ENGR-E 399/599: Embedded Systems Reverse Engineering

Lecture 3: ELFs and metadata for reverse engineering

Austin Roach
ahroach@iu.edu

January 27, 2022

Mystery function

- How many arguments does the function take?
- What are the types of the arguments?
- What does the function do?
- What does the function return?

Mystery function revealed

```
size_t strlen(const char *s)
```

Description

The `strlen()` function calculates the length of the string pointed to by `s`, excluding the terminating null byte.

Return value

The `strlen()` function returns the number of bytes in the string pointed to by `s`.

Today's plan

- What is this ELF file format that we've been looking at?
- Why do we care about the format?
- Where did some of this magical metadata (function starts, function names) come from?
- Bonus topics: relocations and dynamic linking

ELF

- **E**xecutable and **L**inkable **Format**
  - Formerly Extensible Linking Format
- File format for executable files, object files, shared libraries, core dumps
- Supports loading program contents into memory and linking

References:

- Tool interface Standard (TIS) Executable and Linking Format (ELF) Specification
- System V Application Binary Interface
- ELF for the ARM architecture
- System V Application Binary Interface AMD64 Architecture Processor Supplement
- Linux Standard Base Core Specification for AMD64
- Additional specifications for other ABIs, minor variants

So why do we care?

It shows up everywhere!

- Linux software (executables and shared objects)
- Embedded Linux systems (executables and shared objects)
- Embedded system boot loaders
  - Game consoles
- Firmware updates

Why should we study it?

- Source of software reverse engineering metadata
- Non-standard customizations will frustrate your reverse engineering tools
- Other executable formats are likely to have similar elements
  - Tricks with executable files/loading are plentiful in malware

Other executable file formats

- a.out (Primarily older unixes)
- Mach-O (macOS, iOS)
- PE (Windows, EFI environments)
- Innumerable others

Parsing

- Most software RE tools are ELF-aware
- Headers parsed in a limited capacity by a large number of tools (such as `file`)
- `readelf`: standard command-line tool for parsing and displaying header information

ELF Header definition

```
#define EI_NIDENT (16)
typedef struct
{
    unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
    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;
```

readelf -h arm-hello-world

```
Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF32
Data:                              2's complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              DYN (Shared object file)
Machine:                           ARM
Version:                           0x1
Entry point address:               0x3f4
Start of program headers:          52 (bytes into file)
Start of section headers:          7012 (bytes into file)
Flags:                             0x5000200, Version5 EABI, soft-float ABI
Size of this header:               52 (bytes)
Size of program headers:           32 (bytes)
Number of program headers:         9
Size of section headers:           40 (bytes)
Number of section headers:         29
```

Program segments

Loader's view of program

- What parts of file should be loaded into memory
- What addresses the segments should be loaded at
- What permissions should be set on the memory segments

Program segment header definition

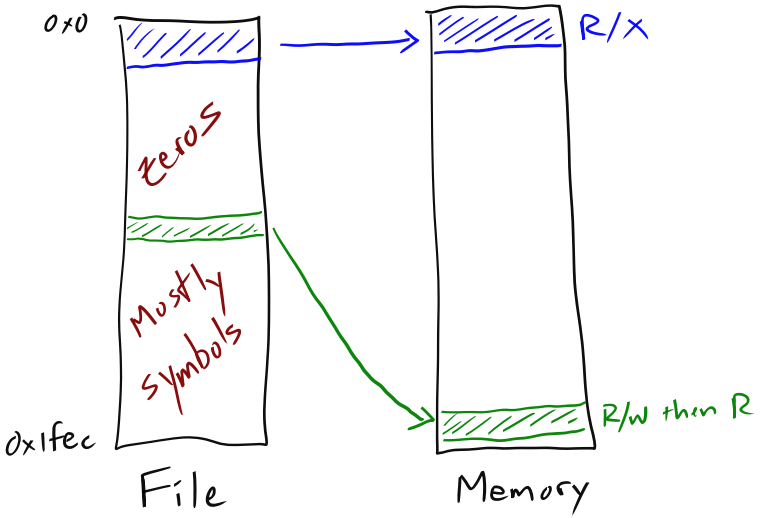
```
typedef struct
{
    Elf32_Word    p_type;     /* Segment type */
    Elf32_Off     p_offset;   /* Segment file offset */
    Elf32_Addr    p_vaddr;    /* Segment virtual address */
    Elf32_Addr    p_paddr;    /* Segment physical address
                                 (ignored for System V) */
    Elf32_Word    p_filesz;   /* Segment size in file */
    Elf32_Word    p_memsz;    /* Segment size in memory
                                 (>= p_filesz) */
    Elf32_Word    p_flags;    /* Segment flags */
    Elf32_Word    p_align;    /* Segment alignment
                                 (offset % alignment must equal
                                  vaddr % alignment)*/

} Elf32_Phdr;
```

