

CAOS 4 – asm & stdlib

Manakov Danila

MIPT

2 октября 2022 г.

Допустим вы скомпилировали проект

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hey, hi!\n");
5  }
```

Т.к. вы немного поучили ассемблер)) Вы знаете что 'вызов' функции (почти) эквивалентен goto (= call / bl)

Внимание вопрос! А где находится printf?

Допустим вы скомпилировали проект

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hey, hi!\n");
5  }
```

Т.к. вы немного поучили ассемблер)) Вы знаете что 'вызов' функции (почти) эквивалентен goto (= call / bl)

Внимание вопрос! А где находится printf?

Понятно, что 'где-то в памяти'... А где конкретнее?

Загадка

Давайте посмотрим на такой код:

```
1  #include <stdio.h>
2
3  // вопрос для заскучавших: А ЗАЧЕМ МНЕ СТРОЧКА 4?)
4  extern void _start(void);
5  void foo(void) {}
6
7  int global = 0;
8
9  int main() {
10     int local = 0;
11
12     printf("_start   : 0x%x\n", (unsigned int)_start);
13     printf("main     : 0x%x\n", (unsigned int)main);
14     printf("foo      : 0x%x\n", (unsigned int)foo);
15     printf("printf   : 0x%x\n", (unsigned int)printf);
16     printf("&global  : 0x%x\n", (unsigned int)&global);
17     printf("&local   : 0x%x\n", (unsigned int)&local);
18 }
```

Собираем...

```
1 $> make
2 arm-linux-gnueabi-gcc -static main.c
```

Собираем...

```
1 $> make
2 arm-linux-gnueabi-gcc -static main.c
```

Запускаем...

```
1 $> make run
2 qemu-arm -L /home/dmanakov/sysroot a.out
```

И весь attention на вывод

```
1  _start   : 0xfefde4a5
2  main     : 0xfefde5c3
3  foo      : 0xfefde5b5
4  printf   : 0xfe6dba41
5  &global  : 0xfefef00c
6  &local   : 0xfefdd3c8
```

А вам вывод не кажется странным???

И весь attention на вывод

```
1  _start   : 0xfefde4a5
2  main     : 0xfefde5c3 <--
3  foo      : 0xfefde5b5
4  printf   : 0xfe6dba41
5  &global  : 0xfefef00c <--
6  &local   : 0xfefdd3c8 <--
```

А вам вывод не кажется странным???

И весь attention на вывод

```
1  _start   : 0xfefde4a5 <--
2  main     : 0xfefde5c3 <--
3  foo      : 0xfefde5b5 <--
4  printf   : 0xfe6dba41
5  &global  : 0xfefef00c
6  &local   : 0xfefdd3c8
```

А вам вывод не кажется странным???

Запускаем отладчик...

Запускаем отладчик...

Пы.Сы. gdb фигня, учите radare2

Начнем с ф-ии `_start` (точка входа)

```
1  ;-- section..text:
2  ;-- .text:
3  ;-- _start:
4  ...
5  pop {r1}                ; int argc
6  mov r2, sp              ; char **ubp_av
7  ...
8  ldr.w r3, [s1, r3]       ; func init
9  ldr r0, aav.0x00000044   ; [0x4e8:4]=68
10 ldr.w r0, [s1, r0]       ; func main
11 ; int __libc_start_main(
12 ;     func main,
13 ;     int argc,
14 ;     char **ubp_av,
15 ;     func init,
16 ;     func fini,
17 ;     func rtld_fini,
18 ;     void *stack_end
19 ; )
20 blx sym.imp.__libc_start_main
```

Ф-ия `sym.__libc_start_main` вызывает `main`

```
1 ; int sym.imp.__libc_start_main (func main, int argc, ...)
2 add ip, pc, 0
3 add ip, ip, 0x10000
4 ; DATA XREF from sym.imp.__libc_start_main @ 0x480
5 ldr pc, [ip, 0xb44]! ; <--- БОТ ТУТЬ
```

В ф-ии main по сути есть мало чего интересного
(вот кусок кода, который печатает адрес foo)

