

Семинар 13: немного о линкерах

10 марта, 2020

Основные понятия

- ▶ **Symbol** — именованная сущность (обычно функция или глобальная переменная), за которой должен быть закреплён адрес
- ▶ **Content** — описание того, как ОС должна загрузить символ в память
- ▶ **Relocation** — указание вычисления адреса по определённым правилам
- ▶ **Object file** — файл-контейнер для символов и их содержимого
- ▶ Релокации обычно состоят из **offset** и **addend**

Что происходит во время статической линковки?

1. Чтение и парсинг объектных файлов
2. Построение таблицы символов
3. Размещение всех известных секций в памяти
4. Применение релокаций

Спецификация ELF



Спецификация ELF для x86-64



Описание релокации в ELF

```
typedef struct {  
    Elf32_Addr r_offset;  
    Elf32_Addr r_info;  
    Elf32_Addr r_addend;  
} Elf32_Rela;
```

Релокации в x86-64

- ▶ На данный момент для x86-64 существует **38** возможных релокаций
- ▶ Самые распространённые: **R_X86_64_PC32**, **R_X86_64_GOT32**, **R_X86_64_PLT32**
- ▶ Полный список можно найти в ABI для x86-64

Динамическая линковка

- ▶ Исторически идея была придумана для экономии памяти
- ▶ Давайте вынесем часто используемые функции в разделяемые библиотеки или **shared objects**
- ▶ Эти объекты будем подклеивать в различные виртуальные адресные пространства, но физически их будем хранить в единственном экземпляре
- ▶ Как устроить процесс линковки для таких библиотек?

Динамическая линковка

- ▶ Position independent code
- ▶ Global offset table
- ▶ Procedure linkage table

Position independent code

- ▶ Код, начало которого может располагаться как угодно в памяти
- ▶ Значит, что нельзя использовать абсолютные адреса, только относительные
- ▶ RIP-relative addressing
- ▶ Кроме разделяемых объектов используется для *address space layout randomization* (ASLR)

Global offset table

- ▶ Специальная секция (`.got` и `.got.plt`) в ELF
- ▶ Хранит в себе отображение из названий символов в адреса
- ▶ При динамической линковке эти адреса заполняются специальным *интерпретатором* (`ld-linux.so`)
- ▶ Релокации применяются на стадии линковки, но ссылаются на секцию, которая будет заполнена во время исполнения
- ▶ Однако, это всё равно называются релокациями, но они уже находятся в других секциях (`.rel.dyn`, `.rel.plt`)

Procedure linkage table

- ▶ Секция, которая содержит специальным образом сформированные заглушки для вызова процедур
- ▶ Все вызовы сторонних функций в реальности делают `call` на символы, объявленные в этой секции
- ▶ Первый вызов будет триггерить динамический линковщик, остальные вызовы будут напрямую вызывать функцию (*lazy binding*)
- ▶ Можно отключить с помощью опции линкера: `-z now`

Устройство заглушки в PLT-секции: до резолва

```
FUNC@plt:
    jmpq    *FUNC_GOT_OFFSET(%rip)  # Прыжок по адресу в GOT
    pushq   index                   # Индекс символа в .rel.dyn
    jmpq    FIRST_PLT_ENTRY         # relative jump

FUNC@got.plt: # адрес pushq
```

Процесс резолва

1. Вызов `FUNC@plt`
2. Прыжок по адресу в GOT
3. Возврат обратно в PLT
4. Подготовка аргументов для линкера
5. Переход в `FIRST_PLT_ENTRY`
6. Вызов линкера
7. Поиск `.so`-файла и его подгрузка через `mmap` + `mprotect`
8. Запись адреса `FUNC` в `FUNC@got.plt`
9. Вызов `FUNC`

Устройство заглушки в PLT-секции: после резолва

```
FUNC@plt:
    jmpq    *FUNC_GOT_OFFSET(%rip)  # Прыжок по адресу в GOT
    # инструкции ниже никогда больше не выполнятся
    pushq   index
    jmpq     FIRST_PLT_ENTRY

FUNC@got.plt: # абсолютный адрес FUNC
```

Оставшиеся вопросы

- ▶ Как линкер узнает, какие shared objects нужно подгрузить?
- ▶ Как получить текущий EIP под x86?
- ▶ Зачем разделять GOT и PLT?

Оставшиеся вопросы

- ▶ Как линкер узнает, какие shared objects нужно подгрузить? Из секции `dynamic`, которую он парсит на старте
- ▶ Как получить текущий EIP под x86?
- ▶ Зачем разделять GOT и PLT?

Как получить текущий EIP
под x86?

__x86.get_pc_thunk.bx

__x86.get_pc_thunk.bx:

mov (%esp), %ebx

ret

...

call __x86.get_pc_thunk.bx

теперь ebx содержит текущий EIP

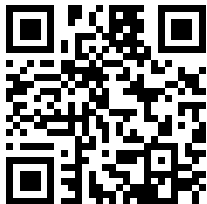
Зачем разделять GOT и PLT?

- ▶ Indirect call — 6 байт, direct call — 5 байт
- ▶ Поэтому компилятор должен был бы различать для чего он компилирует код: для динамической линковки или статической
- ▶ На x86-64 Можно использовать RIP-relative addressing и `-fno-plt`

Статическая vs динамическая линковка

- ▶ Динамическая линковка вызывает *dependency hell*
- ▶ Простота эксплуатации > экономия памяти
- ▶ Молодые языки (например, Rust и Go), поэтому используют статическую линковку
- ▶ Но из-за этого растёт размер исполняемых файлов и время компиляции

Ian Lance Taylor about linkers



vDSO

- ▶ Virtual dynamic shared object
- ▶ Полноценный ELF-объект внутри каждого процесса
- ▶ Ускоряет вызов сисколлов, которые не требуют доступа к «тяжёлым» данным ядра
- ▶ На данный момент содержит в себе: `gettimeofday`, `clock_gettime`, `getcpu`.
- ▶ Идея состоит в том, чтобы реализацию сисколла унести в `userspace`, а необходимые данные периодически подкладывать в адресное пространство процесса

Gratias ago!