Task1. Code analysis

Part 1. Startup

We could start with info about binary Ghidra showed us. As can be seen, final binary requires libc shared object and contains not only task1.c, but also C runtime library, startup routines (crt.o, crtstuff.c) provided by cross-compiler.

```
ELF File Type:              executable
ELF Original Image Base:    0x10000
ELF Prelinked:              false
ELF Required Library [   0]:libc.so.6
ELF Source File [   0]:     /usr/lib/gcc-cross/arm-linux-gnueabi/9/../../../../arm-linux-gnueabi/lib/../lib/crt1.o
ELF Source File [   1]:     /usr/lib/gcc-cross/arm-linux-gnueabi/9/../../../../arm-linux-gnueabi/lib/../lib/crti.o
ELF Source File [   2]:     /usr/lib/gcc-cross/arm-linux-gnueabi/9/../../../../arm-linux-gnueabi/lib/../lib/crtn.o
ELF Source File [   3]:     crtstuff.c
ELF Source File [   4]:     task1.c
ELF Source File [   5]:     elf-init.oS
ELF Source File [   6]:     crtstuff.c
Executable Format:          Executable and Linking Format (ELF)
Executable Location:        /home/incomprehensible/OxygenSoftw/task1
Executable MD5:             f58d35ce2b64c913f64f56c56702b9a3
Executable SHA256:          94e26b746b6d954609d04675ffe87f4b3ad6e5ab8db58c464875c4ebad2249c9
FSRL:                       file:///home/incomprehensible/OxygenSoftw/task1?MD5=f58d35ce2b64c913f64f56c56702b9a3
Relocatable:                false
```

Известно, что перед вызовом main необходима различная инициализация (libc, runtime env etc), как и завершающий код после main, поэтому линкер линкует этот код в каждый elf.

```
        //
spsr = 0x0  (Default)
45 4c       Elf32_Ehdr
01 01
00 00 ...
7f              db      7Fh
45 4c 46        ds      "ELF"
01              db      1h
01              db      1h
01              db      1h
```

Файл начинается с числа 7Fh и символами "ELF". Магические числа, которые в нашем случае формируют строку "ELF", нужны для дифференцирования ОС между форматами файлов. Остальные числа до программы формируют хэдер. Он содержит инфу о том, сколько памяти нужно программе, по какому адресу расположить и тд.

```
 00 00
          dw        2h                          e_type
          dw        28h                         e_machine
 00 00    ddw       1h                          e_version
 01 00    ddw       _start                      e_entry
 00 00    ddw       Elf32_Phdr_ARRAY_00010...   e_phoff
 00 00    ddw       Elf32_Shdr_ARRAY__elfS...   e_shoff
 00 05    ddw       5000200h                    e_flags
          dw        34h                         e_ehsize
          dw        20h                         e_phentsize
          dw        9h                          e_phnum
          dw        28h                         e_shentsize
```