1		<i>; sym.foo</i>
2	<code>mov r1, r3</code>	
3	<code>ldr r3, aav.0x00000100</code>	<i>; [0x680:4]=256</i>
4	<code>add r3, pc</code>	<i>; 0x70c ; "foo : 0x%x\n"</i>
5	<code>mov r0, r3</code>	<i>; const char *format</i>
6	<code>blx sym.imp.printf</code>	<i>; int printf(const char *format)</i>
7	<code>ldr r3, aav.0x00000030</code>	<i>; [0x684:4]=48</i>
8	<code>ldr r3, [r4, r3]</code>	<i>; 0x10fe4</i>

Вот кусок кода с `sym.imp.printf`

```
1 int sym.imp.printf (const char *format);
2 add ip, pc, 0
3 add ip, ip, 0x10000
4 ; DATA XREF from sym.imp.printf @ 0x468
5 ldr pc, [ip, 0xb54]!
```

На этот раз этот кусок кода точно требует пояснений

Мы загружаем в `pc` (instruction pointer) значение из ячейки по адресу $pc + 0x10000 + 0xb54$

Вот кусок кода с `sym.imp.printf`

```
1 int sym.imp.printf (const char *format);
2 add ip, pc, 0
3 add ip, ip, 0x10000
4 ; DATA XREF from sym.imp.printf @ 0x468
5 ldr pc, [ip, 0xb54]!
```

На этот раз этот кусок кода точно требует пояснений

Мы загружаем в `pc` (instruction pointer) значение из ячейки по адресу $pc + 0x10000 + 0xb54$

ну... почти...

Происходит jump по этому адресу

```
1  ...
2  ; section..plt ; sym..plt; RELOC 32 __cxa_finalize
3  0x00010fc0      .dword 0x00000448
4
5  ; section..plt ; sym..plt; RELOC 32 printf
6  0x00010fc4      .dword 0x00000448
7
8  ;-- reloc.__stack_chk_fail:
9  ; section..plt ; sym..plt; RELOC 32 __stack_chk_fail
10 0x00010fc8      .dword 0x00000448
11 ...
```