readelf -l arm-hello-world

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  EXIDX          0x00060c 0x0000060c 0x0000060c 0x00008 0x00008 R   0x4
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120 R   0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.3]
  LOAD           0x000000 0x00000000 0x00000000 0x00618 0x00618 R E 0x10000
  LOAD           0x000f08 0x00010f08 0x00010f08 0x0013c 0x00140 RW  0x10000
  DYNAMIC        0x000f10 0x00010f10 0x00010f10 0x000f0 0x000f0 RW  0x4
  NOTE           0x000168 0x00000168 0x00000168 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000f08 0x00010f08 0x00010f08 0x000f8 0x000f8 R   0x1
```

Memory map! Write/execute permissions of memory segments!

Loading illustrated



Sections

Linker's view of program

- Sections of files that can be rearranged/merged during linking
- Information used for dynamic linking

Section header 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;
```

readelf -S arm-hello-world

```
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
...
  [ 5] .dynsym           DYNSYM          000001c4 0001c4 0000a0 10   A  6   3  4
  [ 6] .dynstr           STRTAB          00000264 000264 000086 00   A  0   0  1
...
  [12] .plt              PROGBITS        000003a4 0003a4 000050 04  AX  0   0  4
  [13] .text             PROGBITS        000003f4 0003f4 0001fc 00  AX  0   0  4
  [14] .fini             PROGBITS        000005f0 0005f0 000008 00  AX  0   0  4
  [15] .rodata           PROGBITS        000005f8 0005f8 000011 00   A  0   0  4
...
  [21] .got              PROGBITS        00011000 001000 00003c 04  WA  0   0  4
  [22] .data             PROGBITS        0001103c 00103c 000008 00  WA  0   0  4
  [23] .bss              NOBITS          00011044 001044 000004 00  WA  0   0  1
...
  [26] .symtab           SYMTAB          00000000 001098 000690 10     27  81  4
  [27] .strtab           STRTAB          00000000 001728 000337 00      0   0  1
...
```

Some common sections

.text	Executable instructions of a program
.data	Initialized, writable data
.rodata	Initialized, read-only data
.bss	Uninitialized data
.symtab	Symbols table
.strtab	Strings table (typically for .symtab entries)
.dynsym	Dynamic linking symbols table
.dynstr	Strings for dynamic linking
.got	Global offset table
.plt	Procedure linkage table

We know where executable code is! Where data is!

Section to segment mapping

```
Section to Segment mapping:
 Segment Sections...
  00     .ARM.exidx
  01
  02     .interp
  03     .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym
         .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init
         .plt .text .fini .rodata .ARM.exidx .eh_frame
  04     .init_array .fini_array .dynamic .got .data .bss
  05     .dynamic
  06     .note.gnu.build-id .note.ABI-tag
  07
  08     .init_array .fini_array .dynamic
```

Symbols

Support linking:

- Symbols used to refer to functions, variables, etc. by name
- Linker resolves symbols to match addresses to resources

Regular symbols

Used at build time, and can be discarded from the final executable or shared library

Dynamic symbols

Used for runtime dynamic linking, and cannot be discarded without breaking this process

Symbol table entries

```
typedef struct
{
    Elf32_Word    st_name;        /* Symbol name (string tbl index) */
    Elf32_Addr    st_value;       /* Symbol value */
    Elf32_Word    st_size;        /* Symbol size */
    unsigned char st_info;        /* Symbol type and binding */
    unsigned char st_other;       /* Symbol visibility */
    Elf32_Section st_shndx;       /* Section index */
} Elf32_Sym;
```

Symbol examples

```
$ readelf -s arm-hello-world
Symbol table '.dynsym' contains 10 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
     5: 00000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.4 (2)
...
Symbol table '.symtab' contains 105 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
   101: 00000570    28 FUNC    GLOBAL DEFAULT   13 main
...
```

Names, addresses, and lengths of functions and global variables!

Dynamic symbols

Support *dynamic linking* of shared-object libraries on an as-needed basis.

- Needed libraries identified in the 'dynamic' section (`readelf -d`)
- Dynamic symbols used to identify functions or variables that will be requested by a program or library
- Dynamic symbols also used to identify functions or variables that are made available by a library
- Dynamic linker identifies requested functions or variables by name

Stripped binaries

- Symbol table entries are extraneous in fully linked executables
- Symbols can take up a lot of space
- Symbols can be stripped with `strip` command
- Dynamic symbols can *not* be stripped without affecting the functionality of the executable
  - Dynamic symbols are used for runtime dynamic linking of shared object files

Consequences of stripping

	Unstripped	**Stripped**
Dynamically linked	All symbols and dynamic symbols intact	Only dynamic symbols
Statically linked	All symbols intact (no dynamic symbols)	No symbols

