Московский физико-технический институт

Ассемблер, лабораторная работа № 2

Стек и локальные переменные.

выполнил студент группы Б03-205

Овсянников Андрей

Долгопрудный, 2023 г.

1. Займемся генерацией листингов с вызовом функции... Для каждого пункта будем приводить несколько листингов. Для двух разных архитектур.

   - без аргументов и возвращаемых значений. Напишем так:

```
#include <stdio.h>
void func() {}

int main() {
    func();
    return 0;
}
```

   Сгенерируем листинги на разных архитектурах.

Рис. 1: x64

```
    .file   "li.c"
    .text
    .globl  func
    .type   func, @function
func:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    nop
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   func, .-func

    .globl  main
    .type   main, @function
main:
    ...
    movl    $0, %eax
    call    func
    movl    $0, %eax
    popq    %rbp
...
```

Рис. 2: x32

```
    .file   "li.c"
    .text
    .globl  func
    .type   func, @function
func:
.LFB0:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    nop
    popl    %ebp
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE0:
    .size   func, .-func

    .globl  main
    .type   main, @function
main:
    ...
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    call    func
    movl    $0, %eax
    popl    %ebp
...
```

   Как мы видим различия между листингами весьма существенные, начнем с того, что в 32-ой системе кроме cfi вызывается еще всякая непонятная гадось как глобальная offset-таблица, поменялись из-за этого же концовки файлов, но не хочу на это много внимания обращать, после этого, ясно, поменялись 64-битные регистры на 32-е, но все таки, осталось и много похожего, как структура кода, объявление функций и их вызовы.

   - без аргументов и с возвращаемым значением (int/char).

```
#include <stdio.h>

int func() {
```

```
    return 1;
}

int main() {
    printf("%d \n", func());
    return 0;
}
```

Сгенерируем листинги на разных архитектурах.

Рис. 3: x64

```
    .file   "li.c"
    .text
    .globl  func
    .type   func, @function
func:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $1, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   func, .-func

    .section        .rodata
.LC0:
    .string "%d \n"

    .text
    .globl  main
    .type   main, @function
main:
    ...
    movl    $0, %eax
    call    func
    movl    %eax, %esi
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    popq    %rbp
...
```

Рис. 4: x32

```
    .file   "li.c"
    .text
    .globl  func
    .type   func, @function
func:
.LFB0:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    movl    $1, %eax
    popl    %ebp
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE0:
    .size   func, .-func

    .section        .rodata
.LC0:
    .string "%d \n"

    .text
    .globl  main
    .type   main, @function
main:
    ...
    call    __x86.get_pc_thunk.bx
    addl    $_GLOBAL_OFFSET_TABLE_, %ebx
    call    func
    subl    $8, %esp
    pushl   %eax
    leal    .LC0@GOTOFF(%ebx), %eax
    pushl   %eax
    call    printf@PLT
    addl    $16, %esp
    movl    $0, %eax
    leal    -8(%ebp), %esp
    popl    %ecx
    .cfi_restore 1
    .cfi_def_cfa 1, 0
    popl    %ebx
    .cfi_restore 3
    popl    %ebp
...
```

Что хочется сказать, во первых, возврат значения идет через регистр ax, так везде. Механика в 32 архитектуре другая, нежели в 64, к примеру там выводится число через esi, в 32ой архитектуре никто не думает об этом регистре. Как бы это странно не было, но printf работает через стек локальных переменных, постоянно регистры туда добавляются и убираются.

- с несколькими аргументами.

```
#include <stdio.h>

int func(int a, int b) {
    return a + b;
}

int main() {
    int a = 2, b = 3;

    printf("%d \n", func(a, b));

    return 0;
}
```

Сгенерируем листинги на разных архитектурах.

Рис. 5: x64

```
    .file   "li.c"
    .text
    .globl  func
    .type   func, @function
func:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   func, .-func
    .section        .rodata
.LC0:
    .string "%d \n"
    .text

    .globl  main
    .type   main, @function
main:
    ...
    movl    $2, -8(%rbp)
    movl    $3, -4(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    movl    %edx, %esi
    movl    %eax, %edi
    call    func
    movl    %eax, %esi
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
...
```

Рис. 6: x32

```
    .file   "li.c"
    .text
    .globl  func
    .type   func, @function
func:
.LFB0:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    popl    %ebp
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE0:
    .size   func, .-func
    .section        .rodata
.LC0:
    .string "%d \n"

    .text
    .globl  main
    .type   main, @function
main:
    ...

...
```

Как мы видим аргументы в функцию так же передаются в 64 через регистры, а именно через edi, esi и дальше можно посмотреть по надобности. В 32 же переменные прямо передаются как-то через просто локальный стек, причем это настолько подогнанно что даже смотреть страшно, то есть мы обращаемся к переменной через новый стекпоинтер ebp, причем прибавляем к нему сдвик, то есть блин лезем в локальный стек main, мне такое не нравится, опасно как-то, хотя работает.

