62 63 5F 73  e.fread.__libc_s
03E0h: 74 61 72 74 5F 6D 61 69 6E 00 73 6E 70 72 69 6E  tart_main.snprin
03F0h: 74 66 00 5F 5F 67 6D 6F 6E 5F 73 74 61 72 74 5F  tf.__gmon_start_
```

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section.

Executable Linkable Format




```
typedef struct
{
  Elf32_Word    sh_name;        /* Section name (string tbl index) */
  Elf32_Word    sh_type;        /* Section type */
  Elf32_Word    sh_flags;       /* Section flags */
  Elf32_Addr    sh_addr;        /* Section virtual addr at execution */
  Elf32_Off     sh_offset;      /* Section file offset */
  Elf32_Word    sh_size;        /* Section size in bytes */
  Elf32_Word    sh_link;        /* Link to another section */
  Elf32_Word    sh_info;        /* Additional section information */
  Elf32_Word    sh_addralign;   /* Section alignment */
  Elf32_Word    sh_entsize;     /* Entry size if section holds table */
} Elf32_Shdr;
```

This is a string table

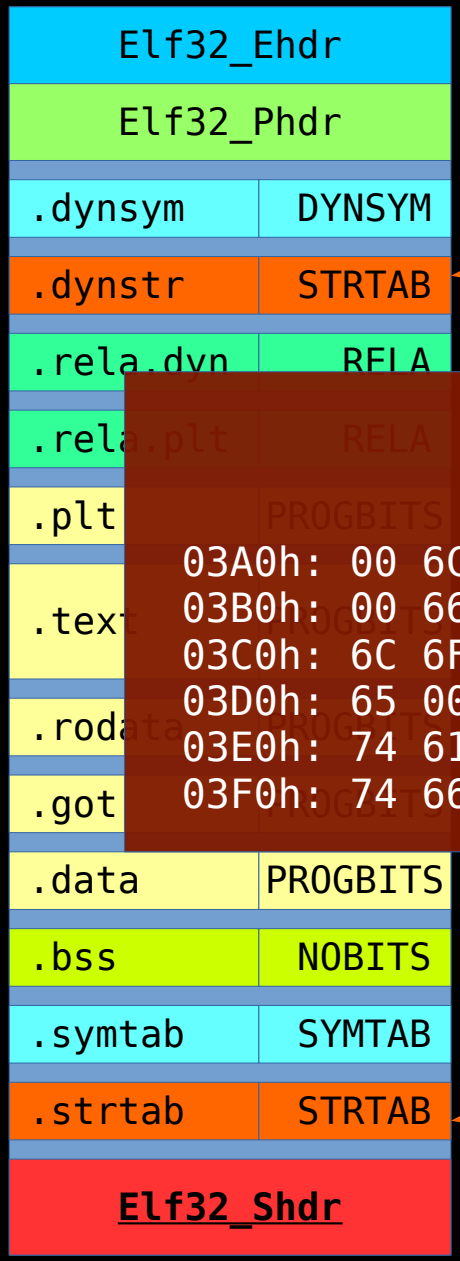
This is what a string table looks like

```
03A0h: 00 6C 69 62 63 2E 73 6F 2E 36 00 66 6F 70 65 6E  .libc.so.6.fopen
03B0h: 00 66 74 65 6C 6C 00 66 73 65 65 6B 00 6D 61 6C  .ftell.fseek.mal
03C0h: 6C 6F 63 00 73 74 64 65 72 72 00 66 77 72 69 74  loc.stderr.fwrit
03D0h: 65 00 66 72 65 61 64 00 5F 5F 6C 69 62 63 5F 73  e.fread.__libc_s
03E0h: 74 61 72 74 5F 6D 61 69 6E 00 73 6E 70 72 69 6E  tart_main.snprin
03F0h: 74 66 00 5F 5F 67 6D 6F 6E 5F 73 74 61 72 74 5F  tf.__gmon_start_
```

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol names. One references a string as an index into the string table section.

This is a string table

Executable Linkable Format




```
typedef struct
{
  Elf32_Wo                                  tring tbl index) */
  Elf32_Wo
  Elf32_Wo                                  /
  Elf32_Ad                                   addr at execution */
  Elf32_Of                                  fset */
  Elf32_Wo                                   bytes */
  Elf32_W                                   section */
  Elf32_W                                   ion information */
  Elf32_W                                   at */
  Elf32_W                                   ection holds table */
```

But wait! How do I know which string table to use when I look up the names of sections?

This i

