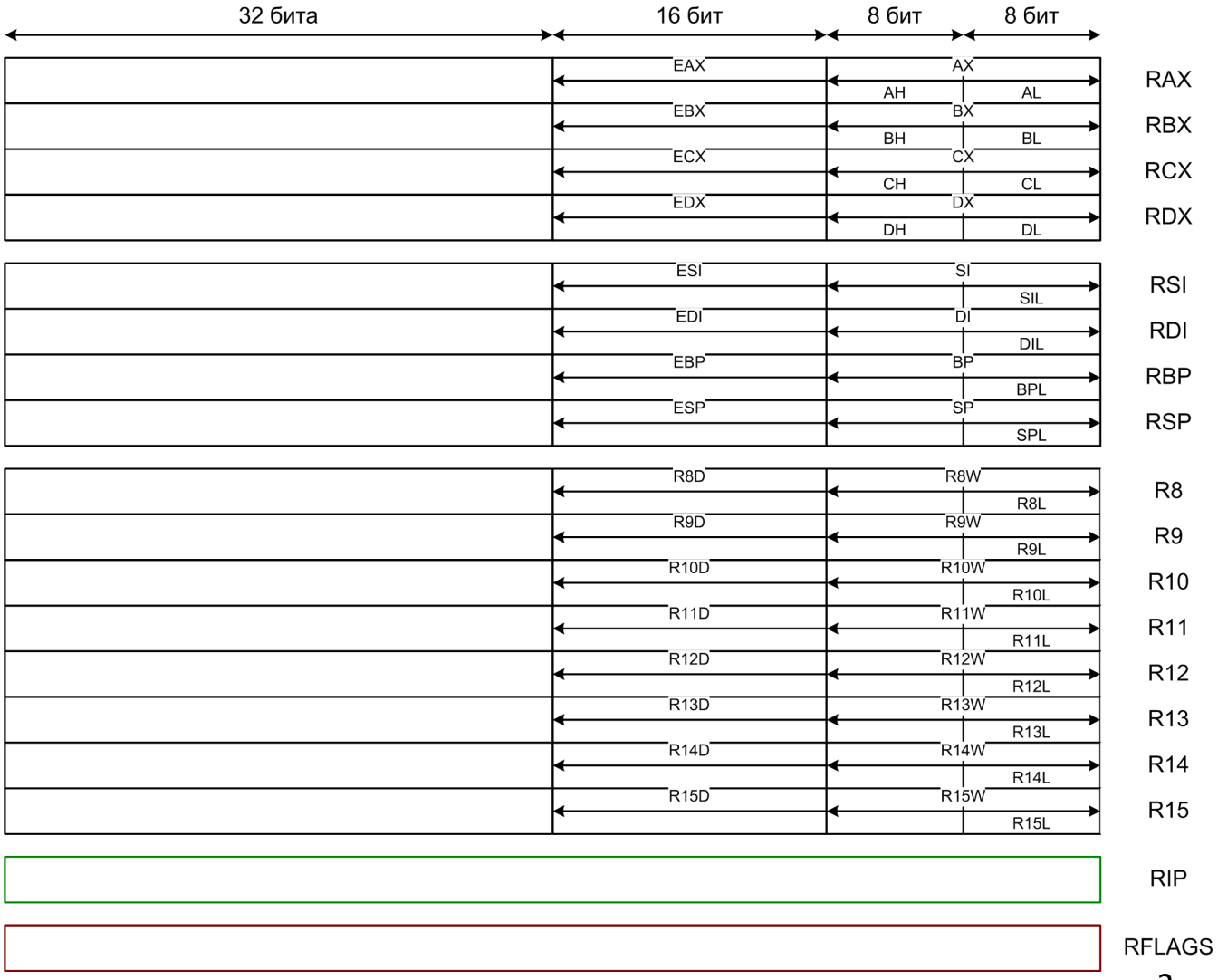
Лекция 0xE

26 марта

x86-64

Регистры x86-64

- Вдвое больше регистров
- Размер регистров удвоился
- Регистры доступны как целиком (64 разряда), так и в виде частей 8, 16, 32 разряда



x86-64

Регистры x86-64: Соглашение по использованию при вызове функций

rax	Возвращаемое значение
rbx	Сохраняется вызванной функцией
rcx	Аргумент #4
rdx	Аргумент #3
rsi	Аргумент #2
rdi	Аргумент #1
rsp	Указатель стека
rbp	Сохраняется вызванной функцией

