Архитектура компьютера и операционные системы. Лекция 3.

23 сентября 2021г.

Руслан Савченко

Школа Анализа Данных

Стек вызовов

Фреймы

- Регистр `%ebp` указывает на начало фрейма
- Фрейм - область, где хранятся локальные переменные функции
- Под каждую переменную выделено конкретное место на фрейме, с константным смещением от начала фрейма
- Когда нужно обратиться к переменной, используется

—offset(%rbp)

- Регистр `%esp` указыват на верхушку стека, где могут находиться промежуточные результаты вычислений.

Фреймы

Указатель на фрейм не меняется при выполнении `call`, программа должна его подвинуть сама:

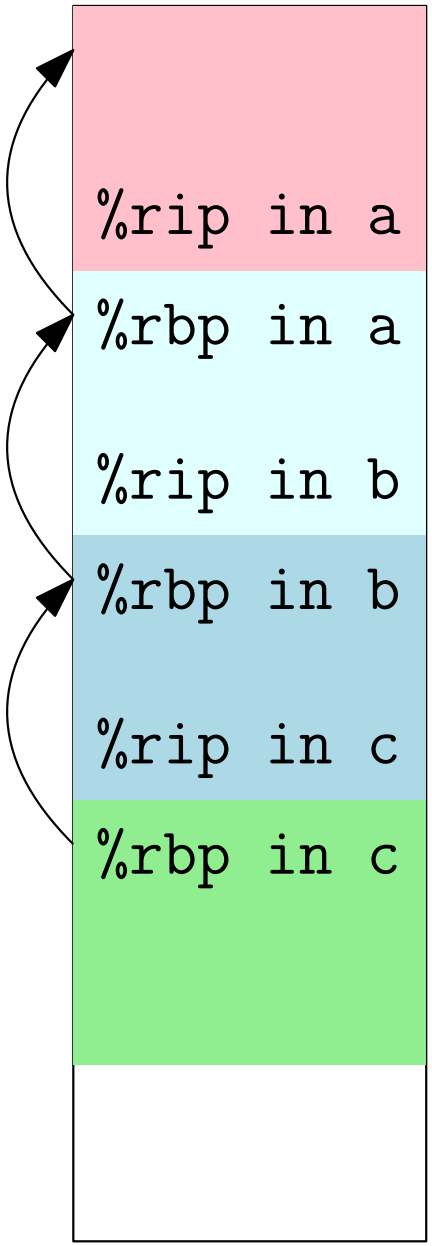
```
func:
        pushq   %rbp
        movq    %rsp, %rbp
        ...
```

Фреймы

Figure 3.3: Stack Frame with Base Pointer

Position	Contents	Frame
`8n+16(%rbp)`	memory argument eightbyte n	Previous
	...	
`16(%rbp)`	memory argument eightbyte 0	
`8(%rbp)`	return address	Current
`0(%rbp)`	previous `%rbp` value	
`-8(%rbp)`	unspecified	
	...	
`0(%rsp)`	variable size	
`-128(%rsp)`	red zone	