```
03A0h: 00 6C 69 62 63                                  libc.so.6.fopen
03B0h: 00 66 74 65 6C                                  ftell.fseek.mal
03C0h: 6C 6F 63 00 73 7                                loc.stderr.fwrit
03D0h: 65 00 66 72 65 61 64 00 5F 5F 6C 69          73 e.fread.__libc_s
03E0h: 74 61 72 74 5F 6D 61 69 6E 00 73 6E 70      5E tart_main.snprin
03F0h: 74 66 00 5F 5F 67 6D 6F 6E 5F 73 74 61 72       tf.__gmon_start_
```

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol names. One reference uses a string as an index into the string table section.

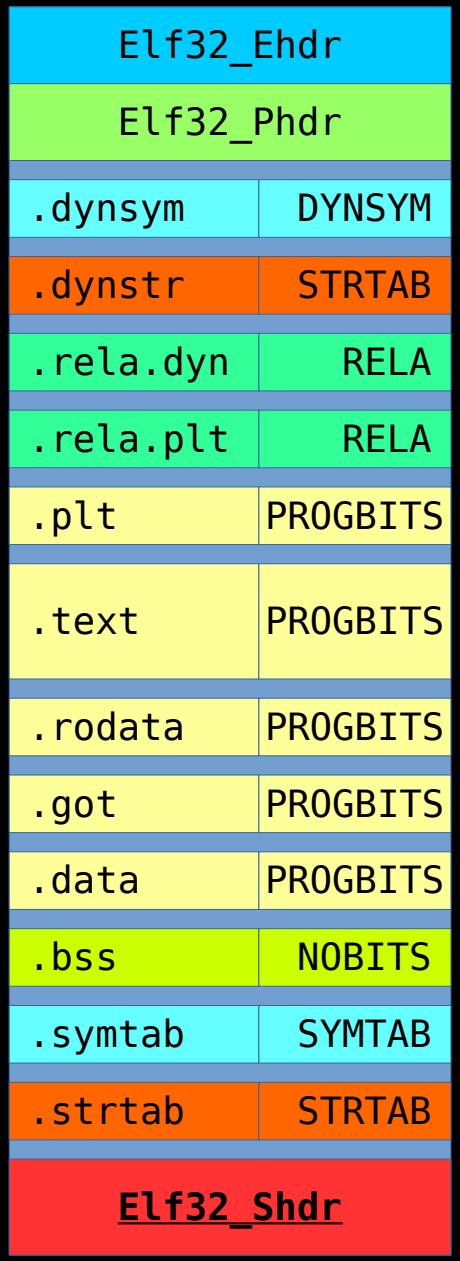
This is a string table

Executable Linkable Format

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half    e_type;       /* Object file type */
  Elf32_Half    e_machine;    /* Architecture */
  Elf32_Word    e_version;    /* Object file version */
  Elf32_Addr    e_entry;      /* Entry point virtual address */
  Elf32_Off     e_phoff;      /* Program header table file offset */
  Elf32_Off     e_shoff;      /* Section header table file offset */
  Elf32_Word    e_flags;      /* Processor-specific flags */
  Elf32_Half    e_ehsize;     /* ELF header size in bytes */
  Elf32_Half    e_phentsize;  /* Program header table entry size */
  Elf32_Half    e_phnum;      /* Program header table entry count */
  Elf32_Half    e_shentsize;  /* Section header table entry size */
  Elf32_Half    e_shnum;      /* Section header table entry count */
  Elf32_Half    e_shstrndx;   /* Section header string table index */
} Elf32_Ehdr;

ELF Header:
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4048c5
  Start of program headers:          64 (bytes into file)
  Start of section headers:          16552 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
```

