БИНАРНЫЙ ТРАНСЛЯТОР

Баринов Денис

МФТИ, гр.Б05-931

2020 г.

ЦЕЛЬ

Написать бинарный транслятор в x86-64

Для этого необходимо:

- Изучить структуру ELF файла
- Определить минимальный исполняемый вид и реализовать программу на его основе
- Установить соответствие между командами на своём процессоре и x86-64
- Перевести решение двух задач: вычисление корней квадратного уравнения и нахождение факториала числа

Executable and Linkable Format

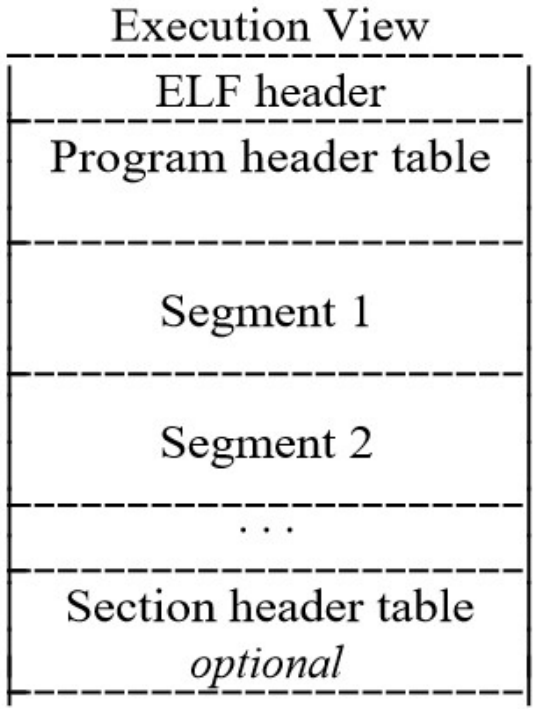
ELF - формат исполняемых (но на самом деле не только) файлов, используемый в Linux (и других, менее популярных системах). Но нас интересует исполняемый.

Структура:

- Elf header (Содержит служебную информацию)
- Section header table (содержит информацию о секциях)
- Sections (.data, .text …)
- Segments (появляются после линковки)

Но для минимального исполняемого файла нужно:

- Elf header
- Program header
-code (сам код программы)



ELF HEADER

1) **0x7f, 0x45, 0x4c, 0x46** (ELF) – первые 4 байта, следующие 5 байт зависят от аппаратуры, а 7 оставшихся резервные (=0)
2) Так как ELF формат подходит для нескольких типов файлов, то надо указать, какой именно. **В нашем случае - 0x02**
3) Зависит от архитектуры аппаратной платформы
4) **0x01** – номер версии формата (единственно корректный)
5) Виртуальный адрес точки входа. (Если такой нет, то 0) Обычно равен 0x400000 + смещение. **В нашем случае 0x400000 + + sizeof (Elf64_Ehdr) + sizeof (Elf64_Phdr)** *(Про Phdr на сл. слайде)*
6) Смещение таблицы заголовков программы от начала файла в байтах. (Если такой нет, то 0). sizeof (Elf32_Ehdr) или sizeof (Elf64_Ehdr)

```
# define EI_NIDENT 16

typedef struct {
  1  unsigned char   e_ident[EI_NIDENT];
  2  Elf64_Half      e_type;
  3  Elf64_Half      e_machine;
  4  Elf64_Word      e_version;
  5  Elf64_Addr      e_entry;
  6  Elf64_Off       e_phoff;
  7  Elf64_Off       e_shoff;
  8  Elf64_Word      e_flags;
  9  Elf64_Half      e_ehsize;
 10  Elf64_Half      e_phentsize;
 11  Elf64_Half      e_phnum;
 12  Elf64_Half      e_shentsize;
 13  Elf64_Half      e_shnum;
 14  Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

ELF HEADER

7) Смещение таблицы заголовков секций от начала файла

в байтах. (Если такой нет, то 0) В нашем случае 0, так как нет.

8) Связанные с файлом флаги, зависящие от процессора.

(Если нет, то 0)

9) Размер заголовка файла в байтах (0x34/0x40 для 32/64 bit соотв.)

10) Размер одного заголовка программы (0x20/0x38 для 32/64 bit соотв.)

11) Число заголовков программы. (Если таких нет, то 0)

В нашем случае будет только 1 program header, так что ставим **0x01**

12) Размер одного заголовка секции. (0x28/0x40 для 32/64 bit соотв.)

13) Число заголовков секций **0** , так как у нас нет

14) Индекс записи в таблице заголовков секций **0**, =//=

```
# define EI_NIDENT 16

typedef struct {
  1 unsigned char   e_ident[EI_NIDENT];
  2 Elf64_Half      e_type;
  3 Elf64_Half      e_machine;
  4 Elf64_Word      e_version;
  5 Elf64_Addr      e_entry;
  6 Elf64_Off       e_phoff;
  7 Elf64_Off       e_shoff;
  8 Elf64_Word      e_flags;
  9 Elf64_Half      e_ehsize;
 10 Elf64_Half      e_phentsize;
 11 Elf64_Half      e_phnum;
 12 Elf64_Half      e_shentsize;
 13 Elf64_Half      e_shnum;
 14 Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

PROGRAM HEADER

1) Определяет тип сегмента, на который указывает заголовок.

Их довольно много, поэтому не буду все перечислять, но нас

интересует **PT_LOAD (0x01)** – загружаемый в память сегмент.

2) Флаги для сегмента (заметим, что *для 32bit это поле находится в другом месте*)

0x01 – разрешение на исполнение,

0x02 – разрешение на запись,

0x04 – разрешение на чтение.