2. • Одна локальная переменная будет выглядеть примерно так:

```
    subq    $16, %rsp
movl        $2, -8(%rbp)
```

То есть мы расширили локальный стек и в какое-то место руками забили 2.

- Две и более локальных переменных будет просто выделятся больше памяти, синтаксис от этого не изменится.
- Реальзация статического массива в стеке локальных переменных ничем не отличается от просто создания стопки локальных переменных, интересная тут деталь - это вот этот участок:

```
subq    $32, %rsp
movq    %fs:40, %rax
movq    %rax, -8(%rbp)
xorl    %eax, %eax
```

вот этот fs:40 это "стековая конарейка на самом деле хранит адрес конца стека локальных переменных, bs аналогично начало стека. Причем в 32 системе используется gs:20. Нужны чтобы отслеживать переполнение в случае множетва операций со стеком и переходами от одного локального стека к другому(функции). Как я понимаю появляется в коде в зависимости от используемой в стеке памяти, у меня появилась при освобождении 32 байт локального стека.

- Переходим к динамическим массивам. Будем анализировать листинг вот такой штуки:

```
#include <stdio.h>
#include <stdlib.h>

int func(int a, int b) {
    return a + b;
}

int main() {
    int size = 10;
    int* p = (int*) malloc(size * sizeof(int));

    for (unsigned int i = 0; i < size; ++i) {
        p[i] = i;
    }

    printf("Conclusion: %d + %d = %d", p[0], p[1], p[0] + p[1]);
        return 0;
}
```

Нам нужны листинги, представим их:

Рис. 7: x64

```
    .file   "li.c"
    .text
    .globl  func
    .type   func, @function
func:
.LFB6:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE6:
    .size   func, .-func
    .section        .rodata
.LC0:
    .string "Conclusion: %d + %d = %d"
    .text
    .globl  main
    .type   main, @function
main:
.LFB7:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $10, -12(%rbp)
    movl    -12(%rbp), %eax
    cltq
    salq    $2, %rax ;#
    movq    %rax, %rdi
    call    malloc@PLT
    movq    %rax, -8(%rbp)
    movl    $0, -16(%rbp) ;# i
    jmp     .L4
```

Рис. 8: x32

```
    .file   "li.c"
    .text
    .globl  func
    .type   func, @function
func:
.LFB6:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %edx, %eax
    popl    %ebp
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE6:
    .size   func, .-func
    .section        .rodata
.LC0:
    .string "Conclusion: %d + %d = %d"
    .text
    .globl  main
    .type   main, @function
main:
.LFB7:
    .cfi_startproc
    leal    4(%esp), %ecx
    .cfi_def_cfa 1, 0
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ebp
    movl    %esp, %ebp
    .cfi_escape 0x10,0x5,0x2,0x75,0
    pushl   %ebx
    pushl   %ecx
    .cfi_escape 0xf,0x3,0x75,0x78,0x6
    .cfi_escape 0x10,0x3,0x2,0x75,0x7c
    subl    $16, %esp
    call    __x86.get_pc_thunk.bx
    addl    $_GLOBAL_OFFSET_TABLE_, %ebx
    movl    $10, -16(%ebp)
    movl    -16(%ebp), %eax
    sall    $2, %eax
    subl    $12, %esp
    pushl   %eax
    call    malloc@PLT
    addl    $16, %esp
    movl    %eax, -12(%ebp)
    movl    $0, -20(%ebp)
    jmp     .L4
```

Рис. 9: x64

```
.L5:
    movl    -16(%rbp), %eax
    leaq    0(,%rax,4), %rdx
    movq    -8(%rbp), %rax
    addq    %rax, %rdx
    movl    -16(%rbp), %eax
    movl    %eax, (%rdx)
    addl    $1, -16(%rbp)
.L4:
    movl    -12(%rbp), %eax
    cmpl    %eax, -16(%rbp)
    jb      .L5
    movq    -8(%rbp), %rax
    movl    (%rax), %edx
    movq    -8(%rbp), %rax
    addq    $4, %rax
    movl    (%rax), %eax
    leal    (%rdx,%rax), %ecx
    movq    -8(%rbp), %rax
    addq    $4, %rax
    movl    (%rax), %edx
    movq    -8(%rbp), %rax
    movl    (%rax), %eax
    movl    %eax, %esi
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    leave
    ...
```