Стек вызовов



Stack in `a`

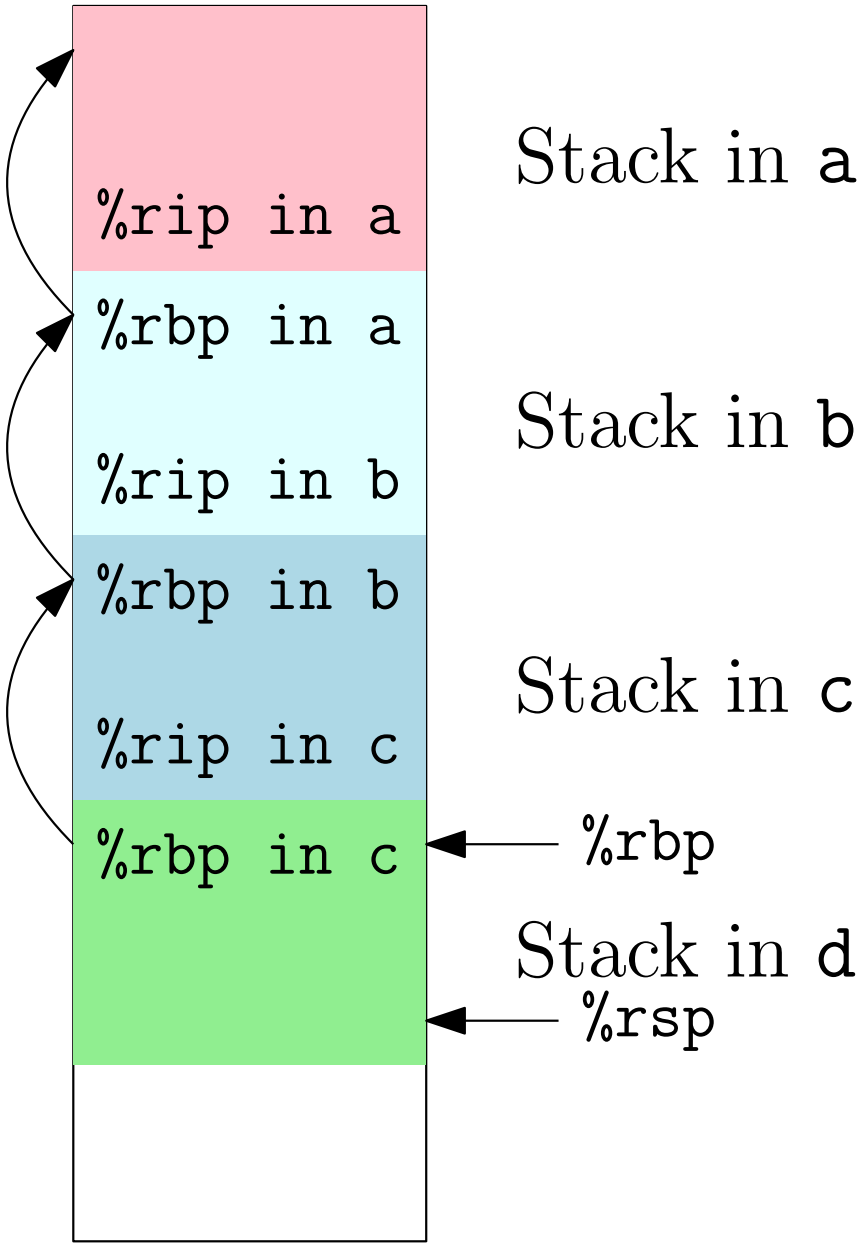
Stack in `b`

Stack in `c`

Stack in `d`

```
call a
call b
call c
call d
```

Стек вызовов




```
call a
call b
call c
call d
```

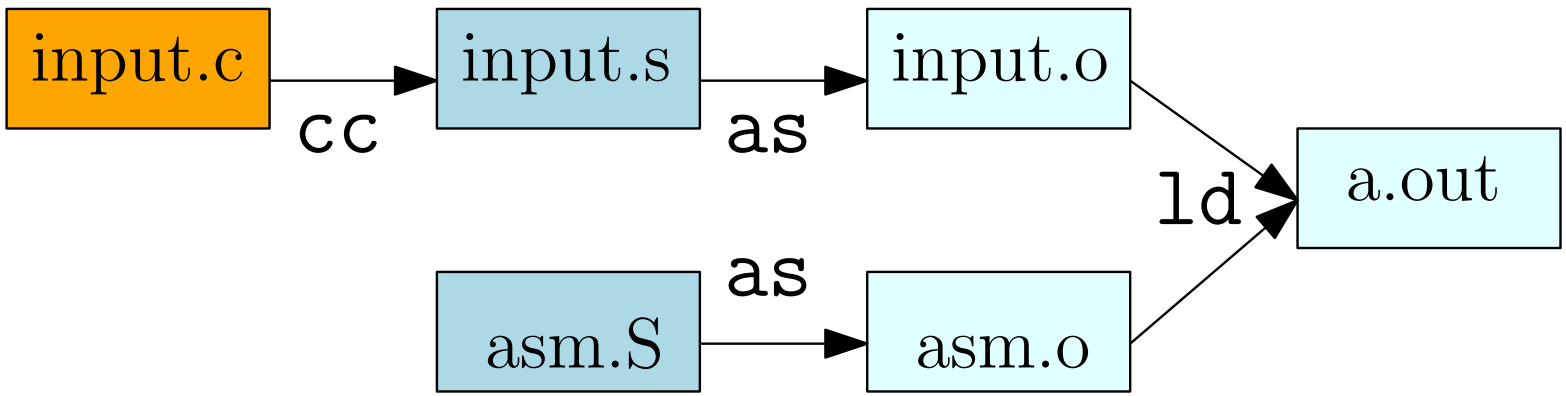
Раскрутка стека

- Указатели на фреймы можно использовать для раскрутки стека назад:
  - `%rbp` указывает на значение `%rbp` предыдущего фрейма
  - Прямо над `(%rbp)` сохранён адрес возврата
- Стоит компилировать с `-fno-omit-frame-pointer`

```
(gdb) backtrace
#0  fib (a=17) at fib.c:3
#1 0x000055555554669 in fib (a=18) at fib.c:3
#2 0x000055555554669 in fib (a=19) at fib.c:3
#3 0x000055555554669 in fib (a=20) at fib.c:3
#4 0x000055555554696 in main () at fib.c:6
```

Линковка

Пайплайн компиляции



Глобальные переменные

x.c

```
int x;
```

main.c

```
extern int x;
int main() {
    return x;
}
```

Глобальные переменные

gcc -O1 -o main main.c x.c
objdump -D main

```
00000000000004f0 <main>:
 4f0: 8b 05 1e 0b 20 00  mov  0x200b1e(%rip),%eax  # 201014 <x>
 4f6: c3                 retq

0000000000201014 <x>:
  201014:       00 00
```

0x201014 = 0x4f6 + 0x200b1e

Глобальные переменные

gcc -O1 -c main.c
objdump -d main.o

```
0000000000000000 <main>:
 0: 8b 05 00 00 00 00 mov  0x0(%rip),%eax  # 6 <main+0x6>
 6: c3                retq
```

Релокации

- Адреса глобальных переменных не известны на этапе трансляции отдельного юнита
- В машинном коде под эти адреса оставляются специальные дырки - релокации
- Релоакции применяются на этапе линковки

