CS-527 Software Security

Reverse Engineering

Asst. Prof. Mathias Payer

Department of Computer Science
Purdue University

TA: Kyriakos Ispoglou

https://nebelwelt.net/teaching/17-527-SoftSec/

Spring 2017

Table of Contents

Compilation: C source

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  if (argc == 2)
    printf("Hello %s\n", argv[1]);
  return 0;
}
```

How much code is generated? How complex is the executable?

`gcc -W -Wall -Wextra -Wpedantic -O3 -S hello.c`

Compilation: assembly

```
1   .file "hello.c"
2   .section .rodata.str1.1,"aMS",@progbits,1
3 .LC0:
4   .string "Hello %s\n"
5   .section .text.startup,"ax",@progbits
6   .p2align 4,,15 .globl main .type main, @function
7 main: .LFB24:
8   .cfi_startproc
9   cmpl $2, %edi
10  je .L6
11  xorl %eax, %eax
12  ret
13 .L6:
14  pushq %rax
15  .cfi_def_cfa_offset 16
16  movq 8(%rsi), %rdx
17  movb $1, %dil
18  movl $.LC0, %esi
19  xorl %eax, %eax
20  call __printf_chk
21  xorl %eax, %eax
22  popq %rdx
23  .cfi_def_cfa_offset 8
24  ret
25  .cfi_endproc
26 .LFE24:
27  .size main, .-main
28  .ident "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"
29  .section .note.GNU-stack,"",@progbits
```

Assembly magic

```
For ELF targets, the .section directive is used like this:
     .section name [, "flags"[, @type[,flag_specific_arguments]]]

a     section is allocatable
e     section is excluded from executable and shared library.
w     section is writable
x     section is executable
M     section is mergeable
S     section contains zero terminated strings
G     section is a member of a section group
T     section is used for thread-local-storage
?     section is a member of the previously-current section's group, if any

@progbits       section contains data
@nobits         section w/o data (i.e., only occupies space)
@note           section contains non-program data
@init_array     section contains an array of ptrs to init functions
@fini_array     section contains an array of ptrs to finish functions
@preinit_array  section contains an array of ptrs to pre-init functions
```

More assembly magic

.global (or .globl) makes the symbol visible to ld. If you define symbol in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, symbol takes its attributes from a symbol of the same name from another file linked into the same program.

.type name , type description
This sets the type of symbol name to be either a function symbol or an object symbol.
* STT_FUNC function
* STT_GNU_IFUNC gnu_indirect_function
* STT_OBJECT object
* STT_TLS tls_object
* STT_COMMON common
* STT_NOTYPE notype

More details are available in the as manual: https://sourceware.org/binutils/docs/as/.

Compilation: linking

```
0000000000400470 <main>:
  400470:        83 ff 02                cmp    $0x2,%edi
  400473:        74 03                   je     400478 <main+0x8>
  400475:        31 c0                   xor    %eax,%eax
  400477:        c3                      retq
  400478:        50                      push   %rax
  400479:        48 8b 56 08             mov    0x8(%rsi),%rdx
  40047d:        40 b7 01                mov    $0x1,%dil
  400480:        be 04 06 40 00          mov    $0x400604,%esi
  400485:        31 c0                   xor    %eax,%eax
  400487:        e8 d4 ff ff ff          callq  400460 <__printf_chk@plt>
  40048c:        31 c0                   xor    %eax,%eax
  40048e:        5a                      pop    %rdx
  40048f:        c3                      retq
```

What is all the other machine code in the file?
What about all the other code in `objdump -d a.out`?

Start file

