Лабораторная работа #2

Анализ исполняемых файлов

Цель: узнать, "что там внутри" у программ, написанных на C или C++.

0. Введение в дизассемблирование

Установка зависимостей и ПО

```
x@vbox:~$ sudo apt install -y linux-headers-$(uname -r)
x@vbox:~$ sudo apt install -y build-essential
x@vbox:~$ sudo apt install -y gcc-multilib
x@vbox:~$ sudo apt install -y g++-multilib
```

gcc

Все тестовые примеры, с которыми мы будем работать в этой лабораторной работе (исходники находятся в папке *examples*), необходимо будет компилировать с помощью ***gcc*** [1-2]. Приведём полезные ключи этого компилятора:

- -o proc_name – указание создать исполняемый файл с именем proc_name
- -m32 – собрать под архитектуру x86 (32-битный исполняемый файл)
- -g – собрать с отладочными символами, это сохраняет в числе прочего информацию о названиях функций и переменных.

gdb

Для отладки исполняемого файла можно и нужно использовать отладчик ***gdb*** [3-9]. Полезные команды:

- **set disassembly-flavor intel**
  устанавливает вывод кода в синтаксисе intel (это нормальный), а не AT&T (это *ненормальный*)
- **disas**
  disas main
  disas <label>
  декомпилировать (показать ассемблерный листинг) текущую функцию, функцию main или функцию по метке *label*, соответственно
- **break main**
  break <label>
  break *0x4005ea
  установить точку останова на начало функции *main*, на метку *label* или на адрес *0x4005ea*, соответственно
- **d n**
  удалить точку останова под номером n
- **start**
  запустить на выполнение бинарник
- **s**
  step
  сделать шаг (во время отладки)

- **si**
  si n
  выполнить текущую инструкцию, соответственно выполнить n инструкций (можно задавать любое число)
- **u *0x400546**
  продолжить выполнение до инструкции по адресу 0x400546
- **i r**
  i r <reg_name>
  вывести содержимое всех регистров или регистра reg_name
- **x/5a $rsp**
  x/5s *0x600342
  x/5w *0x600a0c
  вывести 5 указателей на вершине стека (в памяти по адресу, хранимому в **rsp**), вывести 5 нуль завершённых строк по адресу 0x600342, вывести 6 4-хбайтных числа по адресу 0x600a0c.

IDA

Interactive Disassembler, ***IDA Pro***, наиболее продвинутый дизассемблер из имеющихся на данный момент [10]. Позволяет дизассемблировать, отлаживать и модифицировать приложения для большого количества платформ в интерактивном режиме. Для выполнения примеров и заданий настоящей лабораторной работы более чем достаточно функционала свободно распространяемой версии ***IDA Free***.

Интерфейс ***IDA*** достаточно интуитивен, ответы на возникающие вопросы можно уточнить в справке самого приложения, на официальном сайте, либо в неофициальном (но де-факто стандартном) руководстве [11].

Процедура установки ***IDA Free*** на ОС Windows интуитивна и обычно не требует дополнительных действий (достаточно запустить инсталлятор и следовать инструкциям на экране), на ОС Linux установка может выглядеть примерно следующим образом:

```
x@vbox:~$ cd ~/Downloads
x@vbox:~/Downloads$ chmod u+x ida-free-pc_90_x64linux.run
x@vbox:~/Downloads$ ./ida-free-pc_90_x64linux.run
x@vbox:~/Downloads$
x@vbox:~/Downloads$ mkdir -p ~/.idapro
x@vbox:~/Downloads$ cp idafree_*.hexlic ~/.idapro/
```

[N.B.] Версия 9.0 актуальна на момент написания данного текста, проверьте текущую актуальную версию перед установкой.

На рабочем столе появится иконка «*IDA Free 9.0*», для запуска ***IDA*** с её помощью необходимо разрешить запуск из контекстного меню: правая клавиша мыши -> Allow Launching.

Если (скорее всего) ***IDA*** после установки не запустится, проверьте, каких плагинов QT не хватает:

```
x@vbox:~/Downloads$ QT_DEBUG_PLUGINS=1 ~/ida-free-pc-9.0/ida
```

Если в выводе вы увидите строку наподобие “libxcb-xinerama.so.0: cannot open shared object file: No such file or directory”, требуется доставить библиотеку libxcb-xinerama:

```
x@vbox:~/Downloads$ sudo apt install libxcb-xinerama0
```

1. Идентификация функций и локальных переменных

Функция – основная структурная единица процедурных, объектно-ориентированных и функциональных языков программирования, используемая для структурирования кода программы. Понятие функция весьма расплывчато, и понимаемое под этим термином зависит от языка программирования. Мы будем считать, что функция – это некоторая обособленная последовательность команд. Функция может принимать аргументы, а может и не принимать; функция может возвращать результат выполнения, а может и не возвращать. Характерная особенность функции – она может вызываться из различных мест программы, а по завершению работы функции управление передаётся обратно на место её вызова. Способы передачи аргументов и возвращения значения будут рассмотрены в следующем разделе, пока же исследуем типичную структуру функции и механизм возвращения управления.