Рис. 10: x32

```
.L5:
    movl    -20(%ebp), %eax
    leal    0(,%eax,4), %edx
    movl    -12(%ebp), %eax
    addl    %eax, %edx
    movl    -20(%ebp), %eax
    movl    %eax, (%edx)
    addl    $1, -20(%ebp)
.L4:
    movl    -16(%ebp), %eax
    cmpl    %eax, -20(%ebp)
    jb      .L5
    movl    -12(%ebp), %eax
    movl    (%eax), %edx
    movl    -12(%ebp), %eax
    addl    $4, %eax
    movl    (%eax), %eax
    leal    (%edx,%eax), %ecx
    movl    -12(%ebp), %eax
    addl    $4, %eax
    movl    (%eax), %edx
    movl    -12(%ebp), %eax
    movl    (%eax), %eax
    pushl   %ecx
    pushl   %edx
    pushl   %eax
    leal    .LC0@GOTOFF(%ebx), %eax
    pushl   %eax
    call    printf@PLT
    addl    $16, %esp
    movl    $0, %eax
    leal    -8(%ebp), %esp
    popl    %ecx
    .cfi_restore 1
    .cfi_def_cfa 1, 0
    popl    %ebx
    .cfi_restore 3
    popl    %ebp
    .cfi_restore 5
    leal    -4(%ecx), %esp
    .cfi_def_cfa 4, 4
    ret
    ...
```

Снова у нас разница в funk-ах происходит в передаче аргументов. В 64 они перпедаются по копии через регистры, а в 32 через локальный стек предыдущего уровна глубины. Теперь надо сказать как реализуется само выделение памяти. Обращение осуществляется так же как и к обычному статическому массиву, то есть у нас маллок вернул в $-8(\%rbp)$ адрес на начало нашего динамического массива, собственно дальше идет:

```
    movl    -16(%rbp), %eax
leaq    0(,%rax,4), %rdx
movq    -8(%rbp), %rax
addq    %rax, %rdx
movl    -16(%rbp), %eax
movl    %eax, (%rdx)
addl    $1, -16(%rbp)
```

В которых мы на i-ое место в массиве кладем i. Дальше кроме логики программы смотреть

нечего.

3. Начинаем работу со структурами.

   - Обычная структура. Короче тема такая, структура представляет из себя ровно как и список прсто последовательно лежащих аргументов.

```
#include <stdio.h>
#include <stdlib.h>

struct Dog {
    int n_paw;
    double weight;
    char sex;
};

int print_s(struct Dog dog) {
    return dog.n_paw;
}

int main() {
    struct Dog dog = {1, 2.3, 'm'};
    printf("%d", print_s(dog));
        return 0;
}
```

Рис. 12: x32

Рис. 11: x64

```
movl    $1, -32(%rbp) ;# n_paws
movsd   .LC0(%rip), %xmm0
movsd   %xmm0, -24(%rbp) ;# double weight
movb    $109, -16(%rbp) ;# char sex
pushq   -16(%rbp)
pushq   -24(%rbp)
pushq   -32(%rbp)
```

```
subl    $20, %esp
movl    $1, -24(%ebp)
fldl    .LC2
fstpl   -20(%ebp)
movb    $109, -12(%ebp)
subl    $16, %esp
movl    %esp, %eax
movdqu  -24(%ebp), %xmm0
movups  %xmm0, (%eax)
call    print_s(Dog)
```

Вот мы задали структуру.

   - Теперь добавим как поле статический массив:

```
#include <stdio.h>
#include <stdlib.h>

struct Dog {
    int n_paw;
    double weight;
    char sex;
    int big_teef[4];
};

int print_s(struct Dog dog) {
    return dog.big_teef[2];
}

int main() {
    struct Dog dog = {1, 2.3, 'm', {7, 7, 6, 6}};
    printf("%d", print_s(dog));
    return 0;
}
```