r8	Аргумент #5
r9	Аргумент #6
r10	Сохраняется вызывающей функцией
r11	Сохраняется вызывающей функцией
r12	Сохраняется вызванной функцией
r13	Сохраняется вызванной функцией
r14	Сохраняется вызванной функцией
r15	Сохраняется вызванной функцией

x86-64

Регистры x86-64

- Аргументы передаются в функцию через регистры
  - Если целочисленных параметров более 6, остальные передаются через стек
  - Регистры-аргументы могут рассматриваться как сохраненные на стороне вызывающей функции
- Все обращения к фрейму организованы через указатель стека
  - Отпадает необходимость поддерживать значения EBP/RBP
- Остальные регистры
  - 6 регистров сохраняется вызванной функцией
  - 2 регистра сохраняется вызывающей функцией
  - 1 регистр для возвращаемого значения
    может рассматриваться как регистр, сохраненный на стороне вызывающей функции
  - 1 выделенный регистр – указатель стека

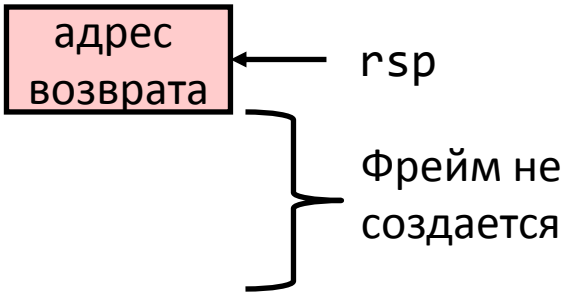
x86-64

Обмен значениями переменных long@x86-64

```
void swap_l(long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   mov  rdx,   qword [rdi]
   mov  rax,   qword [rsi]
   mov  qword [rdi],   rax
   mov  qword [rsi],   rdx
   ret
```

- Параметры передаются через регистры
  - Первый параметр (**xp**) был размещен в **rdi**, второй (**yp**) – в **rsi**
  - 64-разрядные указатели
- Никакие команды не работают со стеком (за исключением **ret**)
- Удалось полностью отказаться от использования стека
  - Все локальные данные размещены на регистрах



x86-64

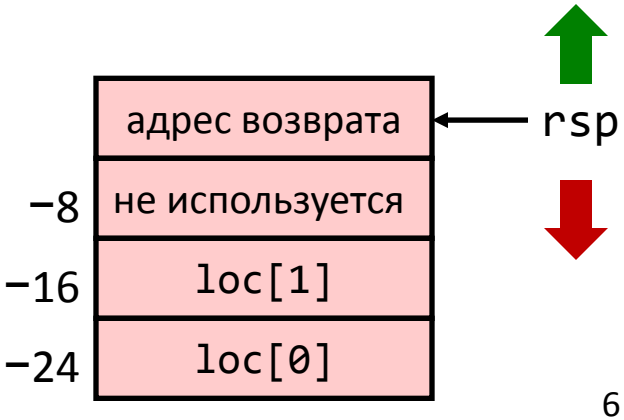
Локальные переменные в «красной зоне»

Листовая функция

```
/*
 * Обмен через локальный массив
 */
void swap_a(long *xp, long *yp) {
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
  mov rax, qword [rdi]
  mov qword [rsp-24], rax
  mov rax, qword [rsi]
  mov qword [rsp-16], rax
  mov rax, qword [rsp-16]
  mov qword [rdi], rax
  mov rax, qword [rsp-24]
  mov qword [rsi], rax
  ret
```

- Обходимся без изменения указателя стека
  - Все данные размещены во «фрейме», неявно организованным под текущим указателем стека



x86-64

Нелистовая функция без организации фрейма

```
/* Обмен a[i] и a[i+1] */
void swap_ele(long a[], int i) {
    swap(&a[i], &a[i+1]);
}
```