Глобальные переменные

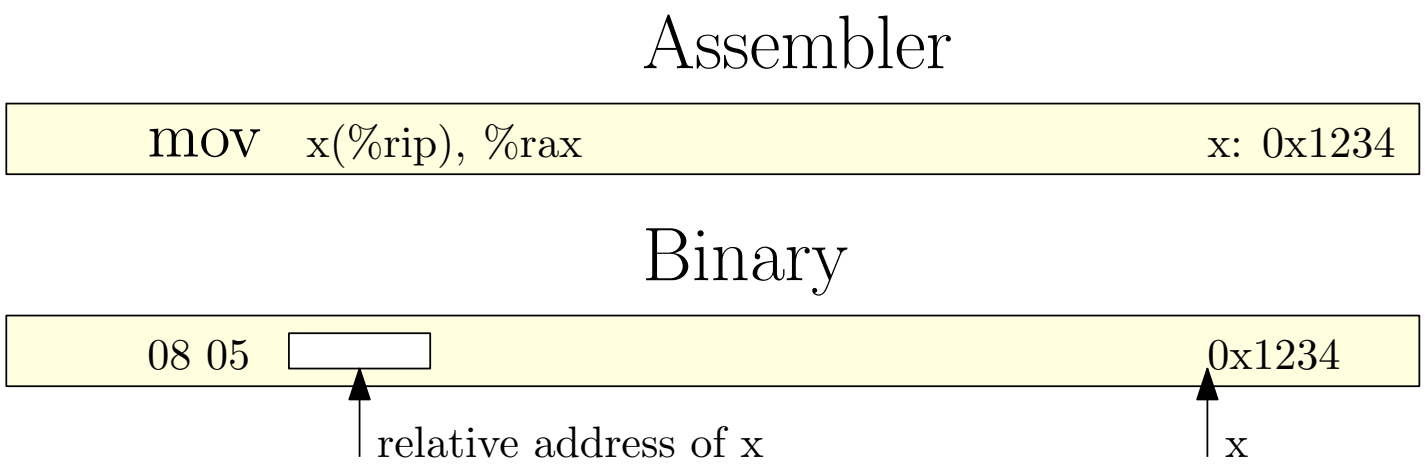
```
Disassembly of section .text:

0000000000000000 <main>:
  0:  8b 05 00 00 00 00     mov   0x0(%rip),%eax      # 6 <main+0x6>
   6:   c3                        retq
```

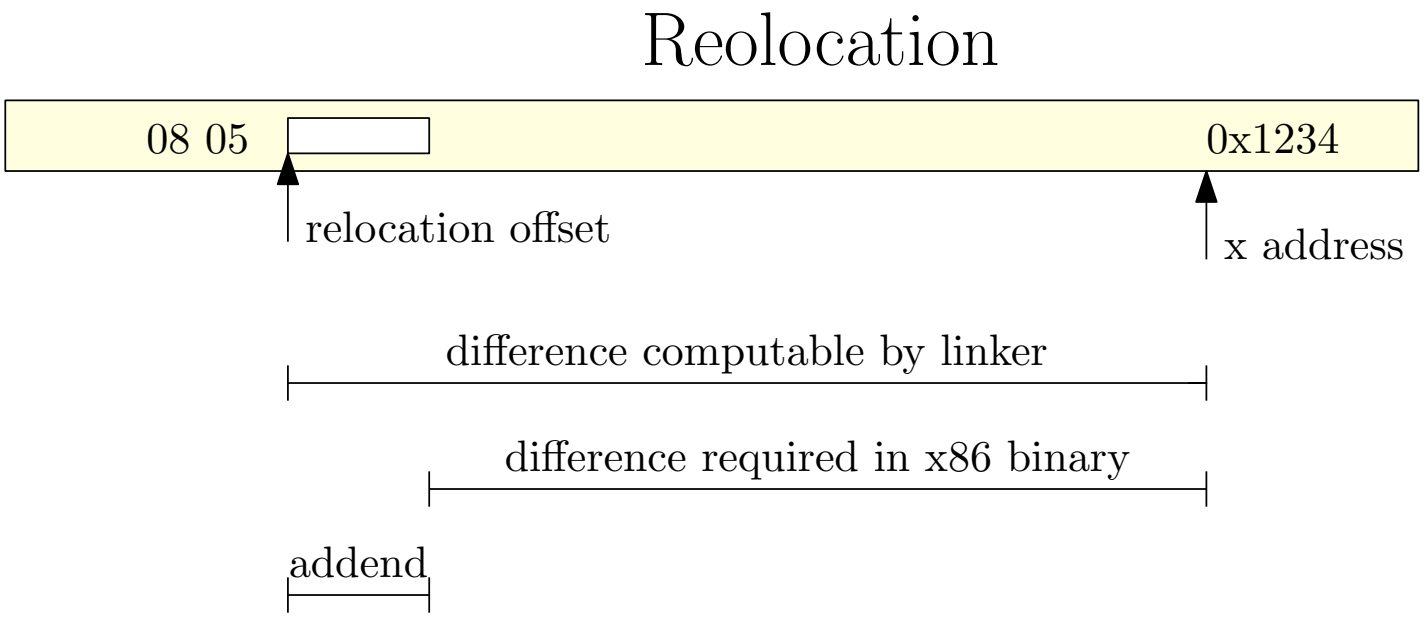
readelf -r main.o

```
Relocation section '.rela.text' at offset 0x1a8 contains 1 entry:
  Offset       Info        Type        Sym. Value   Sym. Name + Addend
000000000002 000900000002 R_X86_64_PC32 0  x - 4
```

Релокация


Assembler
mov x(%rip), %rax
x: 0x1234
Binary
08 05
0x1234
relative address of x
x

Релокация


Reolocation
08 05
0x1234
relocation offset
x address
difference computable by linker
difference required in x86 binary
addend

Релокация

- Линкер знает адрес релокации и адрес символа
- Нельзя просто подставить одно в другое
- Ассемблерная инструкция может требовать абсолютного адреса или относительного
- Относительный адрес считается не от байта с релокацией, а от границы инструкции.
- Иногда эту прибавку хранят не в таблице релокаций, а в самом коде.