В большинстве случаев, для вызова функции и возвращению из неё компиляторы используют команды **call** и **ret**, соответственно. Определить начало функции можно исходя из того, что, как было сказано выше, функция вызывается из других участков программы, следовательно, если какой-то адрес используется в качестве аргумента инструкции **call**, то, скорее всего, это начало функции [12]. Однако, существует немалое количество способов неявного вызова функций, называемых *косвенными*: в частности, так вызываются т.н. виртуальные функции, вызовы по указателю на функцию и т.п. Эти специальные (специфические) способы вызова мы затрагивать не будем. Когда выполнение программы доходит до команды **call**, исполняющая среда (процессор) понимает, что сначала необходимо передать управление на адрес, по которому располагается функция, а по завершению выполнения этой функции – вернуть управление на место вызова. Разумеется, управление возвращается не на ту же инструкцию **call** (что привело бы к зацикливанию программы), а на инструкцию, следующую за ней. Это осуществляется через стек: адрес инструкции, следующей за **call**, помещается на стек (вершина стека в зависимости от разрядности исполняемого файла уменьшается на 4 или 8 байт и на выделенное место копируется адрес – то же, что делает **push** *addr*), вызываемая функция отрабатывает и в конце выполнят команду **ret** или **retn**, которая извлекает адрес (как это делает команда **pop**) и передаёт управление на этот адрес.

Большинство компиляторов (по крайней мере без оптимизаций) помещают в начало функции т.н. *пролог* функции [13], назначение которого состоит в том, чтобы:

- сохранить значения всех регистров, используемых в коде функции
- выделить место на стеке для локальных переменных, это место называется *кадр стека.*

Локальные переменные функции располагаются в соответствующем ей кадре стека. Для адресации (доступа) конкретных переменных может быть использован регистр **ebp**/**rbp**, либо же непосредственно сам регистр **esp**/**rsp**. Характер использования адресованных переменных даёт подсказку о типе этой переменной (например, если 4 байта из кадра стека загружаются в регистр **eax**, то, скорее всего, тип этой переменной – *int*). В конце выполнения функция должна "вернуть всё как было": извлечь свой кадр из стека (ещё говорят "закрыть кадр"), возвращая вершину стека на прежнюю позицию (обычно это выполняется командой **add esp, *N***, либо же командой **leave**), а также восстановить значения регистров (с помощью команды **pop**). Совокупность команд, выполняющих данные операции, называется *эпилог* [13].

Посмотрим на описанные части функции в дизассемблере, для этого скомпилируем программу *1.c*:

```
double func()
{
        char str[] = "This is a local string";
        char* p = "This is NOT a local string, a pointer is local though";
        int l0, l1, l2, l3, l4, l5 = 5;
```

```
        short l6 = 1;
        return 4.0;
}

int main()
{
        int local1, local2 = 55;
        long local3 = 0;
        float local4 = 0.0, local5 = 3.14;
        double local6 = 2.7, ret = 0.;
        ret = func();

        return 0;
}
```

с помощью ***gcc***:

```
x@vbox:~/lab2$ cd examples
x@vbox:~/lab2/examples$ gcc -g -m32 -fno-PIC 1.c -o 1 -no-pie
```

Исследуем полученный исполняемый бинарный файл под x86. Ниже приведена дизассемблированная функция *func* (откройте скомпилированный файл *1* в IDA и найдите функцию *func*).

```
func proc near

str= byte ptr -37h
p= dword ptr -20h
l0= dword ptr -1Ch
l1= dword ptr -18h
l2= dword ptr -14h
l3= dword ptr -10h
l4= dword ptr -0Ch
l5= dword ptr -8
l6= word ptr -2

        push    ebp                         ; начало пролога функции
        mov     ebp, esp
        sub     esp, 40h                    ; выделение памяти для
                                            ; локальных переменных
                                            ; далее инициализация переменных
        mov     eax, ds:dword_8048546
        mov     dword ptr [ebp+str], eax    ; копирование символов строки на стек
        mov     eax, ds:dword_804854A
        mov     dword ptr [ebp+str+4], eax
        mov     eax, ds:dword_804854E
        mov     dword ptr [ebp+str+8], eax
        mov     eax, ds:dword_8048552
        mov     dword ptr [ebp+str+0Ch], eax
        mov     eax, ds:dword_8048556
        mov     dword ptr [ebp+str+10h], eax
        movzx   eax, ds:word_804855A
        mov     word ptr [ebp+str+14h], ax
        movzx   eax, ds:byte_804855C
        mov     [ebp+str+16h], al
        mov     [ebp+p], offset aThisIsNotALoca    ; эта строка хранится в сегменте
                                                   ; данных, но указатель char* p
                                                   ; есть локальная переменная
                                                   ; поэтому хранится на стеке
        mov     [ebp+l5], 5                        ; int l5 = 5
        mov     [ebp+l6], 1                        ; short l6 = 1
        fld     ds:dbl_8048560                     ; возвращаемое значение, через
```

```
                                        ; стек для чисел с плавающей
                                        ; точкой, т.к. регистра xmm0
                                        ;в архитектуре x86 нет
        leave
        retn                            ; извлечь адрес возврата и
                                        ; перейти на место вызова функции
func endp
```

Первые три инструкции – пролог функции, в нём выполняется подготовка к адресации локальных переменных с помощью регистра **ebp** (сохраняется его значение **push ebp** и копируется текущая вершина стека **mov ebp, esp**), затем выделяется место для локальных переменных командой **sub esp, 40h**. Именованные строки в начале функции – это подсказки от ***IDA***, которая создаёт их в процессе автоматического анализа функции (определяя количество и расположение переменных). Эти подсказки содержат название переменной (если оно известно), размер переменной в байтах и смещение переменной от конца области локальных переменных, адрес которого хранится в **ebp**. Например, подсказка l5= dword ptr - 8 означает, что локальная переменная под названием l5 занимает 4 байта и смещена на 8 байт наверх от **ebp**, то есть её адрес ebp-8. Для повышения читаемости числовое смещение меняется на метку. Адресация локальных переменных в данной функции осуществляется с помощью регистра **ebp** следующим образом. Регистр **ebp** указывает на самый низ участка памяти, отведённого для локальных переменных. Когда нам необходимо получить или изменить значение переменной, мы отнимаем нужное смещение от регистра **ebp** и получаем адрес этой переменной. Например, переменная l5 смещена на 8 байт относительно конца области локальных переменных (на который указывает регистр **ebp**), поэтому для инициализации l5 мы используем следующую команду: **mov [ebp+l5], 5**. Обращение к остальным локальным переменным осуществляется так же, только меняется значение смещения и, возможно, размер операндов. На рис. 1 показано расположение локальных переменных на стеке и их значения после выполнения пролога и кода инициализации переменных.