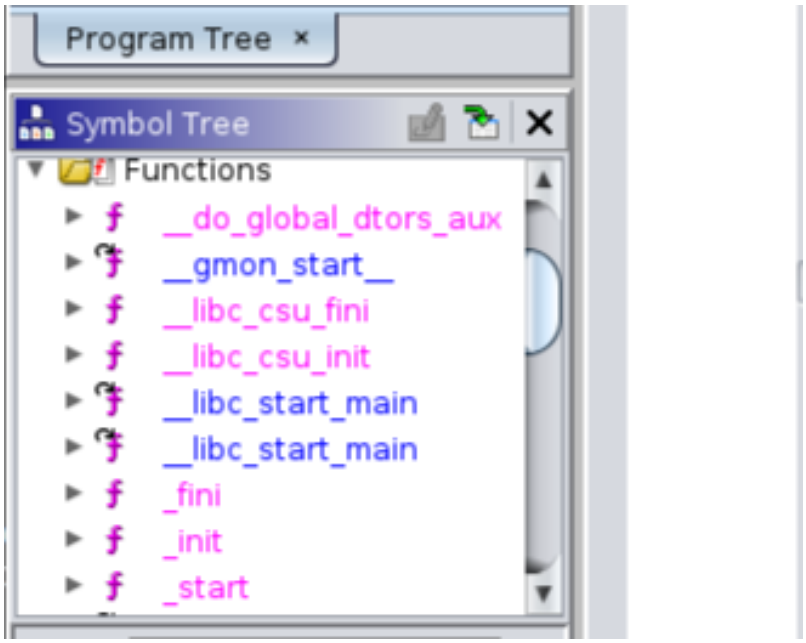
Как известно, _start - это Entry Point программы, указывает, где начинается ее код.

```
          undefined _start(undefined param_1, undefined param_2, u...
ined        r0:1          <RETURN>
ined        r0:1          param_1
ined        r1:1          param_2
ined        r2:1          param_3
ined        r3:1          param_4
ined4       Stack[0x0]:4  param_5                          XREF[1]:     000103b4(W)
ined4       Stack[-0x4]:4 local_4                        XREF[1]:     000103b8(W)
ined4       Stack[-0x8]:4 local_8                        XREF[1]:     000103c0(W)
          _start                                       XREF[3]:     Entry Point(*), 00010018(*),
                                                                    _elfSectionHeaders::00000214(

00 b0 a0 e3     mov       r11,#0x0
00 e0 a0 e3     mov       lr,#0x0
04 10 9d e4     ldr       param_2,[sp],#0x4
0d 20 a0 e1     cpy       param_3,sp
04 20 2d e5     str       param_3,[sp,#param_5]!
04 00 2d e5     str       param_1,[sp,#local_4]!
10 c0 9f e5     ldr       r12,[->__libc_csu_fini]                  = 00010688
04 c0 2d e5     str       r12=>__libc_csu_fini,[sp,#local_8]!
0c 00 9f e5     ldr       param_1=>main,[->main]                   = 000105a8
0c 30 9f e5     ldr       param_4=>__libc_csu_init,[->__libc_csu_init]  = 00010628
e8 ff ff eb     bl        __libc_start_main                        undefined __libc_start_mai
f0 ff ff eb     bl        abort                                    void abort(void)
          -- Flow Override: CALL_RETURN (CALL_TERMINATOR)
```

Сама функция. Параметры загружаются и инициализируются относительно stack pointer. Также загружаются адреса функций инициализации libc - libc_*_fini/init. Что-то сохраняется на стеке, что-то грузится в регистр. Наша main - тоже параметр. Вызывается, как обычно, __libc_start_main. Она никогда не возвращается, если все хорошо. Иначе послее нее есть привелигированная инструкция abort, (иногда еще halt), генерируется exception.

Дальше следуют рутины инициализации символов с .init_array, __init и др, но для небольшого отчета, думаю, достаточно. Функции, следующие до и после main, указаны здесь:



Часть 2. Main

Смотрим на main.

Сначала с помощью инструкции stmdb на стек сохраняются frame pointer/r11 и link register с адресом возврата, одновременно с декрементом значения sp. sp будет равен значению r11.

```
                        main

000105a8 00 48 2d e9     stmdb       sp!,{ r11 lr }
000105ac 04 b0 8d e2     add         r11,sp,#0x4
000105b0 10 d0 4d e2     sub         sp,sp,#0x10
000105b4 10 00 0b e5     str         r0,[r11,#local_14]
000105b8 14 10 0b e5     str         r1,[r11,#local_18]
000105bc 10 30 1b e5     ldr         r3,[r11,#local_14]
```

Наши аргументы командной строки argc и argv - в регистрах r0 и r1 соответственно.

После пролога функции можно сравнить код с исходником. Здесь проверяем значение argc, и если число аргументов равно 2, branch инструкция условного перехода beq отправляет на метку d0. Иначе записываем в r3 значение для return из функции.

```
3 e3      cmp         r3,#0x2
0 0a      beq         LAB_000105d0
0 e3      mov         r3,#0x0
0 ea      b           LAB_00010618
```

```
int main(int argc, char **argv)
{
        if (argc != 2)
                return 0;
```

Перед вызовом функции get_copy загружаем в r3 адрес нашей строки по смещению и перезагружаем в r0 - первый параметр функции по соглашению о вызове.

```
      LAB_000105d0
1b e5     ldr         r3,[r11,#local_18]
83 e2     add         r3,r3,#0x4
93 e5     ldr         r3,[r3,#0x0]
a0 e1     cpy         r0,r3
ff eb     bl          get_copy
```