Релокация по абсолютному адресу

```
    .text
    .globl f
f:
    movabsl x, %eax
    ret
```

Релокация по абсолютному адресу

```
0000000000000000 <f>:
 0: a1 00 00 00 00 00 00  movabs 0x0,%eax
 7: 00 00
 9: c3                    retq

 Offset       Info        Type        Sym. Value   Sym. Name + Addend
000000000001 000500000001 R_X86_64_64     0000000000000000 x + 0
```

Релокация по абсолютному адресу

```
000000000000062a <f>:
 62a:  a1 10 10 20 00 00 00   movabs 0x201010,%eax
 631:    00 00
 633:    c3                        retq

0000000000201010 <x>:
  201010:       78 56
  201012:       34 12
```

Вопрос

Результатом будет адрес переменной или ее значение?

```
movabs x, %eax
```

Разница в интерпретации адреса релокации

Вот так будет адрес. Обратите внимание на разницу мнемоники и опкодов.

```
    movabsq $x, %rax

000000000000063d <y>:
 63d:  48 b8 10 10 20 00 00   movabs $0x201010,%rax
 644:   00 00 00
 647:   c3                       retq
```

Динамическая линковка

Динамические библиотеки

- В системе есть множество библиотек, которые используются многими программами (например GNU libc).
- Код библиотеки хочется переиспользовать между разными процессами
- Возникает две проблемы:
  - Размер библиотек заранее неизвестен, как следствие точные адреса по которым лежит код библиотек можно выяснить только при запуске программы
  - Код библиотек должен быть немодифицируемым

Position Independent Code

- Код динамической библиотеки должен быть готов к тому, чтобы работать, находясь по произвольному адресу
- Как следствие - невозможно обращаться к глобальным переменным напрямую (как по абсолютному адресу, так и по относительному от `%rip`)
- gcc генерирует подобный код если указать `-fPIC`

Position Independent Code

```
int a;
int get_a() {return a;}
void set_a(int _a) {a = _a;}
```

```
gcc -fPIC -fno-asynchronous-unwind-tables -
O2 -S a.c
```

Position Independent Code

```
get_a:
        movq    a@GOTPCREL(%rip), %rax
        movl    (%rax), %eax
        ret

set_a:
        movq    a@GOTPCREL(%rip), %rax
        movl    %edi, (%rax)
        ret
```

Global Offset Table

- Адреса глобальных переменных не известны на этапе компиляции
- В модуле компиляции заводится таблица, в которой будут храниться адреса переменных
- Таблица заполняется динамическим загрузчиком

Global Offset Table

gcc -shared -o liba.so a.o
objdump -D liba.so

```
0000000000005a0 <get_a>:
 5a0: 48 8b 05 49 0a 20 00  mov 0x200a49(%rip),%rax  # 200ff0 <a-
0x34>
 5a7: 8b 00                 mov (%rax),%eax
 5a9: c3                    retq

0000000000005b0 <set_a>:
 5b0: 48 8b 05 39 0a 20 00  mov 0x200a39(%rip),%rax  # 200ff0 <a-
0x34>
 5b7: 89 38                 mov %edi,(%rax)
 5b9: c3                    retq
```

Global Offset Table

readelf -r liba.so

```
 Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000200ff0  000b00000006 R_X86_64_GLOB_DAT 0000000000201024 a + 0
```

Global Offset Table

Благодаря GOT:

- Код не модифицируется - потенциально можно использовать в разных процессах
- Нужно только один раз записать адрес переменной в GOT вместо того чтобы патчить во всех местах

Глобальные переменные

```
int a;
int get_a() {return a;}
```

```
get_a:
        movq    a@GOTPCREL(%rip), %rax
        movl    (%rax), %eax
        ret
```

Два обращения к памяти даже с `-O2`!

Глобальные неэкспортируемые переменные

- По умолчанию глобальные переменные в C видны из других файлов
- Можно ограничить область видимости одним файлом
- Для этого перед переменной нужно написать `static`

Глобальные переменные

```
static int a;
int get_a() {return a;}
```

```
get_a:
        movl    a(%rip), %eax
        ret

        .local a
```

Вопросы на понимание

- Какие релокации будут в `a.o`?
- Какие релокации будут в `liba.so`?

Ответ для a.o

objdump -D a.o

```
0000000000000000 <get_a>:
  0:  8b 05 00 00 00 00     mov   0x0(%rip),%eax      # 6 <get_a+0x6>
   6:   c3                          retq

0000000000000000 <a>:
   0:   00 00
```

readelf -r a.o

```
 Offset       Info        Type        Sym. Value   Sym. Name + Addend
000000000002 000400000002 R_X86_64_PC32 0  .bss - 4
```

Ответ для liba.so

objdump -D liba.so

```
0000000000000560 <get_a>:
 560: 8b 05 be 0a 20 00  mov   0x200abe(%rip),%eax  # 201024 <a>
 566: c3                 retq
```

readelf -r liba.so релокаций по a не показывает.