Рис. 13: x64

```
subq    $48, %rsp
movq    %fs:40, %rax
movq    %rax, -8(%rbp)
xorl    %eax, %eax
movl    $1, -48(%rbp)
movsd   .LC0(%rip), %xmm0
movsd   %xmm0, -40(%rbp)
movb    $109, -32(%rbp)
movl    $7, -28(%rbp)
movl    $7, -24(%rbp)
movl    $6, -20(%rbp)
movl    $6, -16(%rbp)
pushq   -16(%rbp)
pushq   -24(%rbp)
pushq   -32(%rbp)
pushq   -40(%rbp)
pushq   -48(%rbp)
call    print_s
```

Рис. 14: x32

```
movl    $1, -44(%ebp)
fldl    .LC0@GOTOFF(%ebx)
fstpl   -40(%ebp)
movb    $109, -32(%ebp)
movl    $7, -28(%ebp)
movl    $7, -24(%ebp)
movl    $6, -20(%ebp)
movl    $6, -16(%ebp)
pushl   -16(%ebp)
pushl   -20(%ebp)
pushl   -24(%ebp)
pushl   -28(%ebp)
pushl   -32(%ebp)
pushl   -36(%ebp)
pushl   -40(%ebp)
pushl   -44(%ebp)
call    print_s
```

Собственно ничего особенного и удивительного.

- Структура в аргументах функции.

```
print_s:
.LFB6:
    .cfi_startproc
    endbr64
    pushq       %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq        %rsp, %rbp
    .cfi_def_cfa_register 6
    movl        44(%rbp), %eax
    popq        %rbp
```

- Вернем структуру.

```
#include <stdio.h>
#include <stdlib.h>

struct Dog {
    int n_paw;
    double weight;
    char sex;
    int big_teef[4];
};

int print_s(struct Dog dog) {
    return dog.big_teef[2];
}

struct Dog abram_lincoln(struct Dog dog) {
    dog.n_paw += 1;
    return dog;
}

int main() {
    struct Dog dog = {1, 2.3, 'm', {7, 7, 6, 6}};
    printf("%d", print_s(dog));
```

```
    return 0;
}
```

Собственно структура в функцию передается через локальный стек, Структура возвращается

Рис. 15: x64

```
abram_lincoln:
    ...
    movl    $3, -64(%rbp)
    movsd   .LC0(%rip), %xmm0
    movsd   %xmm0, -56(%rbp)
    movb    $102, -48(%rbp)
    movl    $4, -44(%rbp)
    movl    $4, -40(%rbp)
    movl    $1, -36(%rbp)
    movl    $1, -32(%rbp)

    movq    -72(%rbp), %rax ;# rdi -> rax

    movq    -64(%rbp), %rcx
    movq    -56(%rbp), %rbx
    movq    %rcx, (%rax)
    movq    %rbx, 8(%rax)
    movq    -48(%rbp), %rcx
    movq    -40(%rbp), %rbx
    movq    %rcx, 16(%rax)
    movq    %rbx, 24(%rax)
    movq    -32(%rbp), %rdx
    movq    %rdx, 32(%rax)
    ...
```

Рис. 16: x32

```
    movl    $3, -44(%ebp)
    fldl    .LC0@GOTOFF(%eax)
    fstpl   -40(%ebp)
    movb    $102, -32(%ebp)
    movl    $4, -28(%ebp)
    movl    $4, -24(%ebp)
    movl    $1, -20(%ebp)
    movl    $1, -16(%ebp)
    movl    -60(%ebp), %eax
    movl    -44(%ebp), %edx
    movl    %edx, (%eax)
    movl    -40(%ebp), %edx
    movl    %edx, 4(%eax)
    movl    -36(%ebp), %edx
    movl    %edx, 8(%eax)
    movl    -32(%ebp), %edx
    movl    %edx, 12(%eax)
    movl    -28(%ebp), %edx
    movl    %edx, 16(%eax)
    movl    -24(%ebp), %edx
    movl    %edx, 20(%eax)
    movl    -20(%ebp), %edx
    movl    %edx, 24(%eax)
    movl    -16(%ebp), %edx
    movl    %edx, 28(%eax)
    movl    -12(%ebp), %eax
```

достаточно хитрым образом, в функцию, которая должна вернуть структуру передаеся адрес куда вернуть через rdi, дальше структура просто поэлементно вставляется по этому адресу вверх. Это джениусли. Перед линкольном мэйн запихал в конец стека все передаваемое. В 32-ой ничего особо в этом плане не поменялось, суть точно та же.

