

Lecture 11: elf

(From ELF to EXE, Shebang, Static Linking and Loading, and Hacking)

Xin Liu

Florida State University
xliu15@fsu.edu

COP 4610 Operating Systems
<https://xinliulab.github.io/FSU-COP4610-Operating-Systems/>

Today's Key Question:

- Buffer Overflow Is Not Enough!
- How can we understand and exploit program execution?

Main Topics for Today:

- Executable and Linkable Format (ELF):
 - Structure, Creation, and How ELF Is Executed
 - Create Your Own ELF
- Memory Execution Process:
 - From Source Code to Execution
 - Create Your Own `execve`
- Security Implications:
 - Identifying and Addressing Vulnerabilities
 - Techniques for Secure Programming

Executable Linkable File (ELF)

Making the Program Recognizable to the Machine

Which tools recognize your helloworld?

```
$ file helloworld  
$ cat helloworld  
$ cat helloworld | hexdump | less  
$ objdump -d helloworld | less
```

First 4 bytes should be: 0x 457f 464c

That is 16-bit little-endian grouping of "0x7F E L F" and only recognized by execve.

What is an Executable File?

Before Learning OS:

- "That thing you double-click to open a window"



After Learning OS:

- An object in the operating system (a file)
- A sequence of bytes (we can edit it as characters)
- A **data structure** that describes the initial state of a state machine.

The computer is a machine.

Everything in the computer is a state machine.

Executable files describes the initial state of a process.

- Each line of assembly code represents a state transition.
- When using the system call `execve`, the initial state of the program, as defined in the ELF, is fixed.
 - There is a **document** that explicitly defines what the initial state of the program should be.

In-Class Quiz

Key Manuals for This Lesson:

- **System V ABI:** Defines the System V Application Binary Interface for the AMD64 architecture, providing essential specifications for binary compatibility.
- The answer of in-class quiz 3 
- [System V ABI \(AMD64 Architecture Processor Supplement\)](#)
- Section 3.4 Process Initialization
 - Figure 3.9 Initial Process Stack
 - Specifies certain parts of registers and memory.
 - Other states (mainly in memory) are determined by the executable file.
- **Refspecs:** Additional reference specifications to deepen understanding of Linux-based systems.
 - [Linux Refspecs](#)

What Exactly is the State of a Process?

The State of a Process:

- The process state is composed of:
 - **Memory**: Describes the program's address space and its contents.
 - **Registers**: Includes general-purpose registers and program-specific configurations.

However,

- Figure 3.9 (System V ABI) shows the **initial process stack**, but this is not part of the executable file itself.
- It is the responsibility of the operating system to construct the initial stack based on the ABI specification.

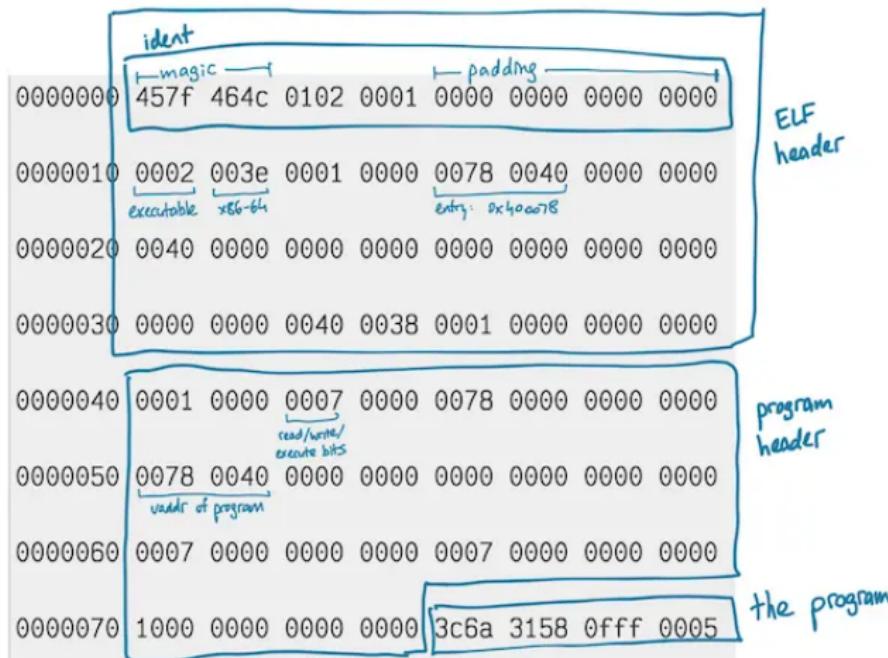
What Does the ELF Actually Define?