Динамическая линковка функций

Линковка функций

main.c

```
int get_a();
void set_a(int);

int main() {
    set_a(12);
    return get_a();
}
```

Линковка функций

gcc -fno-asynchronous-unwind-tables -O2 -S main.c

```
main:
        subq    $8, %rsp
        movl    $12, %edi
        call    set_a@PLT
        xorl    %eax, %eax
        addq    $8, %rsp
        jmp     get_a@PLT
```

Линковка функций

gcc -fno-asynchronous-unwind-tables -O2 -c main.c objdump -d main.o

```
0000000000000000 <main>:
   0: 48 83 ec 08          sub    $0x8,%rsp
   4: bf 0c 00 00 00       mov    $0xc,%edi
   9: e8 00 00 00 00       callq  e <main+0xe>
   e: 31 c0                xor    %eax,%eax
  10: 48 83 c4 08          add    $0x8,%rsp
  14: e9 00 00 00 00       jmpq   19 <main+0x19>
```

Линковка функций

readelf -r

```
 Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000a  000a00000004 R_X86_64_PLT32 0 set_a-4
000000000015  000b00000004 R_X86_64_PLT32 0 get_a-4
```

Линковка функций

```
gcc -o main main.o  -la -L.
export LD_LIBRARY_PATH=.
```

- `-la` указывает что нужно слинковаться с библиотекой `liba`
- `-L.` добавляет локальную директорию к директориям в которых ищутся библиотеки
- `LD_LIBRARY_PATH` действует аналогично, но уже при загрузке программы

Динамические библиотеки

ldd main

```
 linux-vdso.so.1 (0x00007ffc870bd000)
liba.so => ./liba.so (0x00007f0a6f3ee000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00
/lib64/ld-linux-x86-64.so.2 (0x00007f0a6f7f2000)
```

Линковка функций

objdump -d main

```
0000000000000640 <main>:
 640:  48 83 ec 08          sub   $0x8,%rsp
 644:  bf 0c 00 00 00        mov   $0xc,%edi
 649:  e8 c2 ff ff ff        callq  610 <set_a@plt>
 64e:  31 c0                xor   %eax,%eax
 650:  48 83 c4 08          add   $0x8,%rsp
 654:  e9 c7 ff ff ff        jmpq  620 <get_a@plt>
 659:  0f 1f 80 00 00 00 00   nopl  0x0(%rax)
```

Линковка функций

objdump -d main

```
0000000000000610 <set_a@plt>:
 610:  ff 25 b2 09 20 00     jmpq  *0x2009b2(%rip)
 616:  68 00 00 00 00        pushq  $0x0
 61b:  e9 e0 ff ff ff       jmpq  600 <.plt>

0000000000000620 <get_a@plt>:
 620:  ff 25 aa 09 20 00     jmpq  *0x2009aa(%rip)
 626:  68 01 00 00 00        pushq  $0x1
 62b:  e9 d0 ff ff ff       jmpq  600 <.plt>
```

Линковка функций

readelf -r main

```
 Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000200fc8  000200000007 R_X86_64_JUMP_SLO 00000000 set_a + 0
000000200fd0  000500000007 R_X86_64_JUMP_SLO 00000000 get_a + 0
```

Procedure Linkage Table

- Динамическое связывание функций происходит не напрямую, а через специальную таблицу, PLT
- Адреса функций хранятся в GOT
- Трамплины в PLT делают переход через адрес в GOT
- Изначально адрес указывает обратно в PLT на следующую инструкцию
- При первом вызове будет снова вызван динамический линковщик, который найдет адрес функции по ее индексу в PLT и пропишет его в GOT

PLT and GOT

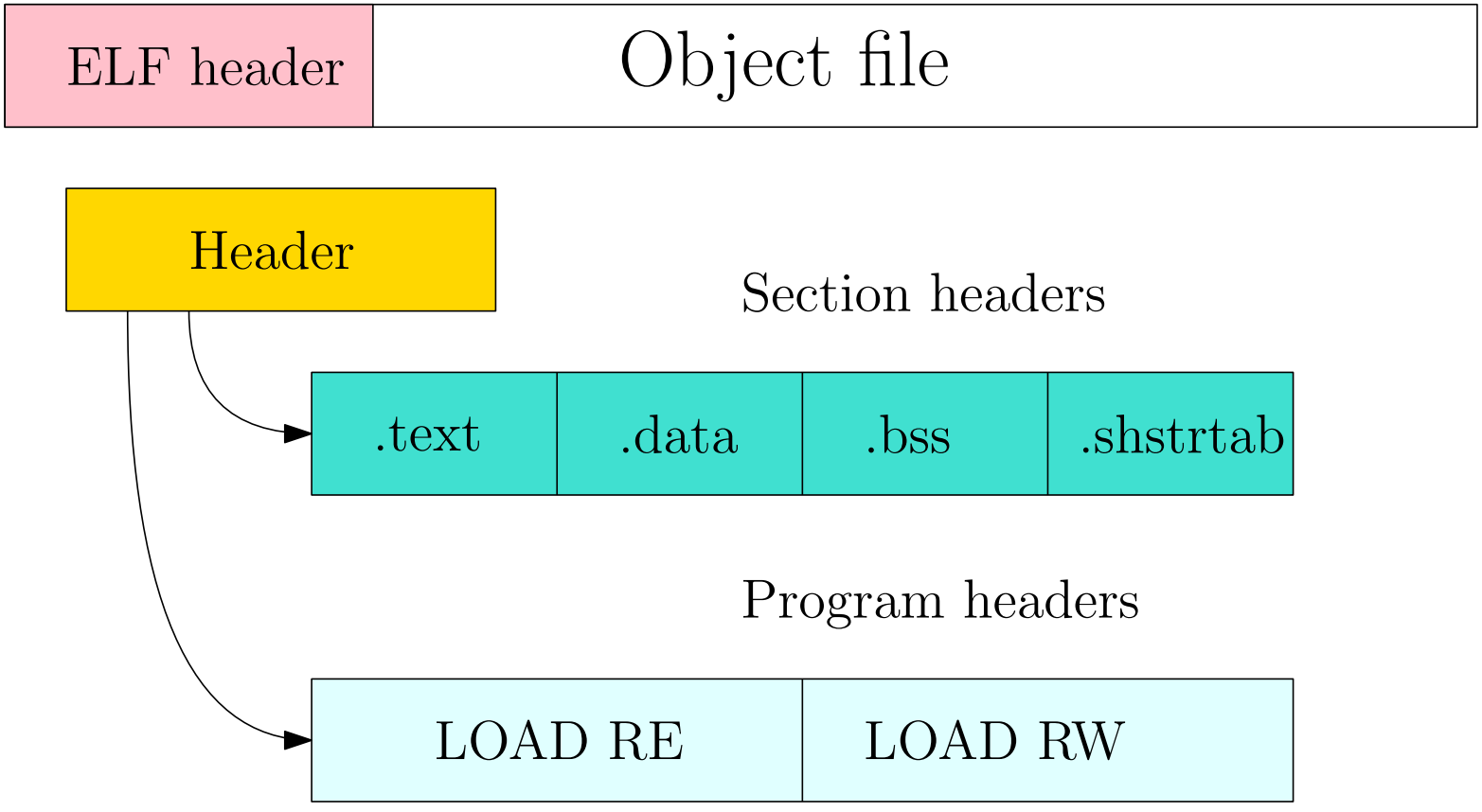
- Первая запись в PLT - трамплин на линковщик
- Дальше в PLT идут трамплины для функций
- Первые 3 записи в GOT указывают на системную информацию для линковщика (в частности 2я запись на сам линковщик)
- Далее идут адреса функций, соответствующие PLT
- Далее идут указатели на глобальные переменные