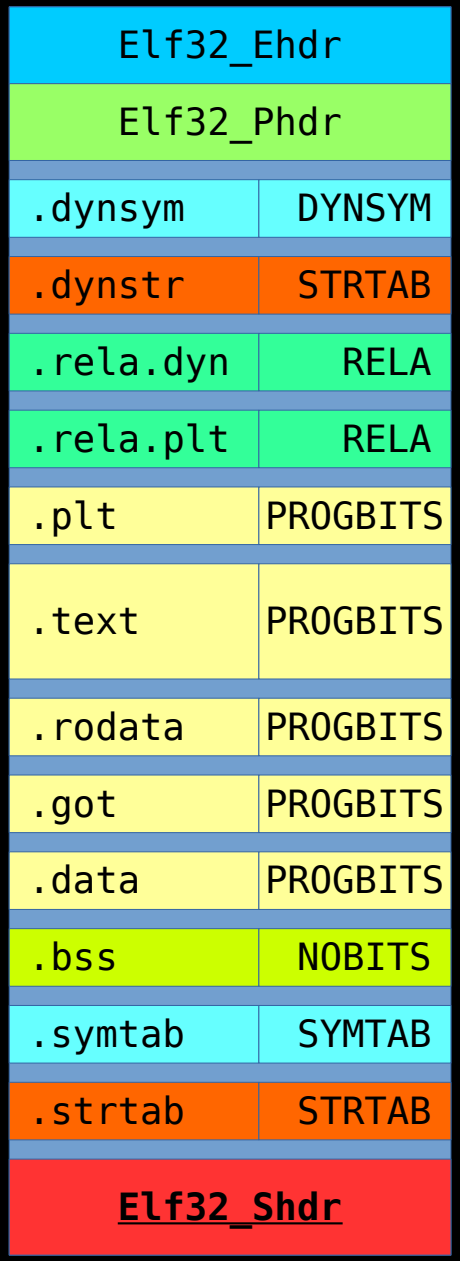
e_shstrndx from Elf32_Ehdr tells us which section holds the the string table for the names of all the sections. Typically this section is called ".shstrtab" and isn't listed in this class because I ran out of room for my cool ELF File graphic with all the other sections.

Executable Linkable Format



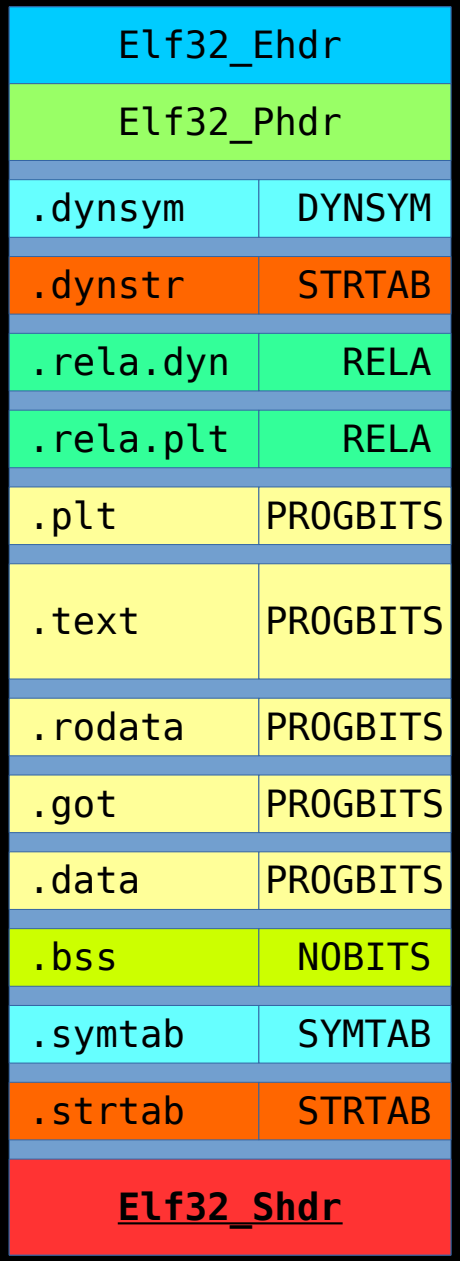

```
typedef struct
{
  Elf32_Word    sh_name;      /* Section name (string tbl index) */
  Elf32_Word    sh_type;      /* Section type */
  Elf32_Word    sh_flags;     /* Section flags */
  Elf32_Addr    sh_addr;      /* Section virtual addr at execution */
  Elf32_Off     sh_offset;    /* Section file offset */
  Elf32_Word    sh_size;      /* Section size in bytes */
  Elf32_Word    sh_link;      /* Link to another section */
  Elf32_Word    sh_info;      /* Additional section information */
  Elf32_Word    sh_addralign; /* Section alignment */
  Elf32_Word    sh_entsize;   /* Entry size if section holds table */
} Elf32_Shdr;

#define SHT_NULL        0     /* Section header table entry unused */
#define SHT_PROGBITS    1     /* Program data */
#define SHT_SYMTAB      2     /* Symbol table */
#define SHT_STRTAB      3     /* String table */
#define SHT_RELA        4     /* Relocation entries with addends */
#define SHT_HASH        5     /* Symbol hash table */
#define SHT_DYNAMIC     6     /* Dynamic linking information */
#define SHT_NOTE        7     /* Notes */
#define SHT_NOBITS      8     /* Program space with no data (bss) */
#define SHT_REL         9     /* Relocation entries, no addends */
```