ELF and Memory Data Structures:

- The ELF defines **how data is structured in memory**, including both fixed and dynamic components.
 - Basic information (version, architecture, ...).
 - Memory layout (which parts contain which data).
 - Other details (debug info, symbol table, ...).
 - Dynamic libraries.
- These structures are binary and can be complex to interpret directly.
- Specialized tools like `readelf` and `objdump` are essential for reading and understanding these memory structures.

Minimal ELF

Have A Try: [elf-minimal.py](#)



All executables are byte sequences, especially when you see Ox3C (the exit system call number) and 0xF05 (syscall) in the program section.

GNU Binutils: Essential Tools for Executable Files

- **Creating Executable Files:**

- `ld` (Linker): Combines object files into a single executable.
- `as` (Assembler): Translates assembly code into machine code.
- `ar` and `ranlib`: Manage static libraries.

- **Analyzing Executable Files:**

- `objcopy`, `objdump`, `readelf`: Inspect and modify executables, often used in computer systems basics.
- `addr2line`: Maps addresses to line numbers for debugging.
- `size`, `nm`: Display size information and symbol tables.

Learn More: [GNU Binutils Official Page](#)

So, I can use the command `size` to determine the smallest 'Hello World' program from each student's HW2 and give extra credit to

the one with the smallest.



The Minimal HelloWorld in HW 2

```
.global _start
_start:
    mov $1,%eax
    mov %eax,%edi
    lea m(%rip),%rsi
    mov $13,%dl
    syscall
    mov $60,%al
    dec %edi
    syscall
m:.ascii "Hello, World!"
```



2025

```
$ size a.out
      text      data      bss      dec      hex filename
      37          0          0      37      25 a.out
```

1 Extra Credit!

The Minimal HelloWorld in HW 2

```
.global _start
.section .rodata
st:
    .ascii "Hello, World!"
ed:
    .section .text
_start:
    mov    $1, %rax
    mov    $1, %rdi
    lea    st(%rip), %rsi
    mov    $ed - st, %rdx
    syscall
    mov    $60, %rax
    xor    %rdi, %rdi
    syscall
```



2024

```
$ size a.out
```

text	data	bss	dec	hex	filename
55	0	0	55	37	a.out

Strategy 1: Compact Instruction Encoding

The minimalist code uses tricks to generate shorter machine code for each instruction:

- **Smaller Registers:** Uses 32-bit (%eax, %edi) and 8-bit (%dl, %al) registers. Instructions operating on these are often shorter than their 64-bit (%rax, %rdi) counterparts.
- **Hardcoded Length:** Uses a tiny, immediate value for the string length (`mov $13, %dl`), which is a very short instruction.
- **Clever Zeroing:** Uses `dec %edi` (a 1-byte instruction) to change the exit code from 1 to 0, which is much shorter than the standard `xor %rdi, %rdi` (a 2 or 3-byte instruction).

Strategy 2: Non-Standard Code and Data Layout

Standard programs separate code and data into different memory sections.

- **Conventional:** The 55-byte version correctly places the string in the `.rodata` (read-only data) section and instructions in the `.text` section. This separation has alignment and management overhead, increasing size.
- **Minimalist:** The 37-byte version "cheats" by placing the string data *directly inside* the executable `.text` section. This avoids the overhead of managing a separate data section.

Why Can We See All This Information?

Debugging Information Added During Compilation:

- When we compile with debug flags, the compiler includes extra information in the binary.
- This information allows tools like `objdump` and `addr2line` to map assembly code back to the original source code.