Executable and Linkable Format

ELF

- Двоичные файлы записываются в некотором формате
- В Linux объектные файлы, исполняемые файлы и файлы динамических библиотек записываются в одном формате ELF
- В ELF описано как загружать файл в память, какие там есть символы и какие нужно применить релокации

Структура ELF



Структура ELF

- Выделяют Header, Section headers и Program headers
- Header содержит общие данные (entry point,
- Section headers описывают секции (`.text`, `.data`)
- Program headers описывают как раскладывать данные в памяти: смещения, режимы доступа (read, write, execute)

Заголовок ELF

Вывод `readelf -h`:

```
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:               0x4000e5
  Start of program headers:          64 (bytes into file)
  Start of section headers:          504 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         2
  Size of section headers:           64 (bytes)
  Number of section headers:         6
  Section header string table index: 5
```

Секции в ELF

- `.text` - собственно исполняемый код
- `.data` - статические переменные
- `.bss` - динамические выделямая память
- `.text.rela` - релокации для кода
- `.symtab` - таблица символов
- `.strtab` - имена символов
- `.shstrtab` - имена секций
- `.dynamic` - таблица для динамической линковки
- `.dynstr` - таблица строк для динамической линковки
- `.dynsym` - таблица символов для динамической линковки

Чтение ELF самостоятельно

Структуры данных ELF определены в заголовке `elf.h`:

```
#include <elf.h>

FILE *f = fopen("/path/to/object", "r");
Elf64_Ehdr header;
fread(&header, sizeof(header), 1, f);
```

Заголовок ELF

```
typedef struct
{
  unsigned char e_ident[16]; /* ELF identification */
   Elf64_Half e_type; /* Object file type */
   Elf64_Half e_machine; /* Machine type */
   Elf64_Word e_version; /* Object file version */
   Elf64_Addr e_entry; /* Entry point address */
   Elf64_Off e_phoff; /* Program header offset */
   Elf64_Off e_shoff; /* Section header offset */
  Elf64_Word e_flags; /* Processor-specific flags */
   Elf64_Half e_ehsize; /* ELF header size */
  Elf64_Half e_phentsize; /* Size of program header entry */
  Elf64_Half e_phnum; /* Number of program header entries */
  Elf64_Half e_shentsize; /* Size of section header entry */
  Elf64_Half e_shnum; /* Number of section header entries */
  Elf64_Half e_shstrndx; /* Section name string table index */
} Elf64_Ehdr;
```

Чтение секций

- Заголовки секций идут последовательно начиная с оффсета `e_shoff`
- Всего заголовков `e_shnum` штук
- В `e_shstrndx` содержится номер секции `.shstrtab`
- В секции `.shstrtab` содержатся названия всех секций в файле
- Секцию `.shstrtab` нужно читать также как и любую другую

Чтение секции

Сперва нужно прочитать заголовок, только потом данные

```
typedef struct
{
  Elf64_Word sh_name; /* Section name */
  Elf64_Word sh_type; /* Section type */
  Elf64_Xword sh_flags; /* Section attributes */
 Elf64_Addr sh_addr; /* Virtual address in memory */
  Elf64_Off sh_offset; /* Offset in file */
  Elf64_Xword sh_size; /* Size of section */
  Elf64_Word sh_link; /* Link to other section */
 Elf64_Word sh_info; /* Miscellaneous information */
 Elf64_Xword sh_addralign; /* Address alignment boundary */
 Elf64_Xword sh_entsize; /* Size of entries, if section has table */
} Elf64_Shdr;
```

Чтение секции

- Заголовок секции размера `sizeofElf64_Shdr` записан по смещению `e_shoff + sizeof(Elf64_Shdr) * index` от начала файла.
- Данные секции размера `sh_size` записаны по смещению `sh_offset`.
- Чтобы узнать имя секции, секция `.shstrtab` должна быть подгружена.
- `sh_name` - смещение внутри секции `.shstrtab`.
- Имя - обычная `'\0'`-терминированная строка.
- В `sh_type` указывается тип секции.