- На период работы swap уже никаких значений сохранять на регистрах не требуется
- Не требуется сохранять регистры в качестве вызванной функции
- Команда (префикс) rep используется вместо команды NOP
  - Рекомендации компании AMD для K8

```
swap_ele:
   movsx rsi, esi                  ; знаковое расширение i
   lea   rax, [rdi + 8*rsi + 8]    ; &a[i+1]
   lea   rdi, [rdi + 8*rsi]        ; &a[i] первый аргумент
   mov   rsi, rax                  ;       второй аргумент
   call  swap
   rep                             ; пустая команда / НОП
   ret
```

x86-64

Пример организации фрейма

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i) {
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

```
swap_ele_su:
   mov   [rsp-16], rbx
   mov   [rsp-8],  rbp
   sub   rsp, 16
   movsx rax, esi
   lea   rbx, [rdi + 8*rax + 8]
   lea   rbp, [rdi + 8*rax]
   mov   rsi, rbx
   mov   rdi, rbp
   call  swap
   mov   rax, [rbx]
   imul  rax, [rbp]
   add   [rip + sum], rax
   mov   rbx, [rsp]
   mov   rbp, [rsp+8]
   add   rsp, 16
   ret
```

- Размещаем значения выражений `&a[i]` и `&a[i+1]` в регистрах, сохраняемых на стороне вызванной функции
- Необходимо сформировать фрейм для сохранения этих регистров

Для x86-64 может использоваться одна из четырех моделей построения кода
`-mcmodel=[small | medium | large | kernel]`

x86-64

Как происходит работа с фреймом

```
swap_ele_su:
   mov   [rsp-16], rbx          ; сохраняем rbx
   mov   [rsp-8],  rbp          ; сохраняем rbp
   sub   rsp, 16                ; выделяем на стеке место для фрейма
   movsx rax, esi               ; знаковое расширение i
   lea   rbx, [rdi + 8*rax + 8] ; &a[i+1]
   lea   rbp, [rdi + 8*rax]     ; &a[i]
   mov   rsi, rbx               ; второй аргумент вызова
   mov   rdi, rbp               ; первый аргумент вызова
   call  swap
   mov   rax, [rbx]             ; помещаем в rax a[i+1]
   imul  rax, [rbp]             ; умножаем на a[i]
   add   [rip + sum], rax       ; прибавляем к sum
   mov   rbx, [rsp]             ; восстанавливаем значение rbx
   mov   rbp, [rsp+8]           ; восстанавливаем значение rbp
   add   rsp, 16                ; освобождаем место занятое фреймом
   ret
```

x86-64

Особенности работы с фреймом

- Выделение всего фрейма одной командой
  - Обращения к содержимому фрейма используют адресацию относительно `rsp`
  - Уменьшаем значение в указателе стека
  - Выделение памяти может выполняться не сразу, поскольку в определенных временных пределах хранить данные в «красной зоне» безопасно
- Простое освобождение фрейма
  - Увеличиваем значение в указателе стека
  - Указатель фрейма не требуется

x86-64

Промежуточные итоги
x86-64 : организация вызова функций

- Активное использование регистров
  - Передача параметров
  - Больше регистров – больше возможностей вычислять временные значения и их повторно использовать
- Минимальное использование стека
  - Иногда удается вообще его не использовать
  - Создание/освобождение всего фрейма
- Доступные оптимизации
  - В каком виде будет создан фрейм?
  - Как именно будет выполняться создание?

Ассемблерные вставки

- Нет единого стандарта
- Пример: gcc
  - Наиболее развитый механизм
  - Естественный синтаксис ассемблера для компилятора gcc - AT&T

```
__asm("mov %1, %%eax\n"
      "mov %%eax, %0\n"
      ...
```

```
f:
    push    ebp
    edx, 10
    ebp, esp
# 3 "asm_inline.c" 1
    .intel_syntax noprefix
mov eax, edx
mov edx, eax
# 0 "" 2
#NO_APP
    pop     ebp
    mov     eax, edx
    ret
```

```
int f() {/* тоже самое для синтаксиса Intel
   int a = 10, b;
   __asm(".intel_syntax noprefix\n"
         "mov eax, %1\n"
         "mov %0, eax\n"
         ".att_syntax\n" /* ассемблерная вставка */
        :"=r"(b)         /* выходные операнды    */
        : "r"(a)         /* входные операнды     */
        :"%eax"          /* разрушаемые регистры */
        );
   return b;
}
```

… либо меняем синтаксис опцией компилятора `-masm=intel` во время сборки программы

Ассемблерные вставки и встроенные функции компилятора

```
__attribute__((fastcall)) unsigned f(unsigned u) {
   return __builtin_bswap32(u);
}



__attribute__((fastcall)) unsigned h(unsigned u) {
   __asm(".intel_syntax noprefix\n"
         "bswap %0\n"
         ".att_syntax\n"
         : "=r"(u)
         : "0" (u)
        );
   return u;
}
```