Executable Linkable Format




```
typedef struct
{
  Elf32_Word    sh_name;      /* Section name (string tbl index) */
  Elf32_Word    sh_type;      /* Section type */
  Elf32_Word    sh_flags;     /* Section flags */
  Elf32_Addr    sh_addr;      /* Section virtual addr at execution */
  Elf32_Off     sh_offset;    /* Section file offset */
  Elf32_Word    sh_size;      /* Section size in bytes */
  Elf32_Word    sh_link;      /* Link to another section */
  Elf32_Word    sh_info;      /* Additional section information */
  Elf32_Word    sh_addralign; /* Section alignment */
  Elf32_Word    sh_entsize;   /* Entry size if section holds table */
} Elf32_Shdr;

#define SHF_WRITE       (1 << 0) /* Writable */
#define SHF_ALLOC       (1 << 1) /* Occupies memory during execution */
#define SHF_EXECINSTR   (1 << 2) /* Executable */
#define SHF_MERGE       (1 << 4) /* Might be merged */
#define SHF_STRINGS     (1 << 5) /* Contains nul-terminated strings */
#define SHF_INFO_LINK   (1 << 6) /* `sh_info' contains SHT index */
#define SHF_LINK_ORDER  (1 << 7) /* Preserve order after combining */
```

Executable Linkable Format




```
typedef struct
{
  Elf32_Word    sh_name;      /* Section name (string tbl index) */
  Elf32_Word    sh_type;      /* Section type */
  Elf32_Word    sh_flags;     /* Section flags */
  Elf32_Addr    sh_addr;      /* Section virtual addr at execution */
  Elf32_Off     sh_offset;    /* Section file offset */
  Elf32_Word    sh_size;      /* Section size in bytes */
  Elf32_Word    sh_link;      /* Link to another section */
  Elf32_Word    sh_info;      /* Additional section information */
  Elf32_Word    sh_addralign; /* Section alignment */
  Elf32_Word    sh_entsize;   /* Entry size if section holds table */
} Elf32_Shdr;
```

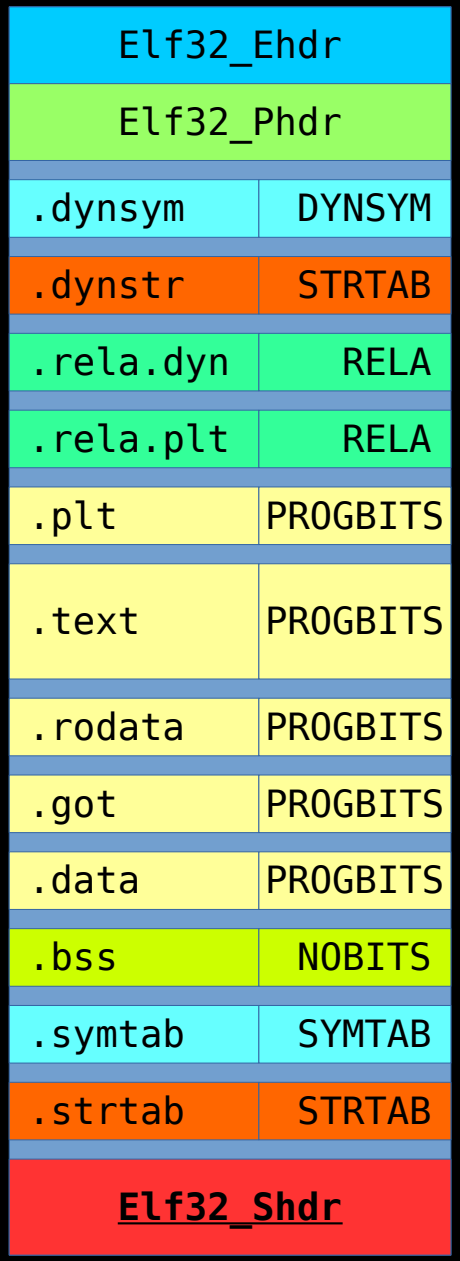
Remember:

sh_addr *is for suckers*

Remember, *program headers* tell us where things will land in memory, *not section headers*.

The only reason sh_addr exists is to break bad tools, or draw attention away from program headers *where people should be anyway*.

Executable Linkable Format




```
typedef struct
{
  Elf32_Word    sh_name;        /* Section name (string tbl index) */
  Elf32_Word    sh_type;        /* Section type */
  Elf32_Word    sh_flags;       /* Section flags */
  Elf32_Addr    sh_addr;        /* Section virtual addr at execution */
  Elf32_Off     sh_offset;      /* Section file offset */
  Elf32_Word    sh_size;        /* Section size in bytes */
  Elf32_Word    sh_link;        /* Link to another section */
  Elf32_Word    sh_info;        /* Additional section information */
  Elf32_Word    sh_addralign; /* Section alignment */
  Elf32_Word    sh_entsize;     /* Entry size if section holds table */
} Elf32_Shdr;
```

`sh_offset:` “This member’s value gives the byte offset from the beginning of the file to the first byte in the section”

`sh_size:` “Section size in bytes”

`sh_link:` Depends on section type, but some sections depend on other sections. For example, Symbol Tables are linked to String Tables that hold the symbol names.