Example Command:

- Using `gcc -g -S hello.c` generates assembly code with debugging information.
- This enables us to see additional sections in the assembly output, including variable names, line numbers, and other metadata.

Standard of Debugging Information

Mapping Machine State to “C World” State:

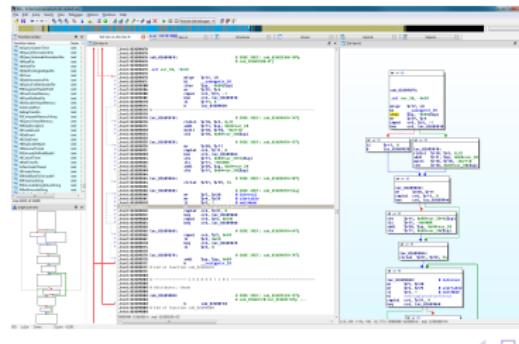
- The DWARF Debugging Standard (dwarfstd.org) defines an instruction set, DW_OP_XXX, that is Turing Complete.
- This instruction set can perform “arbitrary computations” to map the current machine state back to the C language state.

Challenges and Limitations:

- **Limited Support for Modern Languages:** Advanced features (e.g., C++ templates) are not fully supported.
- **Complexity of Programming Languages:** As languages evolve, it becomes increasingly challenging to accurately map machine states to source code.
- **Compiler Limitations:** Compilers may not always produce perfect debug information, leading to issues like:
 - Frustrating instances of variables being <optimized out>
 - Incorrect or incomplete debugging information

Reverse Engineering

- Provides insights into commercial software without access to the original source code.
- Challenges:
 - No Debug Information
 - Stripped Symbols
 - Opaque Instruction Sequences
- Techniques:
 - Analysts use specialized tools (e.g., objdump, IDA Pro, Ghidra) to disassemble and analyze the instruction sequences.
 - Techniques like pattern recognition, control flow analysis, and heuristic methods help infer program functionality.



Funny Little Executable

Let's create our own ELF file from scratch.

Why is learning ELF so challenging?

No difference for you!

```
$ readelf -a helloworld  
$ cat helloworld
```

Reflection:

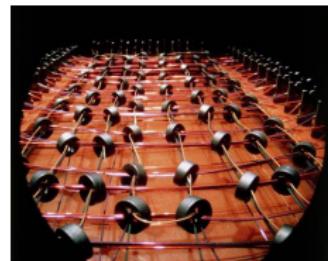
- ELF is not a human-friendly "state machine data structure."
- For the sake of performance, it sacrifices readability, violating the principle of "information locality."

Almost Like Reading a Core Dump:

- "Hell's joke: Today's core dump is an ELF file."

Magnetic Core Memory

- The origin of "Segmentation fault (core dumped)"
- Non-volatile memory!



Magnetic core memory, storing data by the magnetization direction of tiny ferrite cores. Each core represents a single bit, retaining data even when powered off.

But It Wasn't Always Like This

UNIX a.out "assembler output"

- A relatively simple data structure
- Describes the initial state (structure) of the address space
- Once the data is loaded into the process and the pointer is set to the entry point, the program can start running.

```
struct exec {  
    uint32_t a_midmag; //  
    Machine ID & Magic  
    uint32_t a_text; // Text  
    segment size  
    uint32_t a_data; // Data  
    segment size  
    uint32_t a_bss; // BSS  
    segment size  
    uint32_t a_syms; // Symbol  
    table size  
    uint32_t a_entry; // Entry  
    point  
    uint32_t a_trsize; // Text
```

The More Features Supported, the Less Human-Friendly:

- Hearing terms like "program header," "section header" feels overwhelming for the human brain.
- Contains cryptic values like R_X86_64_32, R_X86_64_PLT32.
- A massive amount of "pointers" (essentially unreadable to humans).
 - LLM can help us read them, but it's still far from easy!

A More Human-Friendly Approach:

- Simpler and flatter design is easier to understand.
- All necessary information is immediately visible.