```
0000000000400470 <main>: ...
0000000000400490 <_start>:
  400490:        31 ed                    xor    %ebp,%ebp
  400492:        49 89 d1                 mov    %rdx,%r9
  400495:        5e                       pop    %rsi
  400496:        48 89 e2                 mov    %rsp,%rdx
  400499:        48 83 e4 f0              and    $0xfffffffffffffff0,%rsp
  40049d:        50                       push   %rax
  40049e:        54                       push   %rsp
  40049f:        49 c7 c0 f0 05 40 00     mov    $0x4005f0,%r8
  4004a6:        48 c7 c1 80 05 40 00     mov    $0x400580,%rcx
  4004ad:        48 c7 c7 70 04 40 00     mov    $0x400470,%rdi
  4004b4:        e8 87 ff ff ff           callq  400440 <__libc_start_main@plt>
  4004b9:        f4                       hlt
  4004ba:        66 0f 1f 44 00 00        nopw   0x0(%rax,%rax,1)
...
00000000004004c0 <deregister_tm_clones>: ...
00000000004004f0 <register_tm_clones>: ...
0000000000400530 <__do_global_dtors_aux>: ...
0000000000400550 <frame_dummy>: ...
0000000000400580 <__libc_csu_init>: ...
00000000004005f0 <__libc_csu_fini>: ...
00000000004005f4 <_fini>: ...
```

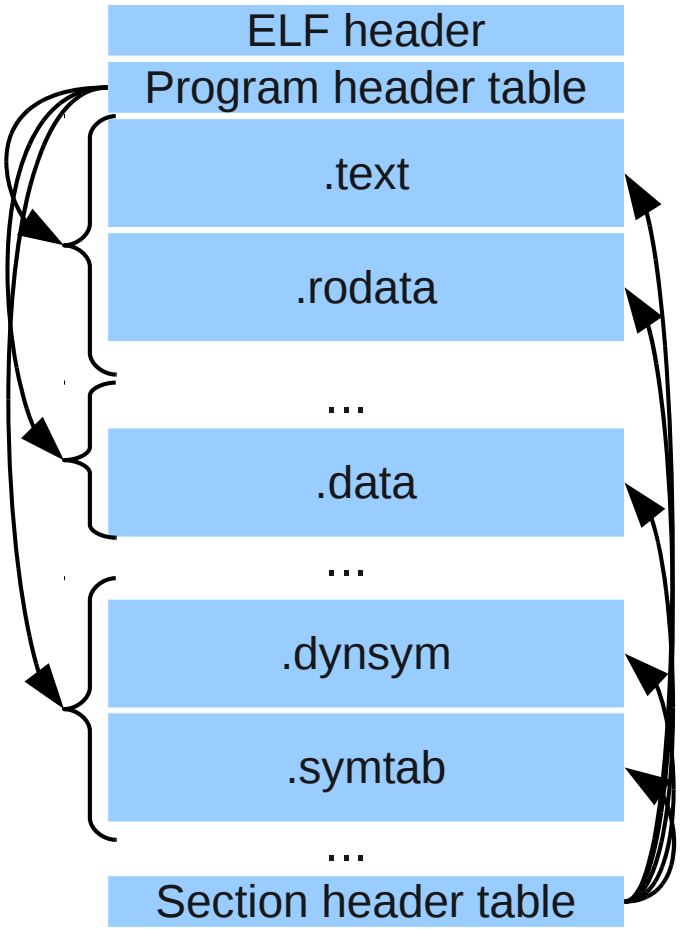
What's the format of an executable?

Executable formats

- Executable format allows a loader to instantiate a program.
- Programs then execute machine code directly and interface with the runtime system (OS).
- Loader may be a program or part of the operating system.
- Executable formats evolved, many different formats exist.
- DOS/Windows executables evolved from COM files that were restricted to 64KB to EXE files executing in 16-bit mode to 32-bit and 64-bit Windows executables.
- On Unix, ELF (Executable and Linkable Format) is common.

Non comprehensive list of executable formats: `https://en.wikipedia.org/wiki/Comparison_of_executable_file_formats`.

ELF format



- ELF allows two interpretations of each file: sections and segments.
- Segments contain permissions and mapped regions. Sections enable linking and relocation.
- OS checks/reads the ELF header and maps individual segments into a new virtual address space, resolves relocations, then starts executing from the start address.
- If `.interp` section is present, the interpreter loads the executable (and resolves relocations).

Details: `http://www.skyfree.org/linux/references/ELF_Format.pdf`.

ELF magic

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  90 04 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  98 11 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1e 00 1b 00  |....@.8...@.....|
```