Выражение в IDA	Реальное выражение	Стек	Комментарии
ebp+str	ebp - 0x37	"This is a local string\0"	char str[] = "This is a local string"
ebp+p	ebp - 0x20	0x8048510	char* p = "..."
ebp+l0	ebp - 0x1C	?	int l0
ebp+l1	ebp - 0x18	?	int l1
ebp+l2	ebp - 0x14	?	int l2
ebp+l3	ebp - 0x10	?	int l3
ebp+l4	ebp - 0xC	?	int l4
ebp+l5	ebp - 0x8	0x5	int l5 = 5
ebp+l6	ebp - 0x2	0x1	short l6 = 1
	ebp	0x8048425	ebp указывает на это место - на низ области памяти, отведённой на локальные переменные; здесь хранится адрес возврата

Рис. 1. Стек после пролога функции и инициализации переменных

2. Соглашения о вызове функции и возвращаемом значении

Идентификация аргументов функций

Существуют три способа передачи аргументов функции [12, 14]: через *стек,* через *регистры и комбинированный* (через стек и через регистры). Аргументы могут передаваться *по значению* и *по ссылке*. В первом случае передаётся копия соответствующей переменной, а во втором – указатель на переменную (т.е. адрес этой переменной в памяти).

Если для передачи аргументов используется стек (в ассемблерном листинге программы мы увидим команды **push** и **pop**), то по завершении работы вызываемой функции стек необходимо выровнять (вернуть регистру **rsp** значение до вызова функции). Это может быть сделано как в вызываемой функции, так и в вызывающей.

Имеется несколько стандартизованных способов передачи аргументов в функцию. Они называются *соглашения о вызове* (*calling conventions*):

1. *C-соглашение* (__cdecl): аргументы помещаются в стек справа налево в порядке их объявления (последний аргумент помещается на стек первым), очистка стека возлагается на вызывающую функцию.
2. *Pascal-соглашение* (Pascal, WINAPI): аргументы помещаются в стек слева направо в порядке их объявления (первый аргумент помещается на стек первым), стек очистка очищает вызывающая функция.
3. *Стандартное соглашение* (__stdcall): гибрид __cdecl и Pascal, аргументы помещаются на стек справа налево, но стек очищает сама вызываемая функция.
4. *Соглашение быстрого вызова*: аргументы передаются через регистры.
5. *AMD64 ABI*. Предыдущие соглашения используются только в архитектуре *x86*. В архитектуре *x86-64* используется следующее правило: аргументы передаются через регистры в установленном порядке (он зависит от операционной системы). Если количество аргументов превышает некоторое установленное значение, то остальные аргументы передаются через стек. Кто в этом случае очищает стек также зависит от ОС и компилятора.

Посмотрим на эти соглашения в дизассемблере. Для начала скомпилируем следующую программу под архитектуру *x86* с помощью компилятора ***gcc***.

[N.B.] Для удобства можно воспользоваться сайтом Compiler Explorer [15], позволяющим в одном окне браузера видеть исходный код на ЯВУ и соответствующий ассемблерный листинг, скомпилированный с заданными ключами (параметрами) компиляции. Однако рекомендуется выполнить описанные ниже шаги с помощью ***gcc*** и ***gdb***/***IDA***, дабы привыкнуть к интерфейсу этих программ.

```
int __attribute__ ((cdecl)) cdecl_call(int a1, int a2, int* a3, char* a4, float a5, double a6, double* a7, unsigned char a8)
{
	return 4;
}

int __attribute__((stdcall)) stdcall_call(int a1, int a2, int* a3, char* a4, float a5, double a6, double* a7, unsigned char a8)
{
	return 4;
}
```

```
int __attribute__((fastcall)) fastcall_call(int a1, int a2, int* a3, char* a4, float a5, double a6, double* a7, unsigned char a8)
{
	return 4;
}


int main()
{
	int lvar3 = 55, ret1 = 0, ret2 = 0, ret3 = 0;
	double lvar7 = 5.435;
	ret1 = cdecl_call(55, 66, &lvar3, "cdecl_call", 4867, 4861574.24, &lvar7, 0);
	ret2 = stdcall_call(56, 67, &lvar3, "stdcall_call", 4868, 4861575.24, &lvar7, 1);
	ret3 = fastcall_call(57, 68, &lvar3, "fastcall_call", 4869, 4861576.24, &lvar7, 0);

	return 0;
}
```

Для компиляции программы под 32-битную архитектуру, необходимо дополнительно указать ключ -m32 для ***gcc***:

```
x@vbox:~/lab2/examples$ gcc -o 2 -g -m32 -fno-PIC -mrtd 2.c -no-pie
x@vbox:~/lab2/examples$
```