Design Your Own FLE:

- **FLE:**
 - Funny (Fluffy) Linkable Executable
 - Friendly Learning Executable (my favorite!)
 - Have A Try: [fle](#)

Core Design Principles:

- Make everything human-readable (all information should be at the top).
- Revisit the core concepts of linking and loading: code, symbols, relocations.
- How would you design it?

Let's use emojis!

Code , Symbols , and Relocations

By combining these three elements, we can create an executable file!

: ff ff ff ff ff ff ff ff

: ff ff ff ff ff ff ff ff

: _start

: 48 c7 c0 2a 00 00 00

: 48 c7 c7 2a 00 00 00

: 0f 05 ff ff ff ff ff ff

: ff ff ff ff ff ff ff ff

 : i32(unresolved_symbol - 0x4 - 

- You can use text to hack the executable file.
- You can also get the debugging information.

Implementation of FLE Binutils

Implemented Tools:

- exec (loader)
- objdump/readfle/nm (display)
- cc/as (compiler/assembler)
- ld (linker)

Most Components Reuse GNU Binutils:

- elf_to_fle

Step 1: Preprocessing and Compilation

Source Code (.c) → Intermediate Code (.i):

- Ctrl-C & Ctrl-V (#include)
 - GCC first performs a preprocessing step without macros
 - `gcc -E foo.c`
- String substitution
- Today: We use [macros](#)

Intermediate Code (.i) → Assembly Code (.s):

- `gcc -S foo.c`
- Translation from "high-level state machine" to "low-level state machine"
- Final output: annotated instruction sequences

Generating Executable Files (2): Compilation

Assembly Code (.s) → Object File (.o):

- `gcc -c foo.c`
- File = sections (`.text`, `.data`, `.rodata`, `.str1`, ...,)
 - For ELF, each section has its own permissions and stores corresponding information.
- Three key elements in a section:
 - **Code:** Sequence of instructions.
 - **Symbols:** Marks the location of "current."
 - **Relocations:** Values that cannot be determined yet (resolved during linking).

So how do we convert an ELF file into FLE? It is simple:
Read its object file, extract the useful information (symbols, data, relocations), and pack it into an FLE file that we can easily read.

Generating Executable Files (3): (Static) Linking

Multiple Object Files (.o) → Executable File (a.out):

- Combine all sections:
 - Merge code from .text, .data, .bss, etc.
 - Flatten sections into a linear sequence.
 - Determine the locations of all symbols.
 - Resolve all relocations.
- Produce a single **executable file**:
 - A description of the program's initial memory state.

FLE Program: Loading

Load the "byte sequence" into memory:

- That's all there is to do.
- Then set the correct PC (program counter) and start running.

```
mem = mmap.mmap(
    fileno=-1, length=len(bs),
    prot=mmap.PROT_READ | mmap.
    PROT_WRITE | mmap.PROT_EXEC,
    flags=mmap.MAP_PRIVATE | mmap.
    MAP_ANONYMOUS,
)
mem.write(bs)
mem.flush()
call_pointer(mem, file['symbols']['
_start'])
```

Shebang

#! - Shebang

Easter Egg:

- Our FLE files can be executed directly:

```
#!/./exec
```

The "magic" of #! in UNIX:

- Example: file.bin

```
#!A B C
```

- The operating system executes:

```
execve(A, ["A", "B C", "file.bin"], envp)
```

- The shebang (# !) on the first line of a text script tells the kernel which interpreter to run.
- When you execute the script, the kernel runs that interpreter and passes the script path as an argument, so you do not need to type the interpreter manually.
- You can include arguments after the interpreter, e.g.,
`#!/usr/bin/env python3 -O`.
- This applies to text scripts with execute permission, not native binaries.
 - Must have execution ('x') permission: `chmod +x`
 - Must be in a format that the loader can recognize as executable.

Who Decides If a File is Executable?

The Operating System (Syscall - execve) Determines Executability:

- The OS, through `execve`, decides whether a file can be executed.

