CAOS 4 – asm & stdlib

Manakov Danila

MIPT

2 октября 2022 г.

Загадка

Допустим вы скомпилировали проект

```c
#include <stdio.h>

int main() {
    printf("Hey, hi!\n");
}
```

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

Внимание вопрос! А где находится printf?

Загадка

Допустим вы скомпилировали проект

```c
#include <stdio.h>

int main() {
    printf("Hey, hi!\n");
}
```

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:

```c
#include <stdio.h>

// вопрос для заскучавших: А ЗАЧЕМ МНЕ СТРОЧКА 4?)
extern void _start(void);
void foo(void) {}

int global = 0;

int main() {
    int local = 0;

    printf("_start  : 0x%x\n", (unsigned int)_start);
    printf("main    : 0x%x\n", (unsigned int)main);
    printf("foo     : 0x%x\n", (unsigned int)foo);
    printf("printf  : 0x%x\n", (unsigned int)printf);
    printf("&global : 0x%x\n", (unsigned int)&global);
    printf("&local  : 0x%x\n", (unsigned int)&local);
}
```

Загадка

Собираем...

```
$> make
arm-linux-gnueabihf-gcc -static main.c
```

Загадка

Собираем...

```
$> make
arm-linux-gnueabihf-gcc -static main.c
```

Запускаем...

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

Загадка

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???

Загадка

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???

Загадка

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???

Отгадка

Запускаем отладчик...

Отгадка

Запускаем отладчик...

Пы.Сы. gdb фигня, учите radare2

Отгадка

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, [sl, r3]                ; func init
9  ldr r0, aav.0x00000044            ; [0x4e8:4]=68
10 ldr.w r0, [sl, 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
```

Отгадка

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]!   ; <--- ВОТ ТУТЬ
```

Отгадка

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

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

Отгадка

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 $pc + 0x10000 + 0xb54$

Отгадка

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...