Finding main() in a stripped Linux ELF

- Entry address is *not* the address of main()
- Typical to call `__libc_start_main()` as first function call

```
int __libc_start_main
  (int *(main) (int, char **, char **), // Address of main()
   int argc,                            // argc
   char * * ubp_av,                     // char *argv
   void (*init) (void),                 // Initialization routines
   void (*fini) (void),                 // Termination routines
   void (*rtld_fini) (void),            // Dynamic linker's finalizer
   void (* stack_end));                 // Stack pointer (?)
```

Application examples: stripped versus unstripped

For reference, `arch/arm/tools/syscall.tbl` in the Linux kernel source tree:

```
0       common  restart_syscall          sys_restart_syscall
1       common  exit                     sys_exit
2       common  fork                     sys_fork
3       common  read                     sys_read
4       common  write                    sys_write
5       common  open                     sys_open
6       common  close                    sys_close
...
```

Finding function entry points in absence of symbols

- Defined entry points (reset/interrupt vectors)
- Targets of call instructions
- Function preamble heuristics

Bonus material

Debugging symbols

- DWARF standard format for Linux ELFs
- Encoded in various `.debug` sections
- Provides information about variable types, mapping of instructions to source code lines, etc.
- Typically quite large. Very uncommon to find these in production software.

You will probably not be this lucky

Why do these addresses keep changing?

```
int foo = 1;
int main(int argc, char **argv) {
        printf("Address of foo: %p\n", &foo);
        return 0;
}
```

```
$ ./print-pointer
Address of foo: 0x4ce044
$ ./print-pointer
Address of foo: 0x42a044
$ ./print-pointer
Address of foo: 0x475044
```

Relocations

To support relocating code (ASLR, dynamically loaded shared libraries), values in memory (especially pointers) can be specified as relocatable and will be modified at load time.

This is primarily a concern if you want to patch the binary.

Relocations

```
typedef struct
{
    Elf32_Addr  r_offset;   /* Address */
    Elf32_Word  r_info;     /* Relocation type and symbol index */
} Elf32_Rel;

typedef struct
{
    Elf32_Addr  r_offset;   /* Address */
    Elf32_Word  r_info;     /* Relocation type and symbol index */
    Elf32_Sword r_addend;   /* Addend */
} Elf32_Rela;
```

Dynamic linking mechanism

Relevant sections:

- `.plt`: Procedure linkage table
  - Stubs to look up and jump to entries in GOT
- `.got`: Global offset table
  - Addresses of external symbols filled in dynamically by linker
  - Sometimes a special GOT just for the PLT (.got.plt)

This is primarily a concern for the development of software exploits.

Example

```
(gdb) b main
(gdb) run
(gdb) x/3i $pc
=> 0x400518 <main+12>: add r3, pc
   0x40051a <main+14>: mov r0, r3
   0x40051c <main+16>: blx 0x4003cc <puts@plt>
```

The program wants to call a dynamically linked function, `puts()`. It calls the associated PLT stub.

Example

```
(gdb) x/3i 0x4003cc
   0x4003cc <puts@plt>: add r12, pc, #0, 12
   0x4003d0 <puts@plt+4>: add r12, r12, #16, 20 ; 0x10000
   0x4003d4 <puts@plt+8>: ldr pc, [r12, #3132]! ; 0xc3c
(gdb) x/1xw 0x411010
0x411010 <puts@got.plt>: 0x004003ac
```

The called PLT stub fetches an address from the GOT and jumps to it.

Example

```
gdb) x/5i 0x4003ac
   0x4003ac: push {lr} ; (str lr, [sp, #-4]!)
   0x4003b0: ldr lr, [pc, #4] ; 0x4003bc
   0x4003b4: add lr, pc, lr
   0x4003b8: ldr pc, [lr, #8]!
(gdb) x/1xw 0x4003bc
0x4003bc: 0x00010c44
```

The first time, the PLT stub invokes the dynamic linker. The GOT entry address is still stored in r12, which indicates the desired function to the linker.

Example

```
(gdb) b *0x400520
Breakpoint 2 at 0x400520
(gdb) c
Continuing.
Hello world!

Breakpoint 2, 0x00400520 in main ()
```

The dynamic linker executes `puts()` and fixes up the GOT entry behind the scenes.

Example

```
(gdb) x/1xw 0x411010
0x411010 <puts@got.plt>: 0xb6f24ab1
(gdb) x/5i 0xb6f24ab1
   0xb6f24ab1 <puts>: stmdb sp!, {r4, r5, r6, r7, r8, lr}
   0xb6f24ab5 <puts+4>: mov r6, r0
   0xb6f24ab7 <puts+6>: bl 0xb6f35380 <strlen>
   0xb6f24abb <puts+10>: ldr r5, [pc, #316] ; (0xb6f24bf8 <puts+328>)
   0xb6f24abd <puts+12>: ldr r2, [pc, #316] ; (0xb6f24bfc <puts+332>)
```

And now the `puts()` GOT entry points directly to `puts()` in `libc`!

Dynamic linking illustration