Давайте тут закончим и запустим gdb

```
all:
    arm-linux-gnueabi-gcc -static main.c
    arm-linux-gnueabi-gcc main.c

run:
    # qemu-arm -L /home/dmanakov/sysroot a.out
    qemu-arm -L /usr/arm-linux-gnueabi-hf a.out

clean:
    rm -rf a.out

(base) [dmanakov@NOTEBOOK-k0zF55 ~/caos/04-asm-stdlib/02-c-adr]
$ qemu-arm -L /usr/arm-linux-gnueabi-hf -g 4321 a.out
start : 0xfefde4a5
main : 0xfefde5c3
foo : 0xfefde5b5
printf : 0xfefde4a1
global : 0xfefde40c
local : 0xfefdd3d8
(base) [dmanakov@NOTEBOOK-k0zF55 ~/caos/04-asm-stdlib/02-c-adr]
$ qemu-arm -L /usr/arm-linux-gnueabi-hf -g 4321 a.out
qemu-arm: QEMU: Terminated via GDBstub
(base) [dmanakov@NOTEBOOK-k0zF55 ~/caos/04-asm-stdlib/02-c-adr]
$ vim Makefile
(base) [dmanakov@NOTEBOOK-k0zF55 ~/caos/04-asm-stdlib/02-c-adr]
$ make clean
rm -rf a.out
(base) [dmanakov@NOTEBOOK-k0zF55 ~/caos/04-asm-stdlib/02-c-adr]
$ make
arm-linux-gnueabi-hf-gcc -static main.c
arm-linux-gnueabi-hf-gcc -ggdb main.c
(base) [dmanakov@NOTEBOOK-k0zF55 ~/caos/04-asm-stdlib/02-c-adr]
$ qemu-arm -L /usr/arm-linux-gnueabi-hf -g 4321 a.out
start : 0xfefde4a5
main : 0xfefde5c3
foo : 0xfefde5b5
printf : 0xfefde4a1
global : 0xfefde40c
local : 0xfefdd3d8
(base) [dmanakov@NOTEBOOK-k0zF55 ~/caos/04-asm-stdlib/02-c-adr]
$ qemu-arm -L /usr/arm-linux-gnueabi-hf -g 4321 a.out
start : 0xfefde4a5
```

```
0xfefde468 <printf@plt>      add    r12, pc, #0, 12
0xfefde46c <printf@plt+4>    add    r12, r12, #16, 20
0xfefde470 <printf@plt+8>    ldr     pc, [r12, #2960]! ; 0x100
0xfefde474 <_stack_chk_fail@plt> add    r12, pc, #0, 12
0xfefde478 <_stack_chk_fail@plt+4> add    r12, r12, #16, 20 ; 0x100
0xfefde47c <_stack_chk_fail@plt+8> ldr     pc, [r12, #2892]! ; 0xb4c
0xfefde480 <_libc_start_main@plt> add    r12, pc, #0, 12
0xfefde484 <_libc_start_main@plt+4> add    r12, r12, #16, 20 ; 0x100
0xfefde488 <_libc_start_main@plt+8> ldr     pc, [r12, #2884]! ; 0xb44
0xfefde48c <_gmon_start_@plt> add    r12, pc, #0, 12
0xfefde490 <_gmon_start_@plt+4> add    r12, r12, #16, 20 ; 0x100
0xfefde494 <_gmon_start_@plt+8> ldr     pc, [r12, #2876]! ; 0xb3c
0xfefde498 <abort@plt>      add    r12, pc, #0, 12
0xfefde49c <abort@plt+4>    add    r12, r12, #16, 20 ; 0x100
0xfefde4a0 <abort@plt+8>    ldr     pc, [r12, #2868]! ; 0xb34
0xfefde4a4 <_start>        mov.w   r11, #0
0xfefde4a8 <_start+4>       mov.w   lr, #0
0xfefde4ac <_start+8>       pop      {r1}
0xfefde4ae <_start+10>      mov     r2, sp
0xfefde4b0 <_start+12>      push     {r2}
0xfefde4b2 <_start+14>      push     {r0}
0xfefde4b4 <_start+16>      ldr.w    r10, [pc, #36] ; 0xfefde4dc <
0xfefde4b8 <_start+20>      ldr     r3, pc, #32 ; (adr r3, 0xfef
0xfefde4ba <_start+22>      add     r10, r3
0xfefde4bc <_start+24>      ldr.w    r12, [pc, #32] ; 0xfefde4e0 <

remote Thread 1,517 In: printf@plt      L?? PC: 0xfefde468
Do you need "set solib-search-path" or "set sysroot"?

Breakpoint 1, main () at main.c:8
(gdb) b printf
Breakpoint 2 at 0xfefde460
(gdb) c
Continuing.

Breakpoint 2, 0xfefde468 in printf@plt ()
(gdb) c
Continuing.

Breakpoint 2, 0xfefde468 in printf@plt ()
(gdb) _
```

0) @ gdb-multiarch "NOTEBOOK-k0zF55" 03:39 28-Sep-2

Маааааленький спойлер: до ф-ии printf мы бы не дошли, ибо она динамическая

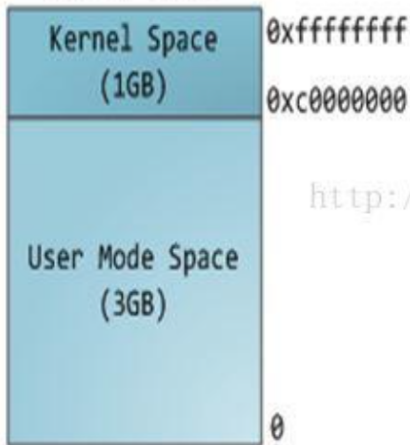
А теперь по-нормальному)

Сейчас кажется, что адреса функций и переменных выдаются чуть ли не случайно.

Очевидно, это не так)) Сейчас пройдемся по теории

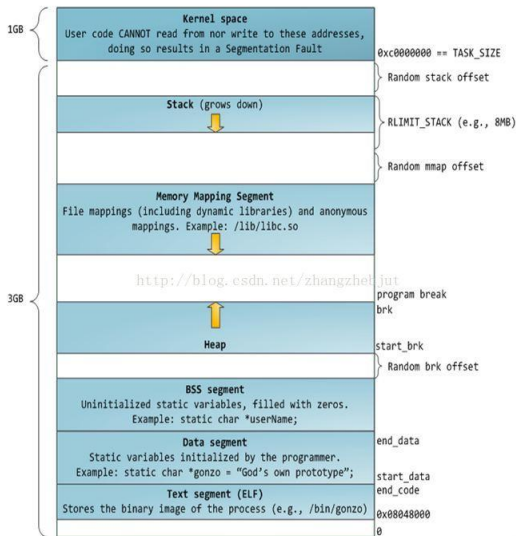
Адресное пространство процесса

Linux User/Kernel
Memory Split



<http://>

Адресное пространство процесса



Кажется [тут](http://blog.csdn.net/zhangzhejun) написано неплохо, можете почитать)