`sh_link:` Depends on section type, but some sections depend on other sections. For example, Symbol Tables are linked to String Tables that hold the symbol names.

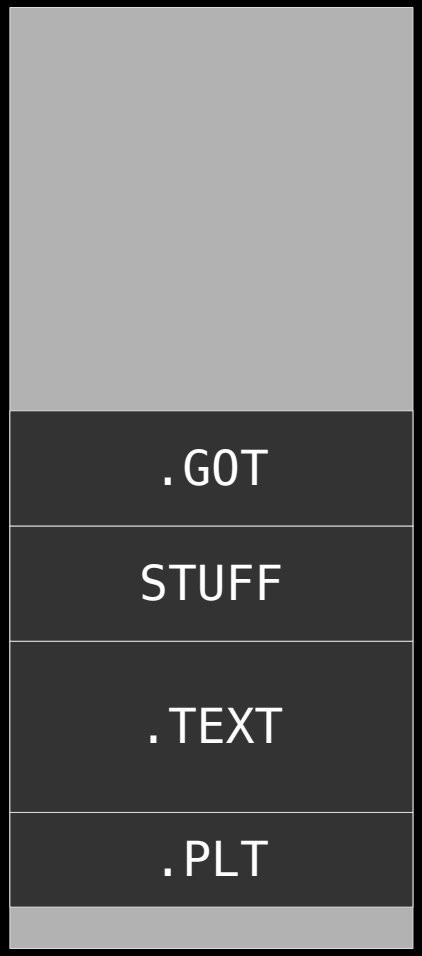
`sh_addralign:` ...

`sh_entsite:` ...

Dynamic Linking

Time Out!

Let's talk about dynamic linking.