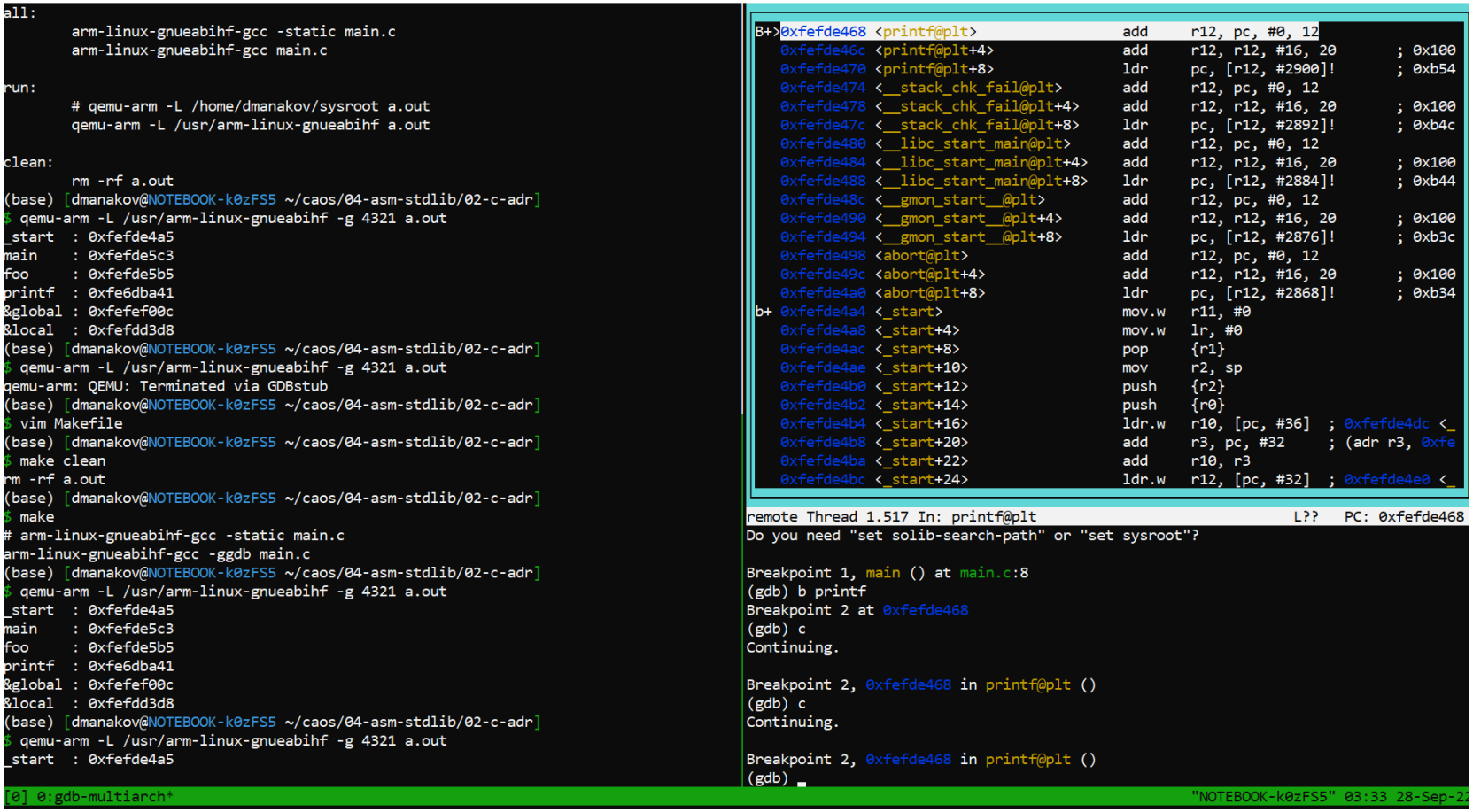
Отгадка

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  ...
```

Отгадка

Давайте тут закончим и запустим gdb

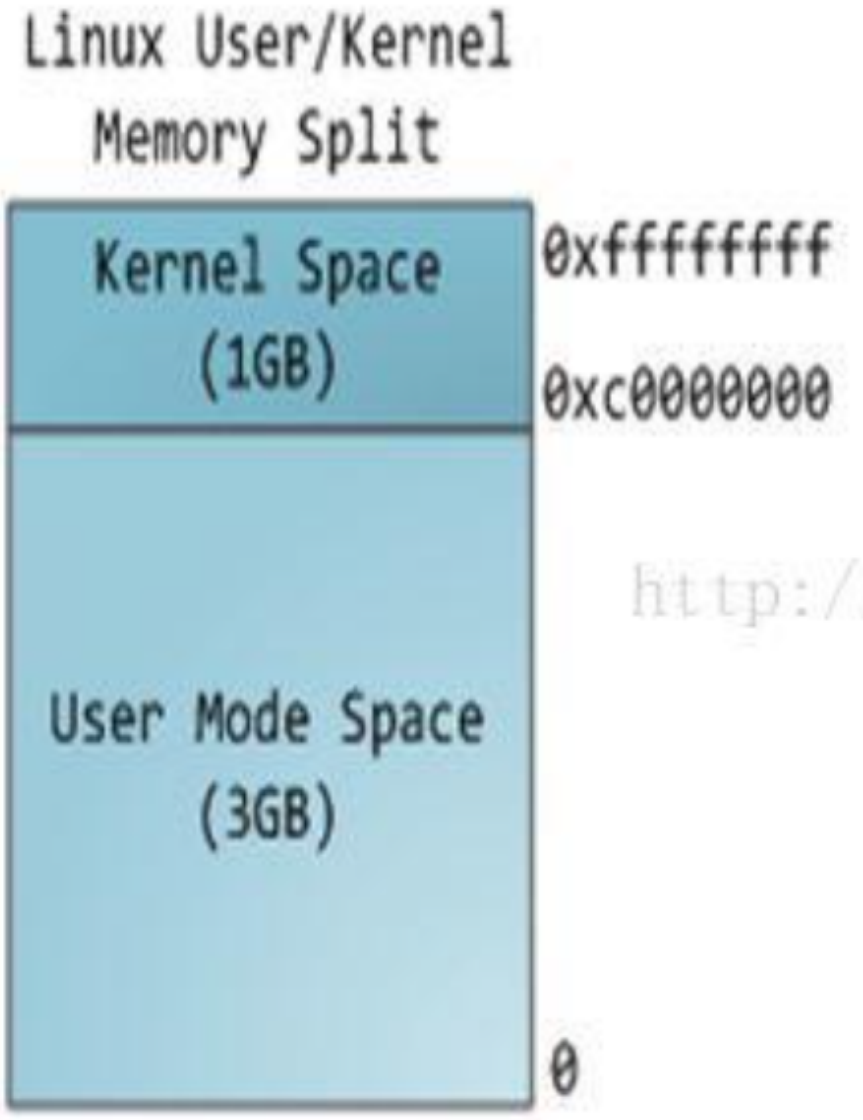


Маааааленький спойлер: до ф-ии printf мы бы не дошли, ибо она динамическая

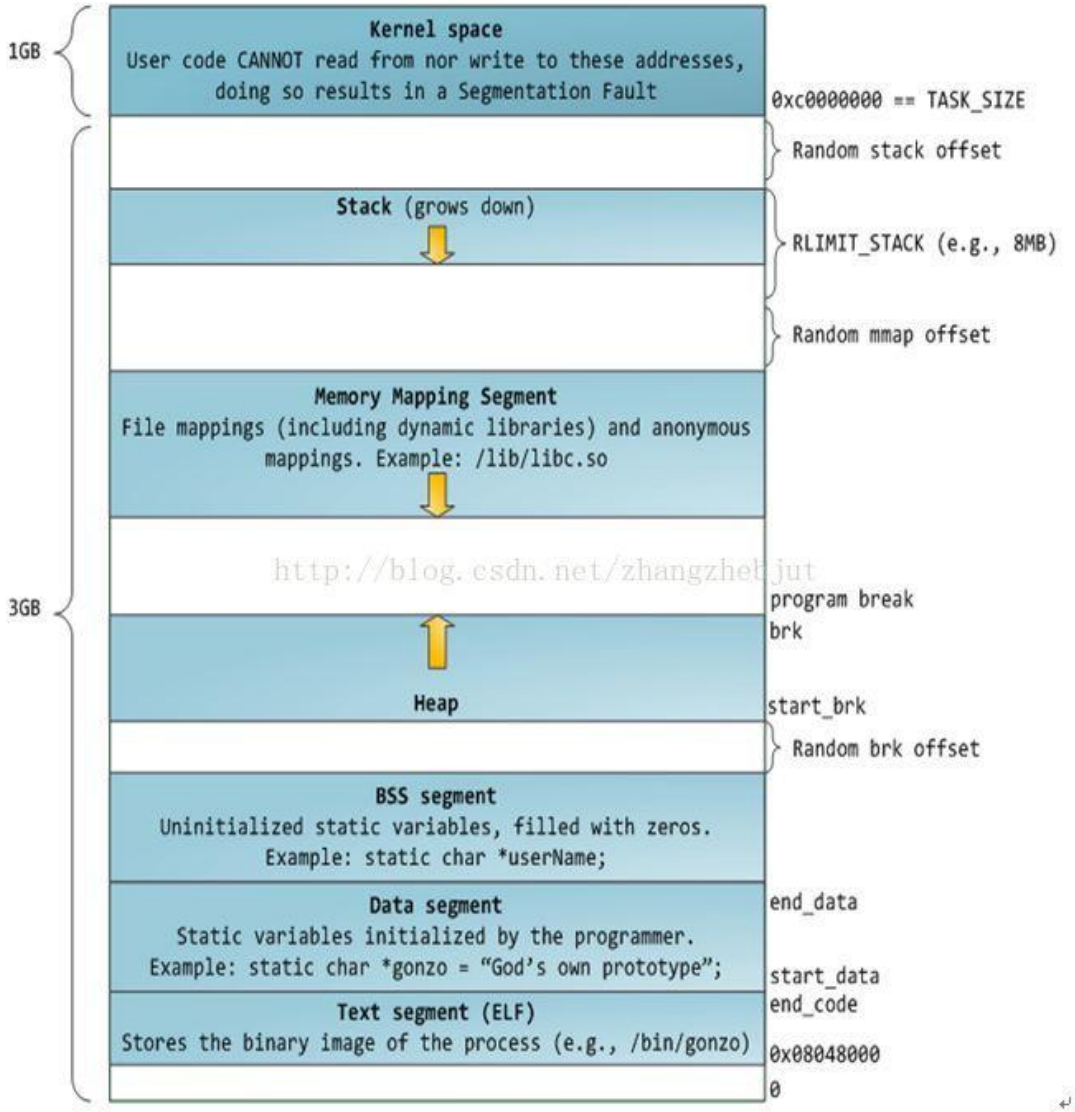
А теперь по-нормальному)

Сейчас кажется, что адреса функций и переменных выдаются чуть ли не случайно.
Очевидно, это не так)) Сейчас пройдемся по теории

Адресное пространство процесса



Адресное пространство процесса



Кажется тут 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?

Глобальные и локальные переменные

Чем глобальные переменные отличаются от локальных?

Локальные переменные

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 = 0xfffeeeed;
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 = 0xfffeeeed;
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
```