Откроем полученный исполняемый файл *2* в ***IDA***. Так как мы скомпилировали файл с отладочными символами (с ключом -g), мы можем видеть названия аргументов, помещаемых на стек при вызове функции. Перейдём в функцию *main*. Участок кода, в котором происходит вызов функции cdecl_call, принимающей аргументы в соответствии с соглашением cdecl, выглядит следующим образом:

```
sub       esp, 4Ch
...
dword ptr [esp+20h], 0                    ; a8
lea       eax, [ebp+lvar7]
mov       [esp+1Ch], eax                  ; a7
fld       ds:dbl_8048610
fstp      qword ptr [esp+14h]             ; a6
mov       eax, 45981800h
mov       [esp+10h], eax                  ; a5
mov       dword ptr [esp+0Ch], offset a4
lea       eax, [ebp+lvar3]
mov       [esp+8], eax                    ; a3
mov       dword ptr [esp+4], 42h          ; a2
mov       dword ptr [esp], 37h            ; a1
call      cdecl_call
mov       [ebp+ret1], eax                 ; возвращаемое значение
```

В комментариях справа указано, какие параметры помещаются на стек в соответствующих командах. Как видно, компилятор сначала зарезервировал на стеке место для локальных переменных и для аргументов (командой **sub esp, 4Ch**). Затем на стек копируются аргументы функции cdecl_call справа налево в порядке их объявления. На момент выполнения **call** стек выглядит примерно так, как на рис. 2.

Напомним, что команда **push** уменьшает значение регистра **esp**/**rsp** на 4 или 8 байт (для 32-битных и 64-битных исполняемых файлов, соответственно), то есть сдвигает вершину стека, и затем копирует данные на эти 4 или 8 байт. Таким образом, последовательное выполнение команд **push** с аргументами функции, выполняемое справа налево в порядке их объявления, приведёт к тому, что значения аргументов будут копироваться на стек друг под другом, и к моменту выполнения инструкции **call** стек процесса будет

Рис. 2. Стек на момент выполнения инструкции call

выглядеть точно также, как на рис. 2. В вызываемой функции стек не очищается (используется команда **retn**).

```
cdecl_call proc near
        ...
        leave
        retn
```

Функция stdcall_call принимает аргументы в соответствии с соглашением stdcall, а значит аргументы записываются на стек в таком же порядке, что и для функции cdecl, в чём можно легко убедиться, посмотрев на участок кода, предшествующий её вызову:

```
        mov       dword ptr [esp+20h], 1            ; a8
        lea       eax, [ebp+lvar7]
        mov       [esp+1Ch], eax                    ; a7
        fld       ds:dbl_8048618
        fstp      qword ptr [esp+14h]               ; a6
        mov       eax, 45982000h
        mov       [esp+10h], eax                    ; a5
        mov       dword ptr [esp+0Ch], offset aStdcall_call ; a4
        lea       eax, [ebp+lvar3]
        mov       [esp+8], eax                      ; a3
        mov       dword ptr [esp+4], 43h            ; a2
        mov       dword ptr [esp], 38h              ; a1
        call      stdcall_call
        sub       esp, 24h
        mov       [ebp+ret2], eax
```

Однако, в отличии от соглашения cdecl, в соглашении stdcall вызываемая функция должна очистить стек самостоятельно, то есть "вытолкнуть" все аргументы из стека (или же просто увеличить значение регистра **esp** на суммарный размер всех аргументов). Команда **retn** *N*, совмещает извлечение адреса возврата и выталкивание *N* байт из стека, таким образом:

```
stdcall_call proc near
        ...
        leave
        retn 24h
```

функция stdcall_call очищает стек по завершении своей работы, в полном соответствии с соглашением.

Соглашение fastcall очень сильно зависит от операционной системы, для которой компилируется исполняемый файл, используемого компилятора и самой функции, поэтому действительный способ передачи аргументов всегда разный.

Теперь рассмотрим передачу аргументов в 64-битных исполняемых файлах. Порядок передачи аргументов по соглашению *AMD64 ABI* в ОС Linux следующий:

целочисленные аргументы и указатели (аргументы, передаваемые по ссылке)

1. **rdi**
2. **rsi**
3. **rdx**
4. **rcx**
5. **r8**
6. **r9**
7. 7-й и все последующие аргументы передаются через стек.

аргументы – числа с плавающей точкой одинарной или двойной точности

1. нецелочисленные аргументы с 1-го по 16-й передаются через регистры **xmm0**-**xmm15**
2. остальные аргументы передаются через стек.

Скомпилируем следующую программу (*3.c*)

```
double func(int a1, int a2, int* a3, char* a4, float a5, double a6, double* a7, unsigned char a8, int a9)
{
        return 4.0;
}

int main()
{
        int lvar3 = 55;
        double lvar7 = 5.435, ret = 0.;
        ret = func(55, 66, &lvar3, "func", 4867, 4861574.24, &lvar7, 0, 100);

        return 0;
}
```

с отладочными символами:

```
x@vbox:~/lab2/examples$ gcc -o 3 -g -fno-PIC 3.c -no-pie
x@vbox:~/lab2/examples$
```

Убедимся в описанном выше правиле:

```
        lea       rax, [rbp+lvar7]
        mov       rdx, 41528BA18F5C28F6h
        lea       rsi, [rbp+lvar3]
        mov       dword ptr [rsp], 64h              ; a9
        mov       r9d, 0                            ; a8
```

```
mov     r8, rax                         ; a7
mov     [rbp+var_28], rdx
movsd   xmm1, [rbp+var_28]              ; a6
movss   xmm0, cs:dword_400670           ; a5
mov     ecx, offset a4
mov     rdx, rsi                        ; a3
mov     esi, 42h                        ; a2
mov     edi, 37h                        ; a1
call    func
movsd   [rbp+var_28], xmm0
```

Обратите внимание на то, как передаются аргументы *a1* и *a2* – так как это переменные типа *int* (4 байта), то из значения передаются не через **rdi** и **rsi**, а через **edi** и **esi**. Указатель *a3* передаётся через **rdx**. Так как целочисленных аргументов у функции 7, то аргумент *a9* (7-й целочисленный) передаётся через стек: mov dword ptr [rsp], 64h. Нецелочисленные аргументы *a5* и *a6* передаются через регистры **xmm0** и **xmm1**.