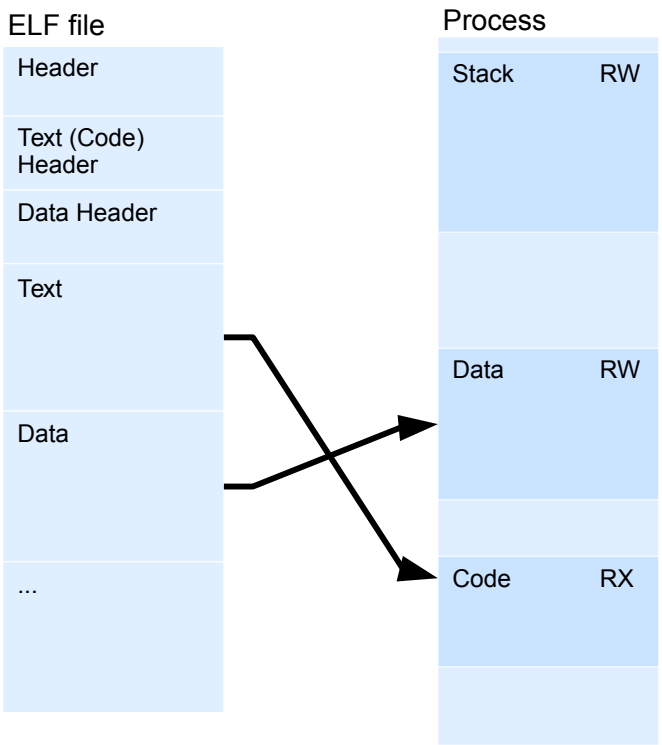
Offset	Field	Purpose
0x00	Magic	Always 0x7f ELF
0x04	Class	32-bit (0x1) or 64-bit (0x2) executable.
0x05	Data	Little (0x1) or Big (0x2) endian (starting at 0x10).
0x06	Version	0x01
0x07	Ident	Identifies system ABI, mostly 0x00 (System V).
0x08	ABI	ABI version, unused in Linux.
0x09	Pad	7b padding.
0x10	Type	Relocatable (0x01), Executable (0x02), Shared (0x03), or Core (0x04).
0x12	ISA	Specifies ISA: not specified (0x0000), x86 (0x0003), or x86-64 (0x003e).
0x14	Version	0x00000001
0x18	Entry	Entry point for executable.
0x20	PHOff	Program header offset.
0x28	SHOff	Segment header offset.
0x30	Flags	Depends on target architecture.
0x34	Heads	Header size, program header size, number of entries in program header, section header size, number of entries in section header, index in section header that contains section names (shstrndx), 2 bytes each.

ELF tools

- `readelf` and `objdump` can display information about ELF files (executables, shared objects, archives, and object files).
- `readelf -h a.out` displays basic information about ELF header.
- `readelf -l a.out` displays program headers, used by loader to map program into memory.
- `readelf -S a.out` displays sections, used by loader to relocate and connect different parts of the executable.

