Анализ multiply_by8

```
volatile void multiply_by8(int *arr, int sz)
{
        for (int i = 0; i < sz; ++i)
                arr[i] *= 8;

        printf("After multiplying: ");
        for (int i = 0; i < sz; ++i)
                printf("%d ", arr[i]);
        putchar('\n');
}
```

Тело функции для референса.

```
        int *           r0:4            arr
        int             r1:4            sz
        int             Stack[-0xc]:4   i_1



        int             Stack[-0x10]:4 i



        undefined4      Stack[-0x14]:4 local_14

        undefined4      Stack[-0x18]:4 local_18
```

r0 и r1, содержащие параметры функции, задефайнены в ассемблере как arr и sz.
Пролог функции.

```
                    multiply_by8

000104cc 00 48 2d e9    stmdb       sp!,{ r11 lr }
000104d0 04 b0 8d e2    add         r11,sp,#0x4
000104d4 10 d0 4d e2    sub         sp,sp,#0x10
000104d8 10 00 0b e5    str         arr,[r11,#local_14]
000104dc 14 10 0b e5    str         sz,[r11,#local_18]
000104e0 00 30 a0 e3    mov         r3,#0x0
000104e4 0c 30 0b e5    str         r3,[r11,#i]
000104e8 0d 00 00 ea    b           LAB_00010524
```

Сначала сохраняем на стеке адрес возврата и frame pointer вызывающей функции. Обновляем fp. Выделяем место на стеке под локальные переменные. Также сохраняем

на стеке адрес нашего массива, как и его размер.

Загружаем начальную immediate value по адресу (смещение относительно r11) в локальный счетчик i через r3.

Переходим к циклу.

```
                        LAB_000104ec
000104ec 0c 30 1b e5     ldr         r3,[r11,#i]
000104f0 03 31 a0 e1     mov         r3,r3, lsl #0x2
000104f4 10 20 1b e5     ldr         r2,[r11,#local_14]
000104f8 03 30 82 e0     add         r3,r2,r3
000104fc 00 20 93 e5     ldr         r2,[r3,#0x0]
00010500 0c 30 1b e5     ldr         r3,[r11,#i]
00010504 03 31 a0 e1     mov         r3,r3, lsl #0x2
00010508 10 10 1b e5     ldr         sz,[r11,#local_14]
0001050c 03 30 81 e0     add         r3,sz,r3
00010510 82 21 a0 e1     mov         r2,r2, lsl #0x3
00010514 00 20 83 e5     str         r2,[r3,#0x0]
00010518 0c 30 1b e5     ldr         r3,[r11,#i]
0001051c 01 30 83 e2     add         r3,r3,#0x1
00010520 0c 30 0b e5     str         r3,[r11,#i]
```

Еще раз забираем значение i. Для получения адреса нужной ячейки arr[i] делаем логический сдвиг влево на 2, тем самым умножая i на sizeof(int), в r2 помещаем адресс arr и прибавляем к адресу первого элемента полученное смещение. После чего из памяти нужно загрузить значение элемента массива. Это значение теперь в r2. Затем видим точно такой же процесс получения адреса arr[i], повторяющийся для того, положить туда измененное значение элемента. Умножение происходит с помощью lsl сдвига на 3, то есть r2 *= 8. Поместим r2 в arr[i].

В конце данного блока обновляем счетчик i, инкрементируя.

Здесь проверяем значение i.

```
                        LAB_00010524                                    XR
00010524 0c 20 1b e5     ldr         r2,[r11,#i]
00010528 14 30 1b e5     ldr         r3,[r11,#local_18]
0001052c 03 00 52 e1     cmp         r2,r3
00010530 ed ff ff ba     blt         LAB_000104ec
00010534 5c 00 9f e5     ldr         arr=>s_After_multiplying:_00010868,

00010538 95 ff ff eb     bl          printf
0001053c 00 30 a0 e3     mov         r3,#0x0
00010540 08 30 0b e5     str         r3,[r11,#i_1]
00010544 0a 00 00 ea     b           LAB_00010574
```

Загружаем в r2 значение по адресу локальной переменной. Забираем со стека сохраненное значение размера sz. Если сравнение означает, что r2 меньше, продолжаем идти по массиву, вернувшись на метку цикла 4ec.

Иначе загружаем в r0(=arr) адрес форматирующей строки и вызываем printf. Это соответствует printf("After multiplying: "); в коде.

Несмотря на то, что мы могли бы переиспользовать локальную переменную i, для цикла принтования значений массива после умножения есть другая локальная переменная со

смещением -0xc + 4. В конце блока инициализируем этот счетчик =0.

В конце блока прыгаем сразу на последнюю метку, где происходит проверка i < sz. Блок разобран чуть ниже. Мы вернемся выше на меткк 548, так как еще не прошли массив.

Здесь цикл, в котором мы выводим значение всех элементов массива в stdout. В r0 строка "%d", в r1 значение. После printf инкрементируем счетчик. После чего следует блок проверки счетчика 574.

```
                    LAB_00010548                                    XREF[1]
00010548 08 30 1b e5     ldr         r3,[r11,#i_1]
0001054c 03 31 a0 e1     mov         r3,r3, lsl #0x2
00010550 10 20 1b e5     ldr         r2,[r11,#local_14]
00010554 03 30 82 e0     add         r3,r2,r3
00010558 00 30 93 e5     ldr         r3,[r3,#0x0]
0001055c 03 10 a0 e1     cpy         sz,r3
00010560 34 00 9f e5     ldr         arr=>DAT_0001087c,[PTR_DAT_0001059c]

00010564 8a ff ff eb     bl          printf
00010568 08 30 1b e5     ldr         r3,[r11,#i_1]
0001056c 01 30 83 e2     add         r3,r3,#0x1
00010570 08 30 0b e5     str         r3,[r11,#i_1]
```

Завершающий блок кода. Первая часть проверяет, можно ли завершить цикл печатания обновленного массива.

```
                    LAB_00010574
00010574 08 20 1b e5     ldr         r2,[r11,#i_1]
00010578 14 30 1b e5     ldr         r3,[r11,#local_18]
0001057c 03 00 52 e1     cmp         r2,r3
00010580 f0 ff ff ba     blt         LAB_00010548
00010584 0a 00 a0 e3     mov         arr,#0xa
00010588 8d ff ff eb     bl          putchar
0001058c 00 00 a0 e1     cpy         arr,arr
00010590 04 d0 4b e2     sub         sp,r11,#0x4
00010594 00 88 bd e8     ldmia       sp!,{ r11 pc }
```

Если второй счетчик меньше sz, возвращаемся на метку 548 с циклом printf. Если мы достигли конца, печатаем конец строки, зачем-то копируем из r0 в r0 адрес массива, избавляемся от выделенной под локальные данные память стека и восстанавливаем сохраненный fp / возвращаемся, загрузив адрес возврата в pc.