Передача возвращаемого значения

Обычно возвращаемое значение передаётся следующим образом:

- если возвращаемое значения – целочисленный тип (*unsigned char*, *char*, *short*, ..., *int*, ..., *unsigned long*, *size_t*), тогда значение передаётся через регистр **eax**/**rax**
- если возвращаемое значение – указатель, то он также передаётся через **rax**
- если возвращаемое значение – нецелочисленный тип (*float*, *double*), значение передаётся через регистр **xmm0**.

3. Идентификация основных конструкций языков C и C++

Операторы ветвлений и цикл for

В настоящем разделе мы кратко рассмотрим типичные способы компиляции некоторых выражений языка C (и C++). Конкретный вид получаемого ассемблерного листинга сильно зависит от компилятора и архитектуры процессора [12]. Скомпилируем следующую функцию.

```
#include <stdio.h>

int main()
{
        int i = 0;
        int v1 = 5, v2 = 5, v3_1 = 1, v3_2 = 2,  v4_1 = 2, v4_2 = 1;
        int v5 = 1;
        //
        // if clauses
        //
        // 1st if
        if (v1 == 5)
                printf("%s\n", "First if expression (v1 == 5) resolved to true");

        // 2nd if
        if (v2 == 6)
                printf("%s\n", "Second if expression (v2 == 6) resolved to true");
        else
                printf("%s\n", "Second if expression (v2 == 6) resolved to false");

        // 3rd if
        if (v3_1 == 1 && v3_2 == 2)
```

```
                printf("%s\n", "Third if expression (v3_1 == 1 && v3_2 == 2) resolved
to true");
        else
                printf("%s\n", "Third if expression (v3_1 == 1 && v3_2 == 2) resolved
to false");

        // 4th if
        if (v4_1 == 1)
                printf("%s\n", "Fourth if expression (v4_1 == 1) resolved to true");
        else if (v4_2 == 2)
                printf("%s\n", "Fourth if expression (v4_2 == 2) resolved to true");
        else
                printf("%s\n", "Fourth if expressions resolved to false");


        //
        // switch clause
        //
        switch (v5)
        {
                case 0:
                  printf("%s\n", "switch-case expression for v5 == 0");
                  break;

                case 1:
                  printf("%s\n", "switch-case expression for v5 == 1");
                  break;

                case 2:
                  printf("%s\n", "switch-case expression for v5 == 2");
                  break;
        }


        //
        // for clause
        //
        for (i = 0; i < 100; i++)
        {
                printf("i = %d\n", i);
        }


        return 0;
}
```

Скомпилируем следующим образом:

```
x@vbox:~/lab2/examples$ gcc -o 4 -g -m32 -fno-PIC 4.c -no-pie
x@vbox:~/lab2/examples$
```

Дизассемблируя с помощью ***IDA***, мы можем видеть листинг в виде графа потока управления.

Рассмотрим первый *if* (рис. 3). Переменная *v1* сравнивается с 5 командой **cmp**. Если *v1* и 5 не равны, то условие – ложь, управление передаётся на метку loc_8048417 командой **jmp** (переход по зелёной стрелке). Иначе условие - истина, поэтому переход не выполняется и вызывается функция *printf*.

Второй *if* (рис. 4) организован примерно так же, как и первый. Блоки, соответствующие истинной и ложной ветке кода, располагаются вслед за командой **cmp**, друг под другом. Переменная *v2* сравнивается с 6 командой **cmp**. Если *v2* и 6 не равны, то условие – ложь, управление передаётся на метку loc_8048433

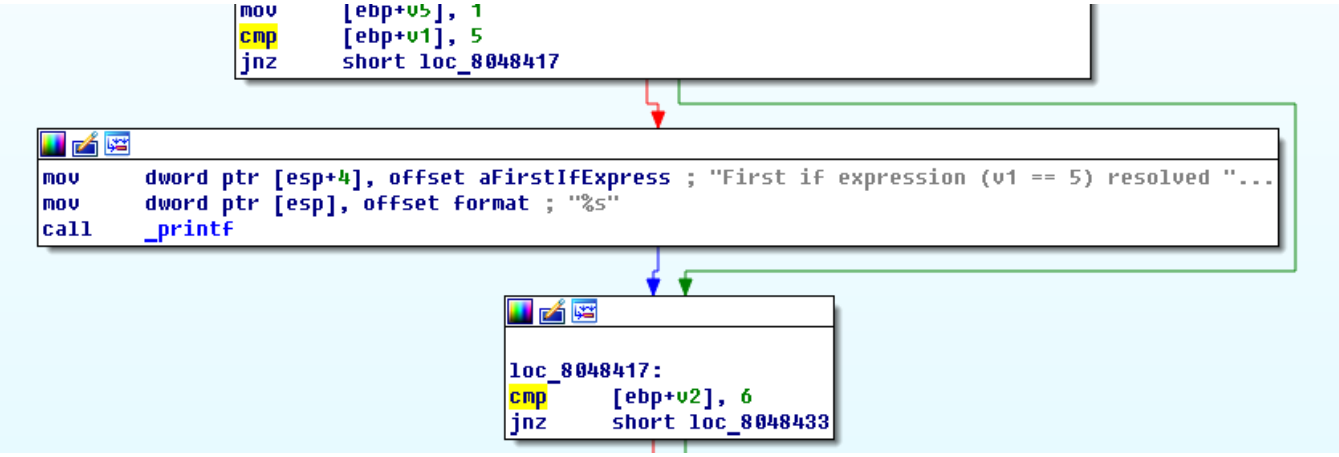

Рис. 3. Граф потока управления первого *if*

командой **jmp** (переход по зелёной стрелке). Если же *v2* == 5, то переход не выполняется и вызывается функция *printf*. Однако, чтобы не выполнять ложную ветку *if*, необходимо эту ветку "перепрыгнуть" – это делается командой **jmp short loc_808447**.