- Ассемблерная вставка мешает работе компилятора
  - Часть регистров «портится»
  - Области кода, в рамках которых распределяются регистры, разбиваются на части меньшего размера, что ухудшает качество распределения
- Многие полезные низкоуровневые функциональности ЦПУ (команды) доступны в виде встроенных функций

Ассемблерные вставки и встроенные функции компилятора

```
__attribute__((fastcall)) unsigned f(unsigned u) {
   return __builtin_bswap32(u);
}


__attribute__((fastcall)) unsigned h(unsigned u) {
   __asm(".intel_syntax noprefix\n"
         "bswap %0\n"
         ".att_syntax\n"
         : "=r"(u)
         : "0" (u)
        );
   return u;
}
```

```
f:
   mov     eax, ecx
   bswap   eax
   ret
```

```
h:
   mov     eax, ecx
   bswap   eax
   ret
```

gcc предоставляет дополнительные возможности по встраиванию ассемблерного кода (для тех, кто понимает, т.е. системных программистов).
Пример: можно явно указывать регистры, которые будут использоваться для размещения операндов ассемблерных вставок.

```
__asm(".intel_syntax noprefix\n"
        "bswap %0\n"
        ".att_syntax\n"
        : "=r"(u)
        : "0" (u)
       );
```

```
h:
    mov     eax, ecx
    bswap   eax
    ret
```

Код	Регистр
a	EAX
b	EBX
c	ECX
d	EDX
S	ESI
D	EDI

```
__asm(".intel_syntax noprefix\n"
        "bswap %0\n"
        ".att_syntax\n"
        : "=c"(u)
        : "0" (u)
       );
```

```
h:
    bswap   ecx
    mov     eax, ecx
    ret
```

- **В прикладных программах ассемблерные вставки не приняты, поскольку из-за них программа не переносима**
- **В системных программах, если вставки и есть, объем ассемблерного кода стараются свести к минимуму**

Далее…

- ***Функции***
  - ***Рекурсия***
  - ***Выравнивание стека***
  - ***Различные соглашения о вызове функций***
    - ***cdecl/stdcall/fastcall, отказ от указателя фрейма***
    - ***Соглашение вызова для x86-64***
  - ***Переменное число параметров***
  - ***Вызов по указателю***
  - ***Ассемблерные вставки***
  - ***Переполнение буфера, эксплуатация ошибок, механизмы защиты***
- Операции над строками / Строковые команды
- Динамическая память
- Числа с плавающей точкой

«Заглянуть за горизонт»

```
void f(int i) {
   int a[3] = {1, 2, 3};
   printf("%x\n", a[i]);
}
```

i	Вывод на экран	
0	1	массив a
1	2	
2	3	
3	b7ff1030	«мусор»
4	8049ff4	
5	bffff748	сохраненный ebp
6	804844b	адрес возврата
7	7	фактический аргумент
8	b7fccff4	содержимое фрейма вызывающей функции
9	8048470	

```
f:
   push    ebp
   mov     ebp, esp
   sub     esp, 40
   mov     eax, dword [ebp+8]
   mov     dword [ebp-20], 1
   mov     dword [ebp-16], 2
   mov     dword [ebp-12], 3
   mov     eax, dword [ebp-20+eax*4]
   mov     dword [esp], .LC0
   mov     dword [esp+4], eax
   call    printf
   leave
   ret
```

gcc 4.9.2

```c
#include <stdio.h>

void f() {
   char buf[5];
   int *ret;

   ret = (int*)(buf + 1);
   (*ret) += 8;
}

void main() {
  int x = 0;

  f();
  x = 1;
  printf("%d\n", x);
}
```

```
f:
   push    ebp
   mov     ebp, esp
   sub     esp, 16
   lea     eax, [ebp-9]
   add     eax, 1
   mov     dword [ebp-4], eax
   mov     eax, dword [ebp-4]
   mov     eax, dword [eax]
   lea     edx, [eax+8]
   mov     eax, dword [ebp-4]
   mov     dword [eax], edx
   leave
   ret
```

```
$gcc -O0 -o retrape retrape.c
$objdump -d -M intel retrape
$ ./retrape
0
$
```

- Можно ли поменять адрес возврата?
- Какие константы (выделены красным) на самом деле должны использоваться?

```
08048439 <main>:
 ...
 8048451:       e8 c5 ff ff ff          call   804841b <f>
 8048456:       c7 45 f4 01 00 00 00    mov    DWORD PTR [ebp-0xc],0x1
 804845d:       83 ec 08                sub    esp,0x8
 8048460:       ff 75 f4                push   DWORD PTR [ebp-0xc]
 8048463:       68 00 85 04 08          push   0x8048500
 8048468:       e8 83 fe ff ff          call   80482f0 <printf@plt>
 ...
```