Некоторые типы секций

- `PROGBITS` - содержит бинарные данные программы (`.text` и `.data` имеют этот тип).
- `SYMTAB` - содержит таблицу символов для линкера (`.symtab`).
- `STRTAB` - содержит строки (имена символов или секций: `.shstrtab` и `.strtab`).
- `RELA` - содержит релокации в формате RELA (`.rela.text`).
- `NOBITS` - в файле нет данных для этой секции (`.bss`).
- `DYNAMIC` - содержит таблицу для динамической линковки (`.dynamic`)

Релокации

- `r_offset` - оффсет до позиции в которой применяется релокация.
- `r_info` - в младших 32битах содержится тип релокации, в старших 32битах индекс в таблице символов.
- `r_addend` - содержит константу, которую нужно прибавить к получившемуся значению.

```
typedef struct
{
  Elf64_Addr r_offset; /* Address of reference */
 Elf64_Xword r_info; /* Symbol index and type of relocation */
 Elf64_Sxword r_addend; /* Constant part of expression */
} Elf64_Rela;
```

Таблица строк

- Строки находятся в специальных таблицах строк
- `.strtab` - имена символов
- `.shstrtab` - имена секций
- `.dynstr` - имена символов для динамической линковки
- В таблицах строк строки записаны подряд. Каждая строка заканчивается `\0`.
- Первая строка пустая
- Записи ссылаются на таблицу символов просто по позиции первого симовла. Тем самым получается строка в стиле языка C.

Таблица символов

- Таблица символов содержит записи типа `Elf64_Sym`
- Тип может быть `OBJECT`, `FUNC`, `SECTION`, `FILE`
- Биндинг может быть `LOCAL`, `GLOBAL`, `WEAK`.

```
typedef struct
{
  Elf64_Word st_name; /* Symbol name */
 unsigned char st_info; /* Type and Binding attributes */
  unsigned char st_other; /* Reserved */
  Elf64_Half st_shndx; /* Section table index */
  Elf64_Addr st_value; /* Symbol value */
 Elf64_Xword st_size; /* Size of object (e.g., common) */
} Elf64_Sym;
```

Пример описания

f.c:

```
int f( ) {return 0}
```

f.s:

```
        .file   "f.c"
        .text
        .p2align 4,,15
        .globl  f
        .type   f, @function
f:
        xorl    %eax, %eax
        ret
        .size   f, .-f
```

Пример описания

readelf -s

```
Symbol table '.symtab' contains 8 entries:
Num:   Value        Size Type   Bind  Vis    Ndx Name
  0: 0000000000000000    0 NOTYPE  LOCAL  DEFAULT  UND
  1: 0000000000000000    0 FILE   LOCAL  DEFAULT  ABS f.c
  2: 0000000000000000    0 SECTION LOCAL  DEFAULT   1
  3: 0000000000000000    0 SECTION LOCAL  DEFAULT   2
  4: 0000000000000000    0 SECTION LOCAL  DEFAULT   3
  5: 0000000000000000    0 SECTION LOCAL  DEFAULT   5
  6: 0000000000000000    0 SECTION LOCAL  DEFAULT   4
  7: 0000000000000000    3 FUNC   GLOBAL DEFAULT   1 f
```

Поиск функций

- Функции находятся в секции `.symtab`.
- В секции `.symtab` содержится `sh_size / sh_entsize` записей типа `Elf64_Sym`.
- `st_name` - смещение внутри `.strtab`.
- Функции имеют тип (младшие 4 бита `st_info`) равным `STT_FUNC`.
- ”Адрес” функции записан в `st_value`.

Утилиты

- `readelf` - покажет все про ELF
- `strings`, `nm` - позволяют посмотреть какие символы в бинарнике
- `objdump` - дизассемблер тоже в курсе про ELF, в частности может сразу показывать релокации
- `objcopy` - утилита для манипулирования бинарниками. В частности можно всю дебажную информацию и символы вынести в отдельные файлы.
- `ld` - непосредственно линковщик. С помощью скриптов можно описывать желаемый layout в памяти.

Исполнение

Запуск программы

- Для загрузки динамических библиотек вначале вызывается интерпретатор, обычно `/lib64/ld-linux-x86-64.so.2`
- Интерпретатор указан в секции `.interp` в ELF
- После подгрузки и применения релокаций, управление передается на точку входа в программу
- Точка входа указана в заголовке ELF
- В случае обычных программ точка входа находится в `libc`, сперва выполняется инициализацию и запуск кастомных конструкторов
- Только потом управление передается в `main`.

ASLR

- В домашнем задании вы видели, что может быть когда злоумышленнник знает ваш код
- Чтобы ему помешать применяется техинка address space layout randomization
- В Linux ее принято делать через position independent executable
- GCC понимает какой бинарник собирать по ключам `-fpie` или `-fno-pie`
- Получившиеся ELF отличаются типом, указаным в заголовке ELF: у бинарников собранных для запуска по фиксированному адресу тип EXEC, а у бинарников, не зависящих от адреса, тип DYN.

Запуск программы

- Выполнеие программы на C начинаетс с функции `main`:
  ```
  int main(int argc, char *argv[],
  char *env[]);
  ```
- Вопрос: откуда приходят аргументы?