я дал все разрешения, то есть поставил в этом поле **0x07**.

```
typedef struct {
    1  Elf64_Word      p_type;
    2  Elf64_Word      p_flags;
    3  Elf64_Off       p_offset;
    4  Elf64_Addr      p_vaddr;
    5  Elf64_Addr      p_paddr;
    6  Elf64_Xword     p_filesz;
    7  Elf64_Xword     p_memsz;
    8  Elf64_Xword     p_align;
} Elf64_Phdr;
```

3) Смещение сегмента от начала файла.

4) Виртуальный адрес сегмента в памяти (куда будет загружен)

5) Физический сегмента в памяти (Обычно – **0x400000** (для 4 тоже))

6) Размер сегмента в файле (В нашем случае определяется в самом конце)

7) Размер сегмента в памяти (=//=)

8) Выравнивание сегмента. (p_vaddr % p_align = p_offset)

Минимальный исполняемый файл

Заполнив две структуры – ELF header и program header, для нашего минимального вида, осталось только написать сам код программы. Для этого положим в начало (сразу после заголовков) Hello world! (сразу отметим, что длина = 12), после чего уже вызовем syscall, предварительно правильно изменив регистры. То есть всё что нам надо сделать – положить в наш выходной файл сразу же после структур вот этот код:

```
'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!',     //  text
                                                                //
0xb8, 0x01, 0, 0, 0,                                            //  mov eax, 1
0xbf, 0x01, 0, 0, 0,                                            //  mov edi, 1
0x48, 0xbe, 0x78, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00,     //  mov rsi, text
0xba, 0x0c, 0x00, 0x00, 0x00,                                   //  mov edx, 0x0d
0x0f, 0x05,                                                     //  syscall
                                                                //
0xb8, 0x3c, 0x00, 0x00, 0x00,                                   //  mov eax, 0x3c
0xbf, 0x00, 0x00, 0x00, 0x00,                                   //  mov edi, 0
0x0f, 0x05                                                      //  syscall
```

Здесь всё должно быть понятно, кроме может быть того, какое смещение положить в rsi. Но, в нашем случае, и оно вычисляется очень просто, так как мы кладём сразу же после 2 структур =>
=> смещение равно 0x400000 + sizeof (Elf_header) + sizeof (Program_header) (в зависимости от того, 32 или 64 bit, будет разное смещение. В примере показан случай 64bit)

Соответствие команд

Разберём, как транслируются команды, на примере push.

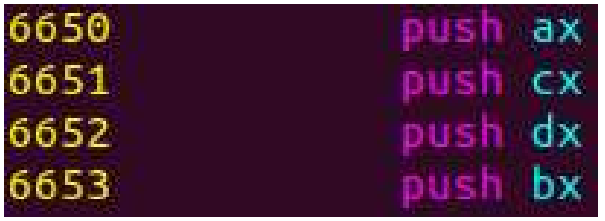
В нашем ассемблере было два варианта аргумента push:
PUSH CONST (push 5)

PUSH REGISTER (push ax)

Посмотрим, как устроены эти команды в x86-64:

Здесь мы видим, что номер команды меняется в зависимости от аргумента, чего не было в нашем процессоре. Поэтому мы отдельно разбираем случай с числами в качестве аргумента, а для регистров замечаем, что opcode меняется следующим образом:

(Точно так же работает и с другими командами)



Соответствие команд

Но есть и более сложные команды. Например, команда sqrt.
Мы понимаем, что вместо одной команды без аргумента нам нужен целый цикл. И, для того чтобы избежать неоправданного роста выходного файла (так как каждый раз будет создаваться цикл), мы отведём в нашем файле место для функций и запишем туда её (также поступим с другими командами, например, OUT, которая печатает число). Функции можно оформить в виде отдельных файлов, и вставлять их с помощью #include в нужное место массива.

Соответствие команд

Также особое внимание стоит отвести на константные строки. Для них нам опять же нужно отвести отдельное место в памяти, куда потом будем ссылаться для вывода строк. Например, для функции PRT (которая так раз и выводит константную строку). Так что важно изначально оценить, сколько памяти на что отвести, и сколько будет таких блоков

Трансляция задач
квадратное уравнение

При трансляции решения квадратного уравнение возникают небольшие проблемы, такие как:

- Команда SQRT
- Вывод числа (в том числе отрицательного)

Но обе эти проблемы решаются уже описанным образом – создаются функции, которые при появлении этих команд вызываются. А в выводе числа надо просто проверить его знак, и если оно оказалось отрицательным, то взять такое же положительное, а потом в самом начале выводимого числа добавить ‘-’. (И не забыть увеличить регистр rdx, отвечающий за длину выводимого буфера)

Трансляция задач
нахождения факториала числа

Для трансляции этой задачи единственное затруднение может вызвать рекурсия. Так как в нашем процессоре был отдельный стек для адресов возврата (для команды ret), а в x86-64 есть только один стек. Но это, опять же, решается выделением памяти. Вместо стека мы просто будем использовать блок памяти и указатель на его конец (эквивалент верхушки стека). Тем самым, всё что нам осталось – реализовать команды call и ret, но первая из них просто кладёт в “стек” адрес следующей команды и делает jmp на нужное место, а команда ret делает обратное действие: jmp на верхний элемент “стека” и сдвигает указатель на предыдущий.

Тесты

Квадратное уравнение

(ax^2 + bx + c = 0)

a b c	
1 1 -6	-3 2 _2_root
0 2 4	-2 _1_root
0 2 0	0 _1_root
0 0 0	_inf_cnt_of_root
0 0 5	_no_root

Факториал числа

n!

n	
4	24
5	120
6	720
7	5040

ВЫВОДЫ

Мы изучили устройство ELF файла и определили его минимальный формат.

Написали перевод команд в x86-64

Перевели решение квадратного уравнения и нахождения факториала числа (с тестами)