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 переменные

глобальные

1. лежат в секции .data (правда почти всегда)
2. 'вшиты' напрямую в elf (правда почти всегда)
3. 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"
```

Разбор нулевки

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

```c
#include <stdio.h>
#include <string.h>

int main() {
    char user_input[8];
    char password[8] = PASSWD;

    printf("Введите пароль: ");
    scanf("%s", user_input);

    if (strcmp(user_input, password) == 0) {
        printf("SECRET: вы красавчек!\n");
    } else {
        printf("Permission denied!\n");
    }
    return 0;
}
```

1. А давайте что-нибудь хакнем)

Неправильный пароль:

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

Правильный пароль:

```
qemu-arm -L /usr/arm-linux-gnueabihf a.out
Введите пароль: 1234567
SECRET: вы красавчек!
```

1. А давайте что-нибудь хакнем)

```c
#include <stdio.h>
#include <string.h>

int main() {
    char user_input[8];          // давайте введем что-нибудь длинное
    char password[8] = PASSWD;   // чтобы перетереть пароль

    printf("Введите пароль: ");
    scanf("%s", user_input);

    if (strcmp(user_input, password) == 0) {
        printf("SECRET: вы красавчек!\n");
    } else {
        printf("Permission denied!\n");
    }
    return 0;
}
```

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)

```c
#include <stdio.h>
#include <string.h>

int main() {
    char user_input[8];           // '76543217'
    char password[8] = PASSWD;    // '654321*\0'

    printf("Введите пароль: ");
    scanf("%s", user_input);

    if (strcmp(user_input, password) == 0) {
        printf("SECRET: вы красавчек!\n");
    } else {
        printf("Permission denied!\n");
    }
    return 0;
}
```

1. А давайте что-нибудь хакнем)

Тогда давайте будем

```c
#include <stdio.h>
#include <string.h>

int main() {
    char user_input[8];          // '76543217'
    char password[8] = PASSWD;   // '654321*\0'

    printf("Введите пароль: ");
    scanf("%s", user_input);

    if (strcmp(user_input, password) == 0) {
        printf("SECRET: вы красавчек!\n");
    } else {
        printf("Permission denied!\n");
    }
    return 0;
}
```

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)

```c
#include <stdio.h>

void win_win() {
    printf("SECRET: кул-хацкер");
    exit(0);
}

int main() {
    char name[12];

    scanf("%s", name);
    printf("By by, %s", name);
    return 0;
}
```