Результат из get_copy пришел в r0, сохраняем на стеке и загружаем в r3 для проверки. Проверяем, что copy != NULL.

```
0b e5     str         r0,[r11,#local_c]
1b e5     ldr         r3,[r11,#local_c]
53 e3     cmp         r3,#0x0
00 1a     bne         LAB_00010600
a0 e3     mov         r0,#0xa
ff eb     bl          putchar
00 ea     b           LAB_00010614
```

Если пришел NULL, загружаем значение '\n' в r0 и, вызвав putchar, идем к return. (Интересно, что в коде был вызов printf, но компилятор изменил его на putchar).

```
    char *copy = get_copy(argv[1]);
    if (!copy)
            printf("\n");
    else {
            printf("copy: %s\n", copy);
            free(copy);
    }
    return 0;
```

Если нет, загружаем в r1 сохраненный на стеке адрес строки, относительно смещения по fp. В r0 грузим форматирующую строку из .rodata. вызываем printf с 2мя аргументами. Аналогично готовим параметр для вызова free и фришим память:

```
     LAB_00010600                                                    XREF
0 e5     ldr         r1,[r11,#local_c]
f e5     ldr         r0=>s_copy:_%s_00010698,[PTR_s_copy:

f eb     bl          printf
0 e5     ldr         r0,[r11,#local_c]
f eb     bl          free
```

Наконец подготовка return value, эпилог функции. С помощью ldmia делаем обратную stmdb операцию - загружаем со стека (эффективный адрес начала - sp) обратно в регистры r11 и pc и инкрементируем sp, при этом в pc теперь лежит значение lr со стека, и за одну инструкцию мы восстанавливаем регистры и возвращаемся из функции. (Note: addr для перехода на адрес форматирующей строки).

```
             LAB_00010614                                    XREF[1]:     000105fc(j)
14 00 30 a0 e3     mov         r3,#0x0

             LAB_00010618                                    XREF[1]:     000105cc(j)
18 03 00 a0 e1     cpy         r0,r3
1c 04 d0 4b e2     sub         sp,r11,#0x4
20 00 88 bd e8     ldmia       sp!,{ r11 pc }

             PTR_s_copy:_%s_00010624                         XREF[1]:     main:00010604(R)
24 98 06 01 00     addr        s_copy:_%s_00010698                          = "copy: %s\n"
```

Часть 3. get_copy() и get_len()

После пролога функции, аналогичного main, мы вызываем get_len с передачей в r0 строки, для которой хотим найти длину. Мы должны сохранить адрес строки на стеке, тк get_len изменит r0.

```
          get_copy
0 48 2d e9     stmdb       sp!,{ r11 lr }
4 b0 8d e2     add         r11,sp,#0x4
8 d0 4d e2     sub         sp,sp,#0x18
8 00 0b e5     str         r0,[r11,#local_1c]
8 00 1b e5     ldr         r0,[r11,#local_1c]
4 ff ff eb     bl          get_len
c 00 0b e5     str         r0,[r11,#local_10]
c 30 1b e5     ldr         r3,[r11,#local_10]
0 00 53 e3     cmp         r3,#0x0
1 00 00 1a     bne         LAB_00010518
0 30 a0 e3     mov         r3,#0x0
0 00 00 ea     b           LAB_0001059c
```

В конце делаем проверку на 0 и либо делаем return, либо идем копировать.

Дизассемблер соответствует коду ниже:

```
char *get_copy(const char *s)
{
        size_t len = get_len(s);
        if (!len)
                return NULL;
```

Get_len() возвращает длину строки.

```
size_t get_len(const char *s)
{
        size_t i = 0;

        while (s[i])
                ++i;
        return i;
}
```

Сразу коротко разберу ее дизассемблер.
Пролог функции. В r0 лежит адрес строки, сохраним на стеке. В r3 кладем стартовое значение 0, переносим в локальную переменную i.

```
          get_len

4 b0 2d e5     str         r11,[sp,#local_4]!
0 b0 8d e2     add         r11,sp,#0x0
4 d0 4d e2     sub         sp,sp,#0x14
0 00 0b e5     str         r0,[r11,#local_14]
0 30 a0 e3     mov         r3,#0x0
3 30 0b e5     str         r3,[r11,#local_c]
2 00 00 ea     b           LAB_000104bc
```

Здесь идет адресная арифметика. Тк мы идем по строке посимвольно, инкрементируем i.

```
  LAB_000104b0
5     ldr         r3,[r11,#local_c]
2     add         r3,r3,#0x1
5     str         r3,[r11,#local_c]
```