Цель: выполнить шелл-код

- Имеется способ передать управление на произвольный адрес
- До каких пределов можно расширить доступный функционал программы?
  - Превращаем ее в командный интерпретатор (шелл)
  - Можем делать все, что система разрешает пользователю (владельцу программы)

```
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

```
#include <unistd.h>
int execve(const char *filename, char *const argv [], char *const envp[]);
```

execve() выполняет программу, заданную параметром *filename*.
argv -- это массив строк, аргументов новой программы.
envp -- это массив строк в формате **key=value**, которые передаются новой программе в качестве окружения (environment).
Как *argv*, так и *envp* завершаются нулевым указателем.

```
...
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
   0x08048bbc <+0>:     lea    ecx,[esp+0x4]
   0x08048bc0 <+4>:     and    esp,0xfffffff0
   0x08048bc3 <+7>:     push   DWORD PTR [ecx-0x4]
   0x08048bc6 <+10>:    push   ebp
   0x08048bc7 <+11>:    mov    ebp,esp
   0x08048bc9 <+13>:    push   ecx
   0x08048bca <+14>:    sub    esp,0x14
=> 0x08048bcd <+17>:    mov    DWORD PTR [ebp-0x10],0x80bcfc8
   0x08048bd4 <+24>:    mov    DWORD PTR [ebp-0xc],0x0
   0x08048bdb <+31>:    mov    eax,DWORD PTR [ebp-0x10]
   0x08048bde <+34>:    sub    esp,0x4
   0x08048be1 <+37>:    push   0x0
   0x08048be3 <+39>:    lea    edx,[ebp-0x10]
   0x08048be6 <+42>:    push   edx
   0x08048be7 <+43>:    push   eax
   0x08048be8 <+44>:    call   0x806c550 <execve>
   0x08048bed <+49>:    add    esp,0x10
   0x08048bf0 <+52>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x08048bf3 <+55>:    leave
   0x08048bf4 <+56>:    lea    esp,[ecx-0x4]
   0x08048bf7 <+59>:    ret
End of assembler dump.
```

```
$gcc -static -ggdb -O0 -o execve execve.c
$gdb execve
```

```
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

- Функция execve – обертка над вызовом функции операционной системы (системный вызов)
- В системных вызовах используется другое соглашение вызова, параметры передаются через регистры

Регистр	Содержимое
EAX	Номер функции
EBX	Адрес строки с именем файла
ECX	Массив строк, аргументы запуска программы
EDX	Массив строк с переменными окружения

```
$gcc -static -ggdb -O0 -o execve execve.c
$gdb execve
```

```
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

```
(gdb) disassemble execve
Dump of assembler code for function execve:
   0x0806c550 <+0>:     push   ebx
   0x0806c551 <+1>:     mov    edx,DWORD PTR [esp+0x10]
   0x0806c555 <+5>:     mov    ecx,DWORD PTR [esp+0xc]
   0x0806c559 <+9>:     mov    ebx,DWORD PTR [esp+0x8]
   0x0806c55d <+13>:    mov    eax,0xb
   0x0806c562 <+18>:    call   DWORD PTR ds:0x80edab0
   0x0806c568 <+24>:    pop    ebx
   0x0806c569 <+25>:    cmp    eax,0xffff001
   0x0806c56e <+30>:    jae    0x80700e0 <__syscall_error>
   0x0806c574 <+36>:    ret
End of assembler dump.
```

Вызов еще одной функции-обертки, отвечающей за выполнение конкретной команды передачи управления на код ОС. В современных версиях Linux это команда sysenter, в «старых» - int 0x80.

Что необходимо для запуска шелла?

- Расположить где-то в памяти
  - Ограниченную нулем строку `"/bin/sh"`
  - Массив из двух указателей: адрес строки `"/bin/sh"`, нулевой адрес
- Поместить `0xb` в EAX
- Поместить адрес `"/bin/sh"` в EBX
- Поместить адрес массива указателей в ECX
- Поместить `0x0` в EDX
- Выполнить команду `int 0x80`
  - «Старый» способ вызова функций ОС все еще работает
- Ограничиваем выполнение шелл-кода бесконечным циклом