# CSci 5980/8980 Manual and Automated Binary Reverse Engineering Slides 5: The ELF Binary File Format

Stephen McCamant University of Minnesota

#### Outline

**ELF** basics

Static and dynamic linking

# Executable/object file formats

- Modern systems usually use a common format for relocatable object files during compilation and final executables
- Mostly binary data representing code and data
- Plus metadata allowing the data to be linked and loaded

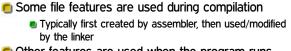
# Brief history of binary file formats (Unix)

- Early Unix had a simple a.out format
  Lasted until early days of free Linux/BSD, now obsolete
  AT&T's second try was named COFF
- Still limited, but widely adopted with changes
- AT&T's third try was ELF, now used in almost all Unix systems

#### Brief history of binary file formats (non-Unix)

- Early DOS and Windows had several limited formats
- Since the 32-bit era, Windows uses the PE (Portable Executable) format
  - Partially derived from COFF
- OS X era Apple (including iOS, etc) uses a format named Mach-O
  - First developed for the Mach microkernel used on the NeXT

# Compile-time and run-time

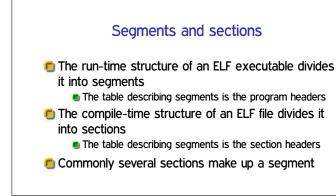


- Other features are used when the program runs
  - By the OS when the program starts
  - And now also by runtime linking

# Static and dynamic/shared linking Traditional "static" linking happens all at compile time Libraries become indistinguishable from the rest of the program For efficiency and flexibility, it is now more common to postpone library linking until runtime At startup, or later in execution Library code stays separate, so its memory can be shared

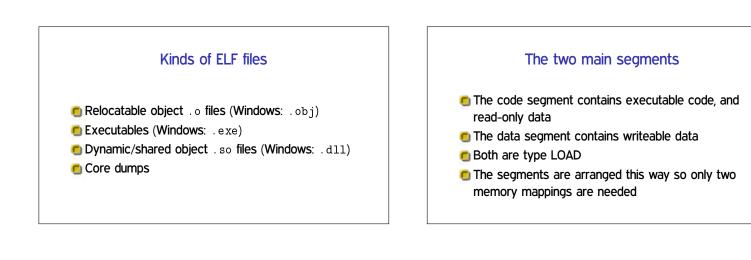
#### ELF

- Executable (or Extensible) and Linking (or Linkable) Format
- Eirst appeared in System V Release 4 Unix, c. 1989
- Linux switched to ELF c. 1995
  - In part because they'd chosen a hard-to-use approach to a.out shared libraries
  - BSD switched later, c. 1998-2000



### High-level ELF file structure

- A fixed-size header with a magic number \x7fELF and basic information
- The program headers
- Code and data that are loaded when the program runs
- Data that isn't normally loaded, like debugging symbols
- The section headers

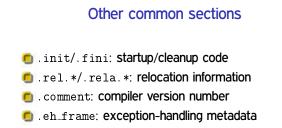


#### Other common segments

- INTERP: holds pathname of dynamic loader
- DYNAMIC: information used by dynamic linking
- (GNU\_)STACK: specifies stack permissions
- NOTE: miscellaneous data; in core dumps, register values

#### The main sections

- 🖲 . text: most code
- I rodata: read-only data like string constants
- 🦲 . data: initialized data (values stored in file)
- bss: zero-initialized data (zeros not stored)

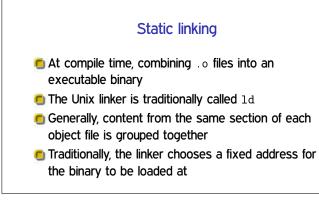


debug\_\*: debugging information

Outline

**ELF** basics

Static and dynamic linking



## Static linking vs. PIE

- Standard fixed addresses:
  - **x86-32: Starting at** 0x08480000
  - **x86-64**: Code at 0x400000, data at 0x600000
- Recent systems default to making even the main executable position independent
  - PIE = position-independent executable
- PIE binaries look like shared libraries, and look on disk like they start at address 0
  - At run-time, these offsets added to a random base

# Static libraries

- Unix static libraries end in .a, and are just archives of .o files
  - Program ar was once a relative of tar
- The . o file are the unit of code inclusion So often one per API function
- Transitive requirements and ordering are not automatic

#### Relocation

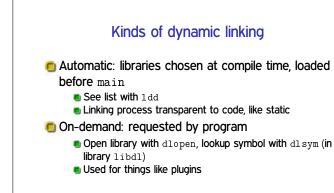
- Content in .o files must be fixed up when final locations chosen
- The relocation table tells the linker how
  - Gives location, target symbol, machine-specific type
    An additional offset ("addend") may be stored in the original bytes or in the table
- Relocations are always size-preserving

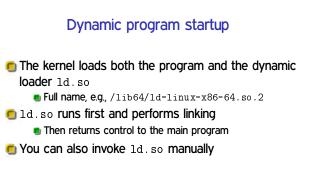
# Symbol table

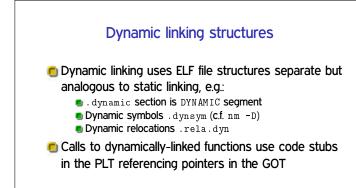
- ELF files that define symbols list them in a symbol table section .symtab
- 🖲 Can examine with nm as well as objdump
- By default, finished executables include the symbol table
  - But it is removed by strip

# Static program startup

- Static programs are loaded just by the kernel, and fairly simple
- Code and data regions are mapped as if by mmap (demand paged)
- Stack is initialized with arguments, environment variables, and auxiliary vector auxv
- Execution starts at the entry point







### The PLT and GOT

The Procedure Linkage Table (PLT) contains a code stub for each called function from an external library
 The static linker makes calls to, e.g., printf@plt, in place of a static function address
 PLT stubs reference function pointers stored in the Global Offset Table (GOT)

E.g. a pointer holding the location of printf in the C library

#### Lazy resolution vs. RELRO

- To save startup time, symbol lookup for a function is often delayed until the first call, "lazy"
- On the other hand, dynamic linker structures are useful for attackers

Writeable function pointers with a standard layout

- RELRO (relocation read-only) configurations make more dynamic linking state read-only after startup
- Full RELRO disables lazy resolution