В третьем *if'e* проверяется два условия. На рис. 5 показан граф этого *if*. Если *v3_1* != 1 или v3_2 != 2, то выполняется переход на метку loc_804869 (командой **jmp short loc_808469**). Этой метке соответствует

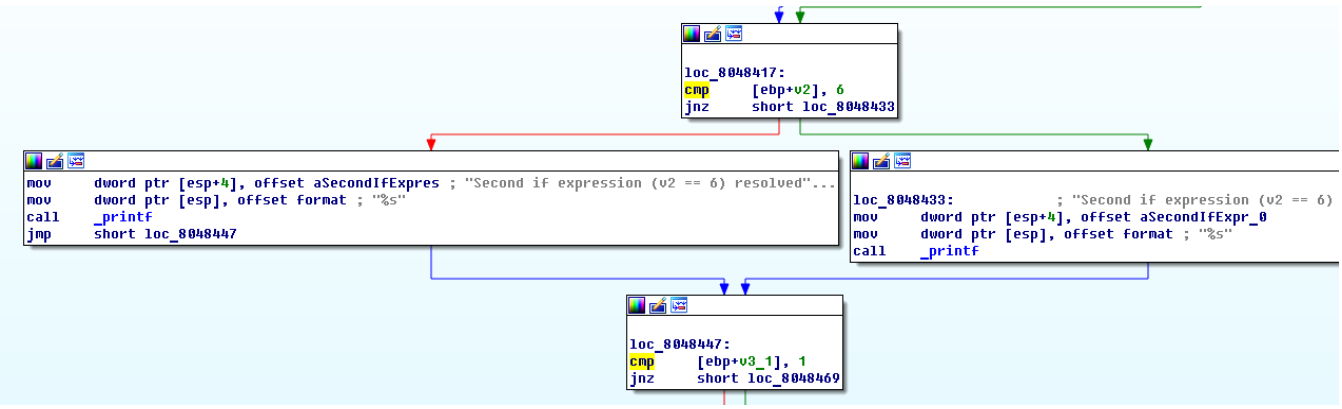

Рис. 4. Граф потока управления второго *if*

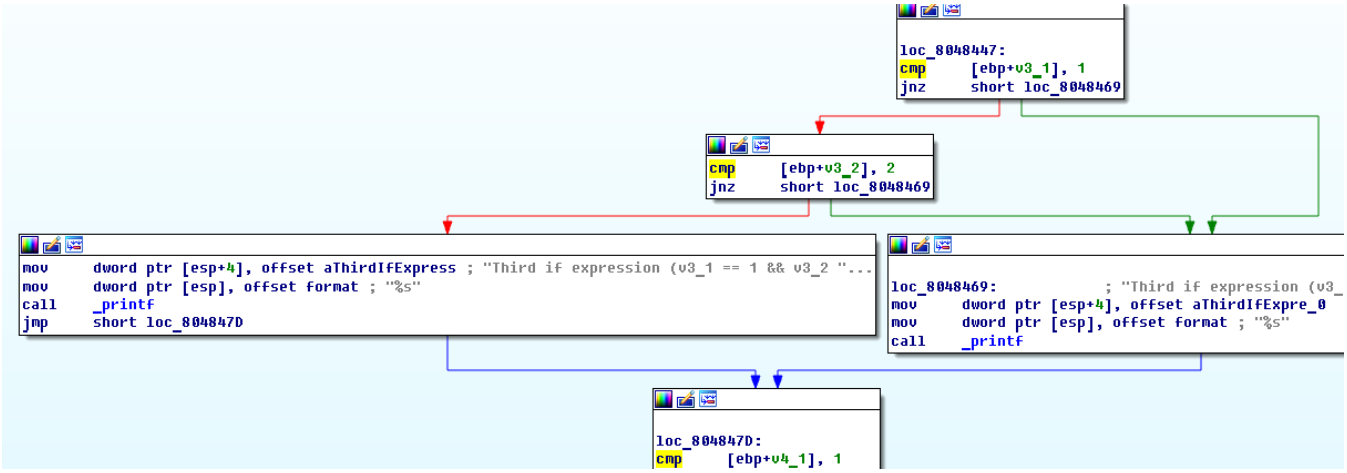

Рис. 5. Граф потока управления третьего *if*

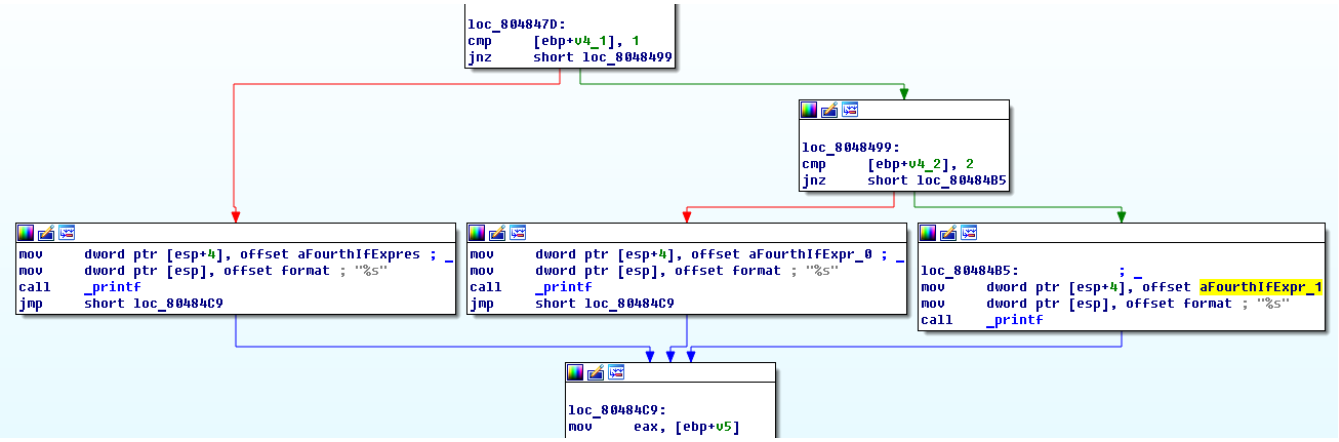

Рис. 6. Граф потока управления четвёртого *if*

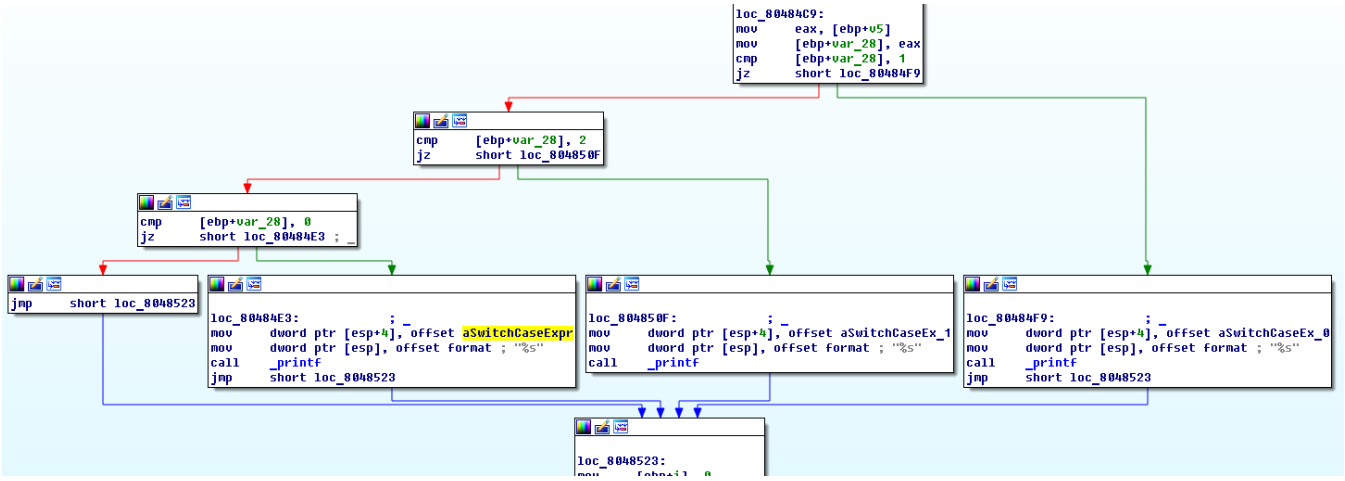

Рис. 7. Граф потока управления *switch-case*

ложная ветвь оператора *if*, т.к. согласно законам де Моргана A and B == not A OR not B. Если перехода не произошло, выполняется истинная ветвь оператора *if*. Оба блока располагаются точно также, как и в предыдущем случае.

В четвёртом *if* (рис. 6) также проверяются два условия. Если *v4_1* == 1, то переходим по красной стрелке: выполняется *printf*, далее переход на конец оператора (**jmp short loc_80484C9**). Если же первое условие не выполняется, проверяем второе командой **cmp [ebp+v4_2], 2**, при этом граф выглядит также, как и в случае второго *if*. Все три блока кода располагаются друг под другом.

Оператор *switch-break* (рис. 7) обычно реализуется как последовательность операторов *if*, поэтому ассемблерный листинг и граф потока управления практически идентичен графу предыдущего, четвёртого *if*.

В тестовой программе оператор *for* скомпилировался в очень простую последовательность команд (рис. 8). Сначала переменная-счётчик i инициализируется (обнуляется), далее проверяется условие i < 100, если оно истинно, то переходим на метку loc_804852C и вызываем *printf*. Команда **cmp [ebp+i], 63h** располагается сразу под этим блоком. Когда i < 100 перестаёт быть истиной, не переходим на метку, таким образом мы выходим из цикла.