Кратенько про секции

Секция - кусок памяти с названием и определенными правами на неё

- .text
- .bss
- .data
- .stack

Кратенько про секции

Секция - кусок памяти с названием и определенными правами на неё

- .text
- .bss
- .data
- .stack
- .rodata
- .insert_your_section_name

Кратенько про секции

Секция - кусок памяти с названием и определенными правами на неё

- .text
- .bss
- .data
- .stack
- .rodata
- .insert_your_section_name

Внимание вопрос - а зачем они нужны?

Чем глобальные переменные отличаются от локальных?

локальные

- 1 лежат на стеке
- 2 'создаются' сразу после входа в функцию
- 3 'удаляются' перед возвратом

Локальные переменные

```
1 void foo_bar_func() {  
2     uint16_t a = 0x00ff;  
3     uint32_t b = 0xffffeeed;  
4 }
```

```
1 ; sym.foo_bar_func ();  
2 ; var int16_t var_ch @ sp+0x0  
3 push {r7} ; main.c:6 void foo_bar_func() {  
4 sub sp, 0xc ; look at me!!!  
5 add r7, sp, 0  
6 movs r3, 0xff  
7 strh r3, [r7, 2]  
8 movw r3, 0xeeed ; main.c:9 }  
9 movt r3, 0xfffe  
10 str r3, [r7, 4]  
11 nop ; main.c:12 foo_bar_func();  
12 adds r7, 0xc ; look at me!!!  
13 mov sp, r7  
14 ldr r7, [sp], 4  
15 bx lr
```

Локальные переменные

```
1 void foo_bar_func() {
2     uint16_t a = 0x00ff;
3     uint32_t b = 0xffffeeed;
4 }

; sym.foo_bar_func ();
; var int16_t var_ch @ sp+0x0
3 push {r7} ; main.c:6 void foo_bar_func() {
4 sub sp, 0xc ; look at me!!!
5 add r7, sp, 0
6 movs r3, 0xff
7 strh r3, [r7, 2]
8 movw r3, 0xeeed ; main.c:9 }
9 movt r3, 0xfffe
10 str r3, [r7, 4]
11 nop ; main.c:12 foo_bar_func();
12 adds r7, 0xc ; look at me!!!
13 mov sp, r7
14 ldr r7, [sp], 4
15 bx lr
```

Умные слова spetial for семинар - 'пролог' и 'эпилог'

Локальные переменные

Вопрос на понимание: почему так делать НЕ надо?

```
1 int* foo() {  
2     int a = 123;  
3     return &a;  
4 }
```

глобальные

- ❶ лежат в секции `.data` (правда почти всегда)
- ❷ 'вшиты' напрямую в elf (правда почти всегда)
- ❸ находятся в таблице символов (`objdump -t`)

Пример создания глобальной переменной на asm

```
1 // foo.s
2 .data
3 .global MY_UI16
4 MY_UI16: .word 0xfffe
```

Кто такой '.word'?

```
1 // main.c
2 #include <stdio.h>
3 #include <stdint.h>
4
5 extern uint16_t MY_UI16;
6
7 int main() {
8     foo_bar_func();
9     printf("%x\n", MY_UI16);
10    return 0;
11 }
```

А зачем нужен 'extern'?

Пример: вызов библиотечной ф-ии

Поскольку ф-ии - тоже метки, можем делать так же)))

Из нетривиальных вещей - нужно сохранить адрес возврата (и выровнять стек!)

БУДЬТЕ ВНИМАТЕЛЬНЕЕ С CALLEE SAVED REGISTERS!!!