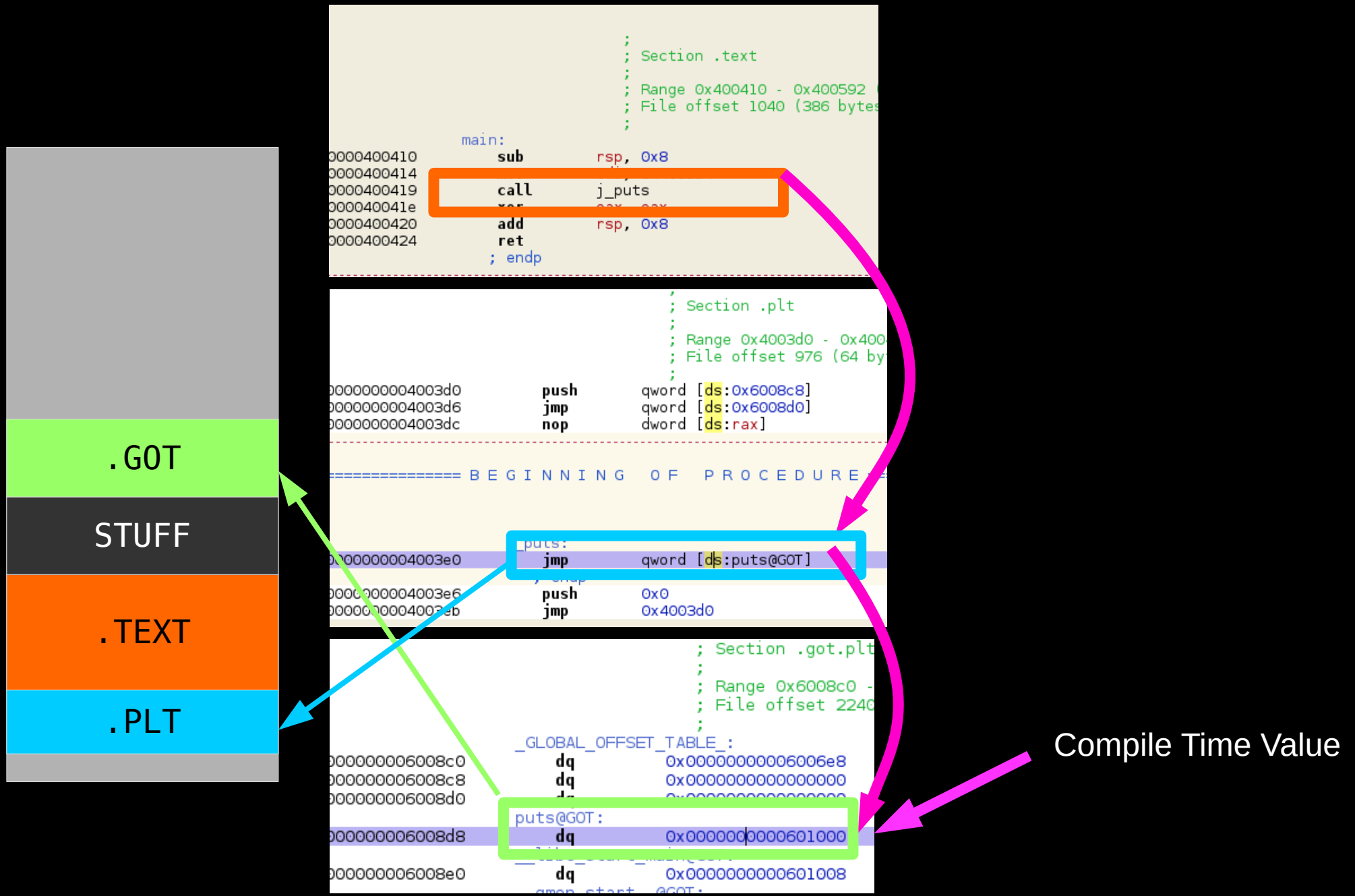


Dynamic Linking

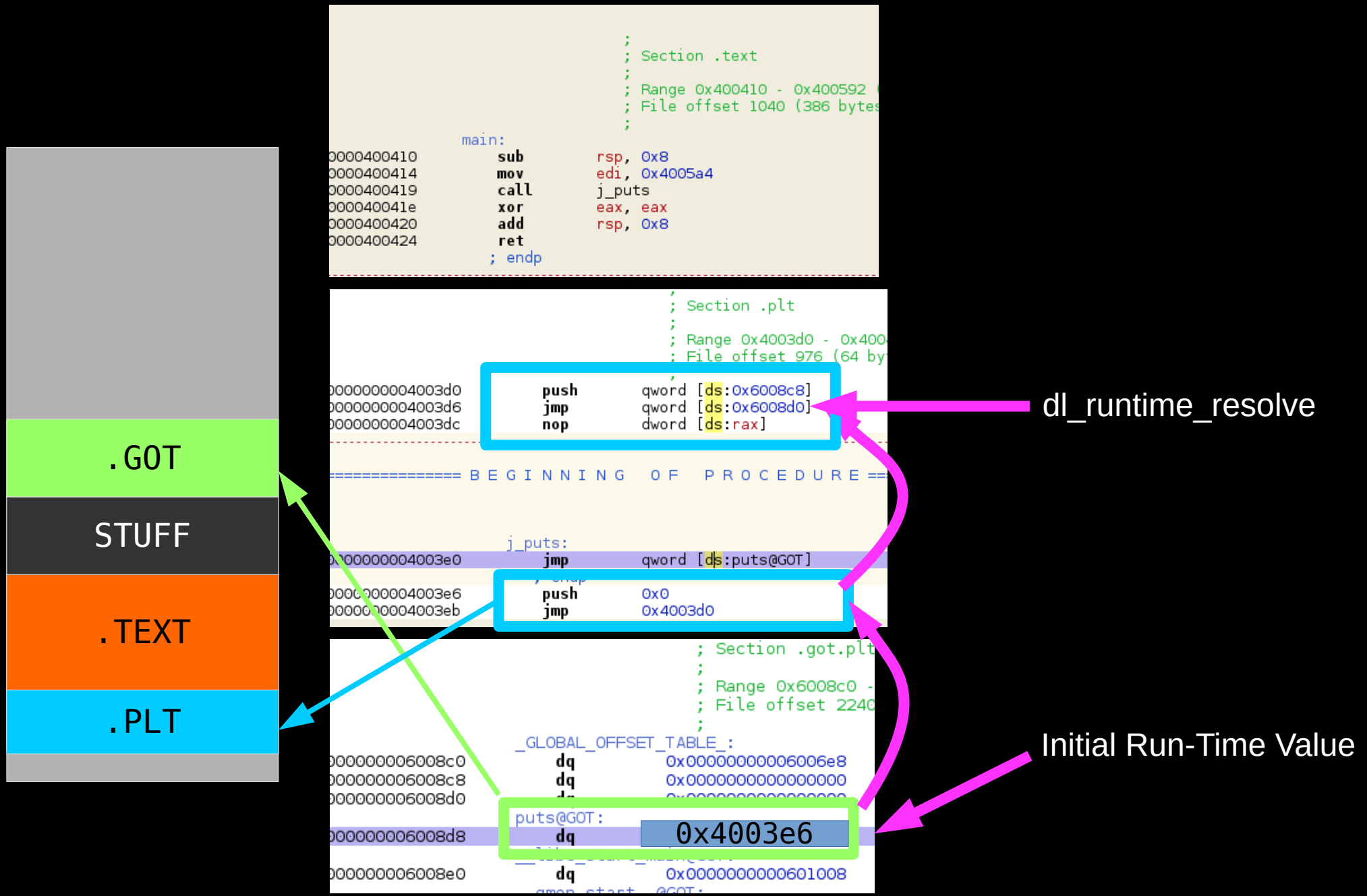
Time Out!