4. А теперь посложнее. Разберемся с референсами. В этот момент делаем финт ушами и неожиданно переключаемся на листинги из плюсов. Так же теперь я не буду предоставлять полное сравнение двух архитектур, буду писать только если замечу сильные отличия. (мне дико лень вставлять листинги) Итак, пусть теперь функция будет

   - со структурой в аргументе: Ну это уже было, давайте дальше сразу.
   - Теперь с указателем на структуру в аргументе.

```
struct Dog {
    int n_paw;
    double weight;
    char sex;
    int big_teef[4];
};


int petrushka(Dog* dog) {
    return dog->n_paw;
}
```

```
...petrushka:
```

```
...
movq    %rsp, %rbp
.cfi_def_cfa_register 6
movq    %rdi, -8(%rbp)
movq    -8(%rbp), %rax
movl    (%rax), %eax
popq    %rbp
.cfi_def_cfa 7, 8
...
```

Вот по сути все сказано, через регистр, здесь rdi, мы передаем в функцию адрес ака указатель на структуру, а там по порядочку у нас все сложено, ну мы уже обсуждали.

- со ссылкой на структуру в аргументе:

```
int petrushka(Dog &dog) {
    return dog.n_paw;
}
```

Ожидание, что ничего не изменится с предыдущего варианта. Ух ты, ничего себе, действительно ничего не поменялось.

- С r-value ссылками все работает аналогично, только появился вызов некой функции $...remove_reference...$ перед вызовом петрушки. В предыдущих сериях этого не было, предположу что это семантика r-value, они же на то и временные.

5. Начинаем испытания на прочность. *** Если что не дополнено примерами, то оно лежит на гите в ./2/structs/structs.cpp(s)
Начинаем испытания просто с жирной структуры в аргументе функции, а именно там теперь находится статический массив с 100000000 элементами, если добавить еще 0 то стек переполнится. Итак, смотрим на листинг и делаем выводы. Обратим внимание на этот кусок:

```
        ...
main:
        ...
    leaq    -399998976(%rsp), %r11
.LPSRL0:
    subq    $4096, %rsp
    orq     $0, (%rsp)
    cmpq    %r11, %rsp
    jne     .LPSRL0
    ...
```

Тут происходит следующее, пока мы, как я понимаю, не достигнем значения $-399998976(\%rsp)$, вот вангую, что это размер моего стека, хотя это где-то 380 Мбайт, многовато.
Короче, по структуре он положил все это дело в локальный стек, там реально теперь лежит эта громадина. Опа, опа, смотрите что там появилось $callmemcpy@PLT$, перед этим он расширил стек еще на дофига $subq\$400000024, \%rsp$, видимо он реально скопировал все это в стек сверху, вот почему так важны референсы...

Вернем структуру
Такс, давайте теперь вернем структуру. Код в той же папке, в *ret_stru.cpp*. Здесь в main-е выделилось нужное колличество байт стека, далее в вызов функции передается указатель на конец стека, куда функция аккуратненько складывает все аргументы структуры. Собственно тут мы уже посмотрели на жирную структуру в локальной переменной. Напомню листинг по адресу *./2/structs/ret_stru.s* гита.

Если менять размер структуры, то меняются очевидные вещи, типо выделенной памяти от стека, но из интересного в один момент оно улетит с ошибкой, об этом я писал уже. На 32 ситуация в целом аналогичная.

6. Ластово будем ломать стек. Итак напишем:

```
int mama_mia(int a){
    return mama_mia(a);
```

```
}

int main() {
    mama_mia(4);
    return 0;
}
```

Для понимания картины нам достаточно листинга функции:

```
_Z8mama_miai:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %edi
    call    _Z8mama_miai
    leave
    .cfi_def_cfa 7, 8
    ret
```

Чуть допишем, чтобы выводить текущее положение rbp.

```
    ...
    subq    $16, %rsp
    movl    %edi, -4(%rbp)

    movq    %rbp, %rsi
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    call    printf@PLT

    movl    -4(%rbp), %eax
    movl    %eax, %edi
    ...
```

Самая большая проблема - это поймать первое значение, с концом стека вопросов то нет, видимо надо жестко прокачать реакцию.
И вот моя реакция не подвела: -670443552 -> -678824128, получилось без немного 8 Мбайт (7.992340087890625), думал будет побольше, конечно. Сделаем так еще 2 раза и сравним результаты.
-670443552 -> -678824128 => 7.992340087890625 Мбайт;
-131822672 -> -140204240 => 7.9932861328125 Мбайт;
-222453328 -> -230836432 => 7.9947509765625 Мбайт;
Начинается отличие собственно только после 2 знака, так что локального стека у меня всего навсего 8 Мбайт.