Мы еще сюда вернемся.
На метке bc грузим в r2 адрес строки, в r3 - значение i. Добавляем к адресу строки r3

оффсет -> r3. Переиспользуем r3, загрузив в него значение символа по полученному адресу с immediate offset 0. Сравниванием его с '\0'. Если строка не закончилась, прыгаем снова на b0.

```
   LAB_000104b0
e5     ldr         r3,[r11,#local_c]
e2     add         r3,r3,#0x1
e5     str         r3,[r11,#local_c]

   LAB_000104bc
e5     ldr         r2,[r11,#local_14]
e5     ldr         r3,[r11,#local_c]
e0     add         r3,r2,r3
e5     ldrb        r3,[r3,#0x0]
e3     cmp         r3,#0x0
1a     bne         LAB_000104b0
```

Возвращаем количество символов i в r0, сделав безусловный переход в ином случае:

```
1a     bne         LAB_000104b0
e5     ldr         r3,[r11,#local_c]
e1     cpy         r0,r3
e2     add         sp,r11,#0x0
e4     ldr         r11=>local_4,[sp],#0x4
e1     bx          lr
```

Вернемся к get_copy. Вызовем malloc с нужной длиной в качестве параметра.

```
     LAB_00010518
.b e5     ldr         r0,[r11,#local_10]
:f eb     bl          malloc
a0 e1     cpy         r3,r0
0b e5     str         r3,[r11,#local_c]
.b e5     ldr         r3,[r11,#local_c]
53 e3     cmp         r3,#0x0
00 1a     bne         LAB_0001053c
a0 e3     mov         r3,#0x0
00 ea     b           LAB_0001059c

     LAB_0001053c
a0 e3     mov         r3,#0x0
0b e5     str         r3,[r11,#local_14]
00 ea     b           LAB_00010574
```

С помощью cpy + str сохраним указатель на выделенную память в локальной переменной copy. Опять проверяем, что malloc не вернул NULL, иначе вернем NULL из функции по метке 9c. На 3c в r3 поместим 0 и инициализируем локальную переменную i. Соответствует коду:

```
char *new = (char*)malloc(len);
if (!new)
        return NULL;
size_t i = 0;
```

Дальше копируем символы строки s в строку new. Помним, что по оффсету от fp #local_c лежит new, #local_lc - s, #local_l4 - i, #local_l0 - len.

```
     LAB_00010548
b e5     ldr         r2,[r11,#local_1c]
b e5     ldr         r3,[r11,#local_14]
2 e0     add         r2,r2,r3
b e5     ldr         r1,[r11,#local_c]
b e5     ldr         r3,[r11,#local_14]
1 e0     add         r3,r1,r3
2 e5     ldrb        r2,[r2,#0x0]
3 e5     strb        r2,[r3,#0x0]
b e5     ldr         r3,[r11,#local_14]
3 e2     add         r3,r3,#0x1
b e5     str         r3,[r11,#local_14]
```

Сначала в r2 грузим адрес s и прибавляем i, затем в r1 грузим адрес new и тоже прибавляем i. Теперь в r2 - s + i, в r3 - new + i. Берем по адресу в r2 со смещением 0 char значение, теперь оно в r2. Копируем его по адресу в r3 + 0 с помощью strb. Снова грузим значение i в r3, инкрементируем и загружаем обратно по адресу i.

```
        LAB_00010574
0 1b e5     ldr         r2,[r11,#local_14]
0 1b e5     ldr         r3,[r11,#local_10]
0 52 e1     cmp         r2,r3
f ff 3a     bcc         LAB_00010548
0 1b e5     ldr         r2,[r11,#local_c]
0 1b e5     ldr         r3,[r11,#local_14]
0 82 e0     add         r3,r2,r3
0 a0 e3     mov         r2,#0x0
0 c3 e5     strb        r2,[r3,#0x0]
0 1b e5     ldr         r3,[r11,#local_c]

        LAB_0001059c
0 a0 e1     cpy         r0,r3
0 4b e2     sub         sp,r11,#0x4
8 bd e8     ldmia       sp!,{ r11 pc }
```

Сравниваем i и len. Если не равны, продолжаем цикл while по метке 48. Иначе берем адрес new, кладем в r3 оффсет i, получаем адрес со смещением и делаем строку new null-terminated. Возвращаем все на место, в r0 <- new, эпилог и return new.