It's time for ASSEMBLY!!!

Dynamic Linking



Dynamic Linking



Dynamic Linking

You are now experiencing the joys of, “Lazy Binding/Linking,” though the process goes by many names.

“Finally, GOT … [blah blah] ... the function named _dl_runtime_resolve, which is basically an assembly stub that does some register/stack setup and calls into a C function called dl_fixup(). dl_fixup is the workhorse that actually resolves the symbol in question. Once the symbol's address is found, the program's GOT entry for it must be patched. This is also the job of dl_fixup(). Once dl_fixup() patches the correct GOT entry, the next time the function is called, it will again jump to the PLT entry, but this time the indirect jump there will go to the symbol's address instead of the following instruction.”

“This method of lazy symbol resolution avoids costly lookups for functions that aren't even called. You can force the linker to eagerly resolve symbols on program startup by setting the LD_BIND_NOW environment variable.”

http://users.eecs.northwestern.edu/~kch479/docs/notes/linking.html

Dynamic Linking


```
                                ;
                                ; Section .text
                                ;
                                ; Range 0x400410 - 0x400592
                                ; File offset 1040 (386 bytes
                                ;
                        main:
0000400410                      sub         rsp, 0x8
0000400414                      mov         edi, 0x4005a4
0000400419                      call        j_puts
000040041e                      xor         eax, eax
0000400420                      add         rsp, 0x8
0000400424                      ret
                              ; endp
```

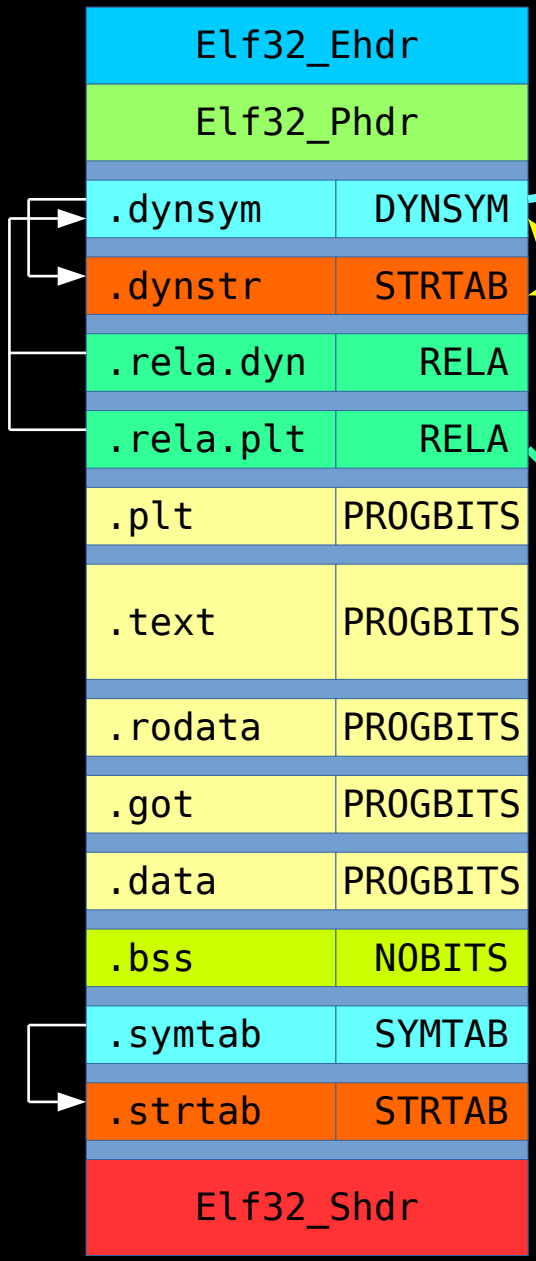
```
                                ;
                                ; Section .plt
                                ;
                                ; Range 0x4003d0 - 0x400
                                ; File offset 976 (64 by
                                ;
0000000004003d0         push        qword [ds:0x6008c8]
0000000004003d6         jmp         qword [ds:0x6008d0]
0000000004003dc         nop         dword [ds:rax]

============== B E G I N N I N G   O F   P R O C E D U R E ==

                        j_puts:
0000000004003e0             jmp         qword [ds:puts@GOT]
                          ; endp
0000000004003e6             push        0x0
0000000004003eb             jmp         0x4003d0
```

```
                                ; Section .got.plt
                                ;
                                ; Range 0x6008c0 -
                                ; File offset 2240
                                ;
                        _GLOBAL_OFFSET_TABLE_:
00000000006008c0            dq          0x00000000006006e8
00000000006008c8            dq          0x0000000000000000
00000000006008d0            dq          0x0000000000000000
                        puts@GOT:
00000000006008d8            dq          0x7ffff7a9ca70
00000000006008e0            dq          0x0000000000601008
                        __gmon_start__@GOT:
```