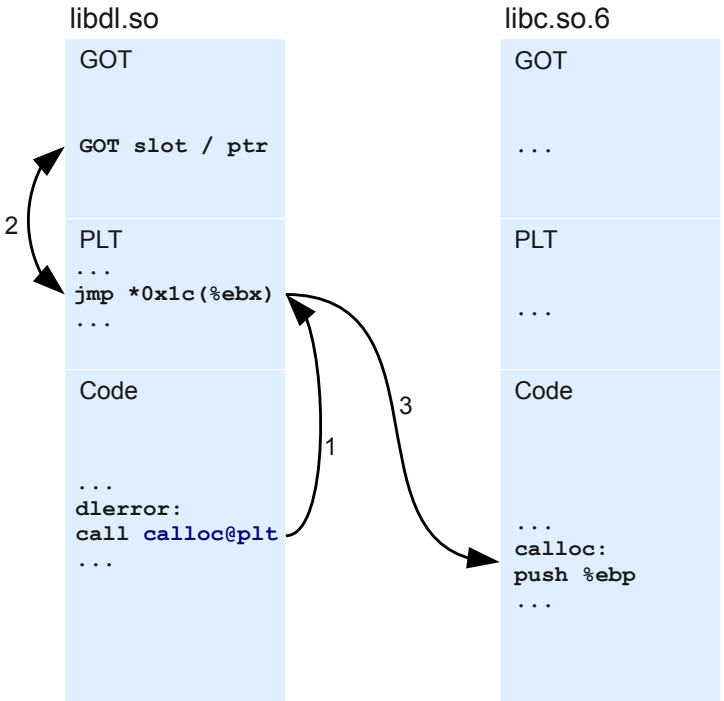
Table of Contents

In core view of executable



- Loader maps runtime sections of all shared objects into virtual address space.
- The loader calls global initialization functions of all libraries.
- Why is the order important?
- Libc initializes heap data structures and uses the `sbrk` and `brk` system calls to set up space for the memory allocator.

Shared libraries



- Global Offset Table contains pointers to symbols in other shared objects.
- Procedure Linkage Table contains code that transfers control through the GOT to a symbol in another shared object.
- The entries in the GOT that point to functions are initialized with the loader's address to resolve it on-the-fly.

Program inspection using GDB

- GDB is the default debugger on Unix/Linux machines and part of the GNU toolchain.
- `gdb ./a.out` to start the debugger.
- `disas symbol` to disassemble a symbol (e.g., main).
- `break *0xfoo` to set a break point.
- `r` to run, `c` to continue, and `si` to step execution.
- `bt` prints a backtrace of the stack (either frame pointer or debug information is needed).
- Use `print`, `printf`, `x` to evaluate information.
- `info registers` displays current register file.
- Use `set` to update memory cells or variables.

Table of Contents

Reverse engineering

```
1  400709   push   %rbx
2  40070a   mov    %rdi,%rbx
3  40070d   test   %rdi,%rdi
4  400710   je     400731 <print+0x28> @1# loop@1
5  400712   mov    0x8(%rbx),%edx
6  400715   mov    $0x400954,%esi
7  40071a   mov    $0x1,%edi
8  40071f   mov    $0x0,%eax
9  400724   callq  4005b0 <__printf_chk@plt>
10 400729   mov    (%rbx),%rbx
11 40072c   test   %rbx,%rbx
12 40072f   jne    400712 <print+0x9>
13 400731   mov    $0x400958,%esi
14 400736   mov    $0x1,%edi
15 40073b   mov    $0x0,%eax
16 400740   callq  4005b0 <__printf_chk@plt>
17 400745   pop    %rbx
18 400746   retq
```

Structure used in the code above:

```
1  struct node {
2    struct node *next;
3    int data;
4  };
```

Reverse engineering

```
1  400709   push   %rbx                    # spill register
2  40070a   mov    %rdi,%rbx               # first argument
3  40070d   test   %rdi,%rdi
4  400710   je     400731 <print+0x28>     # loop
5  400712   mov    0x8(%rbx),%edx          # load rbx[8]
6  400715   mov    $0x400954,%esi          # "%d "
7  40071a   mov    $0x1,%edi               # fortify level (format string protection)
8  40071f   mov    $0x0,%eax
9  400724   callq  4005b0 <__printf_chk@plt>
10 400729   mov    (%rbx),%rbx             # rbx = *rbx
11 40072c   test   %rbx,%rbx               # rbx == NULL?
12 40072f   jne    400712 <print+0x9>      # loop
13 400731   mov    $0x400958,%esi          # "\n"
14 400736   mov    $0x1,%edi
15 40073b   mov    $0x0,%eax
16 400740   callq  4005b0 <__printf_chk@plt>
17 400745   pop    %rbx                    # unspill
18 400746   retq
```

This looks like we are iterating over a linked list:

```
1 struct node {
2   struct node *next;
3   int data;
4 };
```

Source

```
1  struct node {
2    struct node *next;
3    int data;
4  };
5
6  void print(struct node *ptr) {
7    while (ptr != NULL) {
8      printf("%d ", ptr->data);
9      ptr = ptr->next;
10   }
11   printf("\n");
12 }
```

Table of Contents

Summary

- Program instantiation turns an executable into a process.
- Dynamic loading allows us to reuse code in libraries.
- The dynamic loader resolves all dependencies, links shared objects, and resolves relocations.
- Data structures are lost during compilation, recovery of data structures is hard.

Questions?

?