Этот пример так не то чтоб делает

```
1  .global main
2
3  .text
4  main:
5      str x30, [sp, #-16]
6      ldr x0, [hw_str]
7      bl printf
8      mov x0, 0
9      ldr x30, [sp], #16
10     ret
11
12 .data
13     hw_str: .ascii "Hello, asm!\n"
```

Задание: Реализуйте на языке ассемблера armv8 (AArch64) функцию *calculate*, которая вычисляет значение выражения: $R = (A * B) + (C * D)$, где A, B, C, D - это глобальные переменные типа $uint64_t$, объявленные во внешнем модуле компиляции, а R , - глобальная переменная типа $uint64_t$ в текущем модуле компиляции.

Разбор нулевки

```
1  .text
2  .global calculate
3  .global R           ; указываем, что переменная глобальная
4
5  calculate:
6      adr x1, A        ; A, B, C, D - просто метки
7      adr x2, B        ; поэтому их можно использовать вместо адресов
8      adr x3, C
9      adr x4, D
10     adr x5, R
11     ldr x1, [x1]
12     ldr x2, [x2]
13     ldr x3, [x3]
14     ldr x4, [x4]
15     mul x0, x1, x2
16     mul x1, x3, x4
17     add x0, x0, x1
18     str x0, [x5]
19     ret
20
21 .data
22     R: .quad         ; объявляем переменную
```

1. А давайте что-нибудь хакнем)

Задача 1 (difficulty: baby): Не зная пароль, добраться до секрета

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char user_input[8];
6      char password[8] = PASSWD;
7
8      printf("Введите пароль: ");
9      scanf("%s", user_input);
10
11     if (strcmp(user_input, password) == 0) {
12         printf("SECRET: вы красавчик!\n");
13     } else {
14         printf("Permission denied!\n");
15     }
16     return 0;
17 }
```

1. А давайте что-нибудь хакнем)

Неправильный пароль:

```
1 qemu-arm -L /usr/arm-linux-gnueabihf a.out
2 Введите пароль: wrong
3 Permission denied!
```

Правильный пароль:

```
1 qemu-arm -L /usr/arm-linux-gnueabihf a.out
2 Введите пароль: 1234567
3 SECRET: вы красавчик!
```

1. А давайте что-нибудь хакнем)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char user_input[8];           // давайте введем что-нибудь длинное
6      char password[8] = PASSWD;    // чтобы перетереть пароль
7
8      printf("Введите пароль: ");
9      scanf("%s", user_input);
10
11     if (strcmp(user_input, password) == 0) {
12         printf("SECRET: вы красавчик!\n");
13     } else {
14         printf("Permission denied!\n");
15     }
16     return 0;
17 }
```

1. А давайте что-нибудь хакнем)

вАпрОс: получится или нет?)

```
1 qemu-arm -L /usr/arm-linux-gnueabihf a.out
2 Введите пароль: 76543217654321
3 \pause
4 Permission denied!
```

1. А давайте что-нибудь хакнем)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char user_input[8];          // '76543217'
6      char password[8] = PASSWD;   // '654321*\0'
7
8      printf("Введите пароль: ");
9      scanf("%s", user_input);
10
11     if (strcmp(user_input, password) == 0) {
12         printf("SECRET: вы красавчик!\n");
13     } else {
14         printf("Permission denied!\n");
15     }
16     return 0;
17 }
```

1. А давайте что-нибудь хакнем)

Тогда давайте будем

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char user_input[8];           // '76543217'
6      char password[8] = PASSWD;   // '654321*\0'
7
8      printf("Введите пароль: ");
9      scanf("%s", user_input);
10
11     if (strcmp(user_input, password) == 0) {
12         printf("SECRET: вы красавчик!\n");
13     } else {
14         printf("Permission denied!\n");
15     }
16     return 0;
17 }
```


2. А давайте что-нибудь хакнем)

Задача 2 (difficulty: not-so-easy): Добраться до секрета

—
сори, эту (или похожую) задачу разберем когда речь будет идти про x86

я не готов потратить ещё кучу времени на костыли
arm-программ и компиляторов)

```
1  #include <stdio.h>
2
3  void win_win() {
4      printf("SECRET: кул-хацкер");
5      exit(0);
6  }
7
8  int main() {
9      char name[12];
10
11     scanf("%s", name);
12     printf("By by, %s", name);
13     return 0;
14 }
```