Запуск программы

- Выполнеие программы на C начинаетс с функции `main`:
  ```
  int main(int argc, char *argv[],
  char *env[]);
  ```
- Вопрос: откуда приходят аргументы?
- Ядро кладёт их на стек
- Libc подготавливает окружение и вызывает `main`

Стек в начале: x86_64 ABI

Figure 3.9: Initial Process Stack

Purpose	Start Address	Length
Unspecified	High Addresses	
Information block, including argument strings, environment strings, auxiliary information ...		varies
Unspecified		
Null auxiliary vector entry		1 eightbyte
Auxiliary vector entries ...		2 eightbytes each
0		eightbyte
Environment pointers ...		1 eightbyte each
0	`8+8*argc+%rsp`	eightbyte
Argument pointers	`8+%rsp`	argc eightbytes
Argument count	`%rsp`	eightbyte
Undefined	Low Addresses	

Запуск функции main

- В ELF точкой входа в программу указывается `_start`
- `_start` должен подготовть `argc`, `argv` и `envp` для `main`, а также позаботиться о коде возврата.

```
    .globl _start
_start:
    movq (%rsp), %rdi
    leaq 8(%rsp), %rsi
    leaq 16(%rsp, %rdi, 8), %rdx
    call main
    movq %rax, %rdi
    movl $60, %eax
    syscall
```

Подгрузка библиотек в рантайме

Подгрузка библиотек в рантайме

- Динамические библиотеки можно подгружать в рантайме.
- Функциональность подходит для бинарных плагинов.
- Также можно писать обертки и подменять вызовы.

Подгрузка библиотек в рантайме

- `dlopen` - открывает разделяемую библиотеку
- `dlsym` - ищет символ в указаной библиотеке
- `dladdr` - определяет, находится ли адрес в какой-нибудь из подгруженных динамических библиотек
- `dlinfo` - получить некоторую информацию, вроде таблицы линковки или пути библиотеки
- `dlerror` - получить текстовое представление последней ошибки
- `dlclose` - выгрузить библиотеку

dlopen

```
void *dlopen(const char *filename, int flags);
```

- Возвращает хендлер, который потом можно передавать `dlsym` и `dlclose`
- `filename` - путь к динамической библиотеке. Если `NULL`, то будет хендлер на саму программу, если содержит `"/"`, то интерпретирует как путь, иначе ищет по системным директориям
- Во флагах долежн обязательно быть один из `RTLD_LAZY` или `RTLD_NOW` (определяет когда будут применяться релокации)
- Можно указать `RTLD_GLOBAL` (символы будут видны следующим загруженным), `RTLD_LOCAL` (не будут видны)

Пример использования

dplugin.c

```
extern int x;
int f(int);
int g() {
    return f(x)
}
```

Пример использования

dmain.c

```
#include <dlfcn.h>
#include <stdio.h>
#include <err.h>

int x = 3;
int f(int x) {return x*2;}

int main() {
  void *dh = dlopen("./dplugin.so",
    RTLD_LOCAL | RTLD_LAZY);
  if (!dh) err(1, "%s", dlerror());
 int (*g)() = (int (*)()) dlsym(dh, "g");
  printf("%d\n", g());
}
```

Пример использования

```
$ gcc -fPIC -shared -o dplugin.so dplugin.c
$ gcc -rdynamic -o dmain dmain.c -ldl
$ ./dmain
6
```

`-rdynamic` нужен чтобы все символы из dmain.c были добавлены в таблицу динамических символов

Подмена библиотек

LD_PRELOAD

- С помощью механизма динамической линковки можно подменять символы других библиотек на свои
- Например, так работают кастомные аллокаторы - переопределяя функции `malloc`, `free`.
- Также можно сделать враппер над существующей функцией

Пример враппера

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>

FILE* fopen(
    const char *pathname,
    const char *mode) {

   FILE* (*h)(const char*, const char*) =
     (FILE* (*)(const char*, const char*))
        dlsym(RTLD_NEXT, "fopen");
   printf("Called fopen(%s, %s)\n", pathname, mode);
    return h(pathname, mode);
}
```

Пример враппера

```
#include <stdio.h>
int main() {
    FILE *f = fopen("somefile", "r");
    char buf[512];
    fgets(buf, 512, f);
    printf("read: %s", buf);
    fclose(f);
}
```

Пример враппера

```
$ gcc -o wmain wmain.c
$ ./wmain
read: here am I

$ gcc -fPIC -shared -o wrapper.so wrapper.c -
ldl
$ LD_PRELOAD=./wrapper.so ./wmain
Called fopen(somefile, r)
read: here am I
```

Слабые символы

Слабые символы

- ELF позволяет определить **слабые** символы
- При связывании обычный (сильный) символ переопределяет слабые символы
- Если будут два обычных определения одного и того же символа, то будет конфликт. Со слабыми такого нет
- Слабые символы удобно использовать когда нужно написать дефолтную имплементацию, которая скорее всего будет заменена библиотекой

Пример слабых символов

main.c

```
#include <stdio.h>
int popcnt(int x);
int main() {
    printf("%d\n", 5);
}
```

Пример слабых символов

popcnt_slow.c

```
int __attribute__((weak)) popcnt(int x) {
    int r = 0;
    while (x) {
        ++r;
        x &= x-1;
    }
    return r;
}
```

Пример слабых символов

popcnt_fast.c

```
int popcnt(int x){
    return __builtin_popcount(x);
}
```

Пример слабых символов

```
$ gcc -o pmain pmain.c popcnt_slow.c popcnt_fast.c
$ ./pmain
2
```

Литература

- x86-64 Application Binary Interface.
- Ulrich Drepper. How To Write Shared Libraries.
- ELF64 Object File Format