Семинар 8: Intel x86 assembly. Часть 2

14 января, 2020

Работа с памятью

AT&T:

$$\mathrm{disp}(\mathrm{base},\ \mathrm{index},\ \mathrm{scale})$$

Intel:

$$[\mathrm{base} + \mathrm{index}*\mathrm{scale} + \mathrm{disp}]$$

- base — базовый регистр, index — «индексирующий»
- scale $\in \{1, 2, 4, 8\}$
- disp (displacement) — signed 32 bit
- Результирующий адрес: base + index*scale + disp

Работа с памятью: примеры

Intel:

```
mov rax, [rax]
mov rax, [rax + rcx]
mov rax, [rax + 8]
mov rax, [rax + 4*rcx]
mov rax, [rax + 2*rcx + 0x10]
```

AT&T:

```
mov (%rax), %rax
mov (%rax, %rcx), %rax
mov 8(%rax), %rax
mov (%rax, %rcx, 4), %rax
mov 0x10(%rax, %rcx, 2), %rax
```

Подпрограммы

- Уже видели `call` — вызов «подпрограммы»
- Программа с точки зрения ассемблера — последовательность инструкций
- Подпрограмма — это последовательность инструкций с прикреплённым к нему контекстом или *stack frame*
- `call arg` запоминает текущий адрес — *адрес возврата* и совершает безусловный прыжок по адресу в аргументе
- `ret` забирает запомненный адрес возврата и передаёт управление обратно

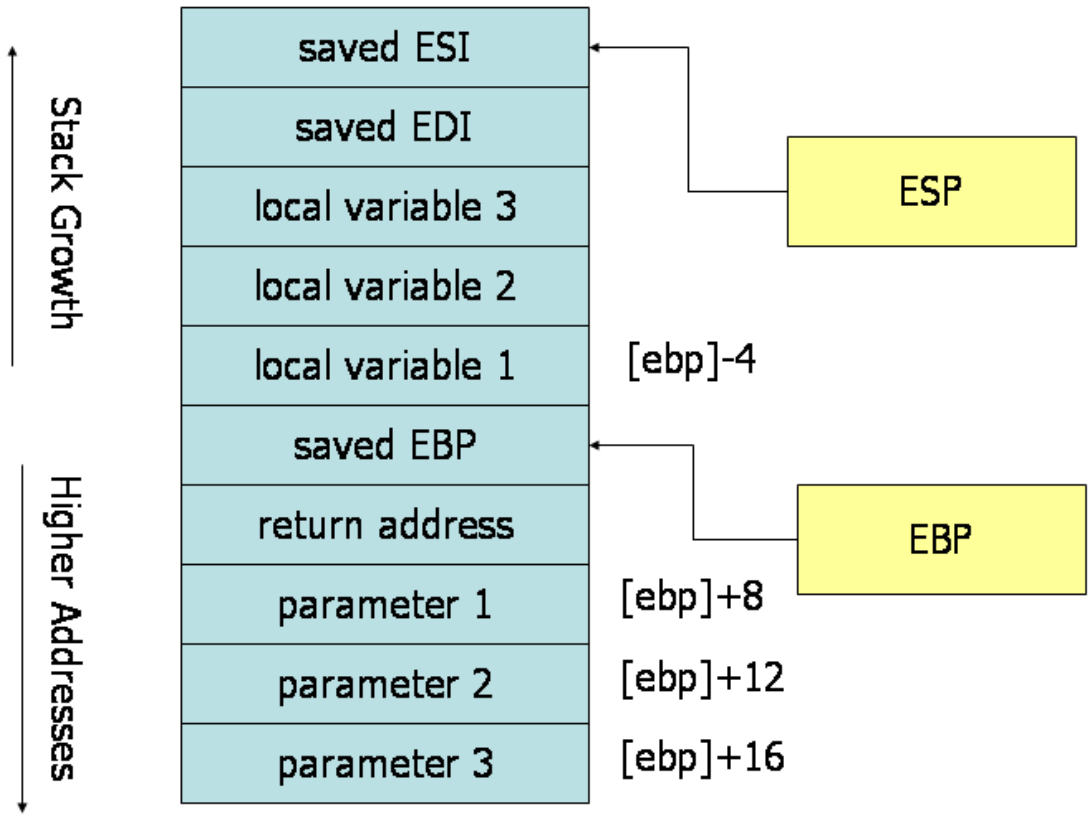
Подпрограммы: аргументы

- Как передавать аргументы?
- Calling conventions или ABI — *application binary interface*
- Будем изучать System V AMD64 ABI

x86 stack and calling conventions

- Состоит из последовательных фреймов, образующих *stack trace*
- Хранит локальные переменные и адрес возврата
- Растёт противоположно росту адресов: больший адрес соответствует «дну» стека
- **esp** указывает на первый свободный байт стека
- **ebp** указывает на начало текущего фрейма
- Всё, что ниже **ebp** — локальные переменные
- Выше расположены: начало предыдущего фрейма (предыдущий **ebp**), адрес возврата и аргументы

Картинка для понимания



Register preserving

- Что делать с регистрами?
- Давайте договоримся какие нужно сохранять, а какие — нет
- Caller-saved registers: **eax**, **ecx**, **edx**
- Callee-saved registers: **ebx**, **esi**, **edi**, **ebp**, **esp**
- **eax** содержит возвращаемое значение (если оно целочисленное)

Передача аргументов под x86

- Кладутся на стек в **обратном** порядке — последний пушится первым
- Первый аргумент лежит в **ebp + 8** (пропускается старый ebp и адрес возврата)
- Заполняются и очищаются вызывающей стороной, callee не должен их изменять

x86-64 calling conventions

- Кладутся в регистры **обратном** порядке — последний пушится первым
- Порядок аргументов: **rdi**, **rsi**, **rdx**, **rcx**, **r8**, **r9**
- Caller-saved: **rax**, **rdi**, **rsi**, **rdx**, **rcx**, **r8**, **r9**, **r10**, **r11**
- Callee-saved: **rbx**, **rsp**, **rbp**, **r12**, **r13**, **r14**, **r15**

Пролог и эпилог функции

```
                # Пролог
push %ebp               # Запоминаем предыдущий фрейм
mov %esp, %ebp          # Запоминаем начало текущего фрейма
push %ebx               # Сохраняем callee-saved регистры
push %edi
# ...
sub $16, %esp           # Выделяем место на стеке

          # ... ваша функция ...

                # Эпилог
add $16, %esp           # Освобождаем место на стеке
# ...
pop %edi                # Восстанавливаем регистры
pop %ebx
mov %ebp, %esp          # Восстанавливаем текущий фрейм
pop %ebp                # Восстанавливаем предыдущий фрейм
ret                     # Возвращаемся из функции
```

Выравнивание стека

- Необходимо, чтобы не использовать лишних and'ов
- В современных ABI — 16 байт
- Под x86 гарантируется, что первый аргумент выровнен по 16ти байтам
- Выравнивать нужно *перед* `call` и *перед* заполнением аргументов
- Если не выровнять стек, то, скорее всего, получите segfault!

Про calling conventions

Gracias!