Try It Out:

- Use `strace` to trace `execve` calls and observe execution failures.
 - `strace ./a.c`
 - Without execute permission on `a.c`: `execve` returns `-1, EACCES`
 - With execute permission but incorrect format on `a.c`: `execve` returns `-1, ENOEXEC`

Example: Running Python Code in a C File

- Save the Following Code as helloworld.c:

```
#! /usr/bin/python3  
print("Hello World!")
```

- Give the file execute permission:

```
$ chmod +x helloworld.c
```

- Now, you can directly run the helloworld.c file to execute the Python code:

```
$ ./helloworld.c  
Hello World!
```

Static Linking and Loading

Why ELF When We Have FLE?

If you want to build Chrome (2017):

- 2 GiB binary (with debug info)
- 17,000 files
- 1,800,000 sections
- 6,300,000 symbols
- 13,000,000 relocations

C++ Name Mangling:

- Example: `_ZNK8KxVectorI6DlTypejEixEj` is:
`KxVector<DlType, unsigned int>::operator[] (unsi`
- (It seems impossible to skip pointers.)

How do we manage a large binary?

- We design data structures.
- A data structure! A big data structure! A super big data structure!
- ELF defines the standard for these structures in executables, object files, and shared libraries.
- It tells tools and the kernel how bytes are laid out on disk and in memory.

Static Linking: ELF vs. FLE

- **Idea:** FLE is a minimal model for static linking. ELF is the full specification used in real systems.
- **What ELF adds beyond FLE:**
 - Rich sections and flags (e.g., .text, .rodata, .data, .bss, .tdata).
 - More symbol kinds and visibility rules.
 - Many relocation types (e.g., PCREL, GOT/PLT, TLS like TPOFF32).
- **Mental model:** FLE links by “symbols + relocations.” ELF does the same idea with more sections and more relocation rules.
- **How to design FLE:** Inspect with `readelf -a` and `objdump -x/-d`. Then map the fields to the simpler FLE concepts.

FLE Loader: Does Only One Thing

- Copies a single byte sequence into the address space:
 - Grants read, write, and execute permissions.
- Then jumps to `_start` for execution.

ELF: Not Much More

- Copies multiple segments into the address space:
 - Separately grants read, write, and execute permissions.
- Then jumps to the specified entry point (default: `_start`) for execution.

They Are Both Data Structures

- Example: ELF is a "binary data structure."
- `readelf -l` describes how it is loaded:
 - **Offset:** Segment's offset in the file.
 - **VirtAddr:** Virtual address where the segment is loaded in memory.
 - **PhysAddr:** Physical address (rarely used).
 - **FileSize:** Number of bytes in the segment in the file.
 - **MemSize:** Number of bytes in the segment in memory (may exceed file size).
 - **Flags:** Permissions, such as RWE (Read, Write, Execute).
 - **Align:** Alignment of the segment's virtual address.

Have A Try: [loader](#)

- **Load ELF:** open file, map Elf64_Ehdr, walk PT_LOAD.
- **Map segments:** use mmap with PF_R/W/X, map file bytes, then map anonymous space for .bss.
- **Build stack:** allocate a local stack buffer, follow System V ABI.
Push argc, argv[], a null, envp[], a null, then auxv with AT_RANDOM, end with AT_NULL.
- **Jump to entry:** set %rdx = 0 for ABI, set %rsp to our stack, jump to e_entry.

We hand-built an execve without calling execve.
The OS is simply a machine and there is no magic as long as you follow the ELF format and the ABI specification.

Understanding Executable Files and Buffer Overflow

- What is an Executable File?
 - An executable file is a data structure (a sequence of bytes) that describes the initial state of a state machine.
 - The loader transfers this "initial state" into the operating system.
 - It is difficult to read because it was never designed for human readability.
- It helps us understanding the buffer overflow:
 - Why can we use `gdb` to compute stack offsets that helps analyze function call stack structures?
 - Observing local variables, return addresses, and how an overflow can overwrite the return address.
 - Redirecting execution to malicious code (e.g., shellcode) reveals how control flow is hijacked.
 - This process provides insight into program execution, stack management, and security vulnerabilities.