Value of puts() from libc

Executable Linkable Format

Talk high-level on what the information for linking puts looks like




```
typedef struct
{
  Elf64_Word    st_name;   // "puts"
  unsigned char st_info;   // 0x12
  unsigned char st_other;  // 0x0 DEFAULT
  Elf64_Section st_shndx;  // 0x0
  Elf64_Addr    st_value;  // 0x0
  Elf64_Xword   st_size;   // 0x0
} Elf64_Sym;

ELF64_ST_BIND(st_info) // 0x1 STB_GLOBAL
ELF64_ST_TYPE(st_info) // 0x2 STT_FUNC
```

```
typedef struct
{
  Elf64_Addr    r_offset;
  Elf64_Word    r_info;
} Elf64_Rel;

Elf64_Rel Puts_Rel = {
  0x6008d8,
  0X000100000007;
};

ELF64_R_SYM(Puts_Rel)  // 0x1
ELF64_R_TYPE(Puts_Rel) //
```

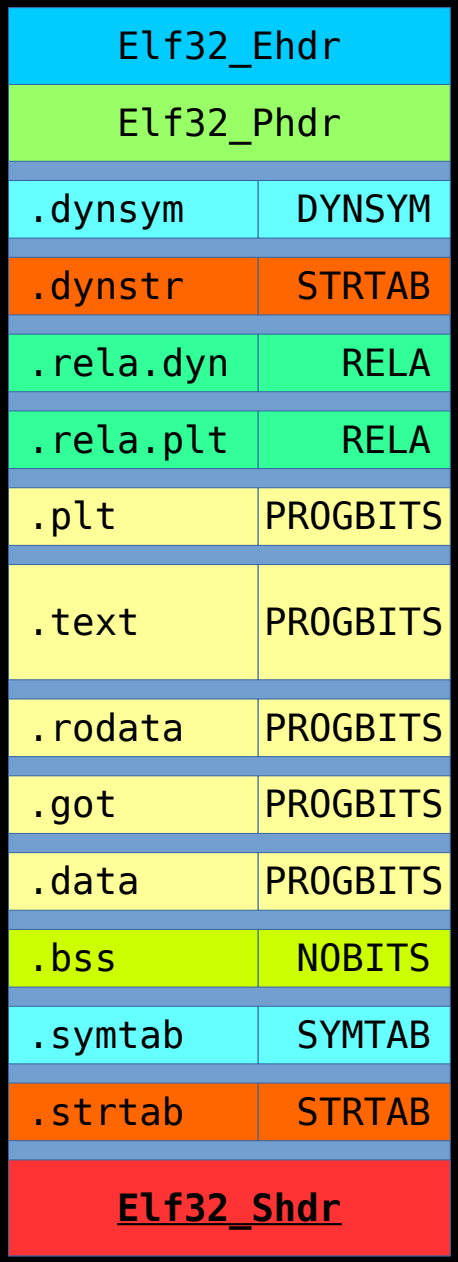
Enough with the Bits

- `/usr/include/elf.h`
- “Executable and Linkable Format (ELF)” (pdf)

Armed with these two documents, and the information we covered today, you can begin working with ELF files

Thought Problems

- What would be required to programmatically modify an ELF in place?
- Could you patch additional functionality into an ELF streaming realtime with only a couple kilobytes of visibility?



Talk is Cheap, Code is Law

- ~~Brief overview of what executable formats do. (15 minutes)~~
- ~~We will break down the ELF file format. (1 hour)~~
- Students will implement a program to parse ELF files. (1 hour)

- Bonus Points: Students can programmatically rearrange, change, or resize portions of an ELF file on disk while maintaining runtime behavior.