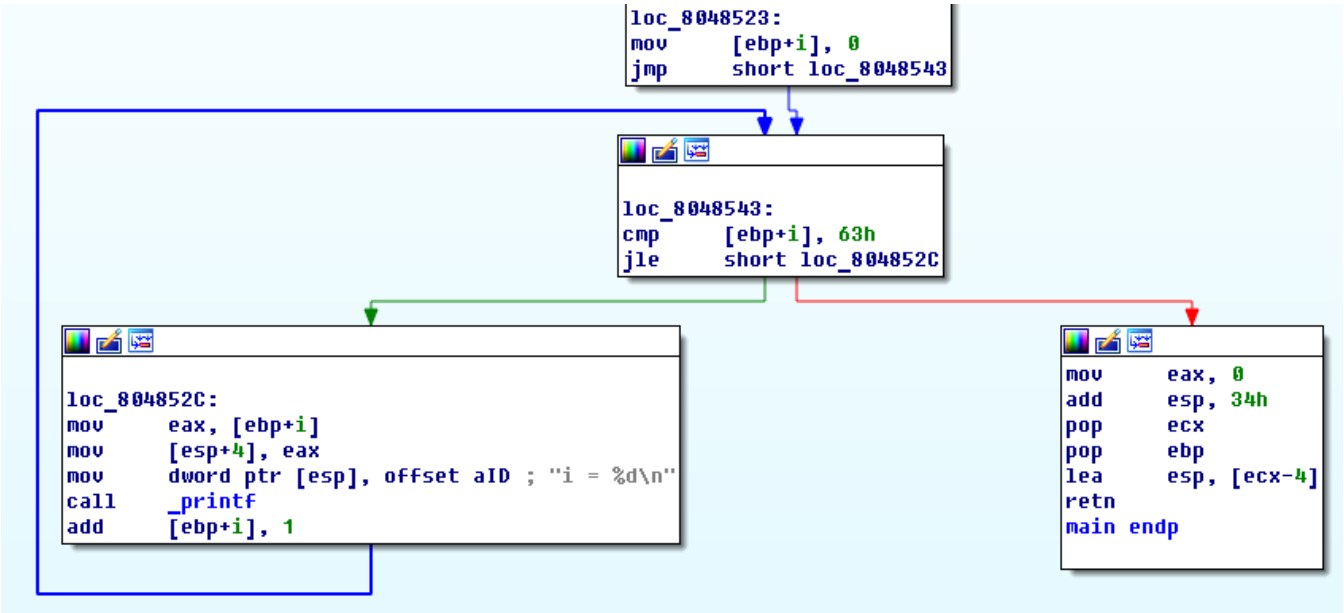

Рис. 8. Граф потока управления *for*

Как можно убедиться из приведённых выше примеров, типичные операторы C (C++) транслируются компилятором в достаточно чёткие и понятные последовательности команд.

Структуры и объекты классов

С точки зрения организации данных в памяти, объекты классов C++ ничем не отличаются от структур языка C. Рассмотрим на следующем примере, как располагаются поля структуры в памяти процесса [12]. Скомпилируем следующую программу:

```
#include <string.h>

struct MyStruct
{
        int a;
        long b;
        float c;
        double d;
        char e[10];
        int* f;
        MyStruct* pNext;
};

class MyClass
{
public:
        int a;
        long b;
        float c;
        double d;
        char e[10];
        int* f;
        MyClass* pNext;

public:
        MyClass()
        {
```

```
                this->a = 1;
                this->b = 2;
                this->c = 3.0;
                this->d = 4.0;
                memset(this->e, '5', sizeof(this->e));
                this->f = 0;
                this->pNext = 0;
        }
};

int main()
{
        MyStruct strct;
        MyClass  cls;
        strct.a++;
        strct.d += 5.0;
        cls.a++;
        cls.d += 5.0;
        return 0;
}
```

в этот раз с помощью *g++*, без отладочных символов:

```
x@vbox:~/lab2/examples$ g++ -o 5 -m32 -fno-PIC 5.c -no-pie
x@vbox:~/lab2/examples$
```

Дизассемблированная в ***IDA*** функция *main* выгладит следующим образом:

```
main proc near

var_54= dword ptr -54h
var_48= qword ptr -48h
var_2C= dword ptr -2Ch
var_20= qword ptr -20h
argc= dword ptr  0Ch
argv= dword ptr  10h
envp= dword ptr  14h

        lea       ecx, [esp+4]
        and       esp, 0FFFFFFF0h
        push      dword ptr [ecx-4]
        push      ebp
        mov       ebp, esp
        push      ecx
        sub       esp, 54h
        lea       eax, [ebp+var_54]
        mov       [esp], eax                    ; this
        call      MyClass::MyClass(void)
        mov       eax, [ebp+var_2C]
        add       eax, 1
        mov       [ebp+var_2C], eax
        fld       [ebp+var_20]
        fld       ds:dbl_8048598
        faddp     st(1), st
        fstp      [ebp+var_20]
        mov       eax, [ebp+var_54]
        add       eax, 1
        mov       [ebp+var_54], eax
        fld       [ebp+var_48]
        fld       ds:dbl_8048598
        faddp     st(1), st
```

```
        fstp      [ebp+var_48]
        mov       eax, 0
        add       esp, 54h
        pop       ecx
        pop       ebp
        lea       esp, [ecx-4]
        retn
main endp
```

Сначала командой **sub esp, 54h** резервируется место по две структуры: структуру типа *MyStruct* и объект класса *MyClass*. Поля структуры типа *MyStruct* не инициализируются, а для объекта класса *MyClass* вызывается конструктор. В качестве единственного аргумента конструктору передаётся указатель *this* – указатель на область памяти, в которой располагается сам объект.

Рассмотрим функцию MyClass::MyClass(void):

```
MyClass::MyClass(void) proc near

this= dword ptr  8

        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+this]             ; аргумент функции - конструктора
        mov       dword ptr [eax], 1
        mov       eax, [ebp+this]
        mov       dword ptr [eax+4], 2
        mov       edx, [ebp+this]
        mov       eax, 40400000h
        mov       [edx+8], eax
        mov       eax, [ebp+this]
        fld       ds:dbl_8048590
        fstp      qword ptr [eax+0Ch]
        mov       eax, [ebp+this]
        add       eax, 14h
        mov       dword ptr [eax], 35353535h
        mov       dword ptr [eax+4], 35353535h
        mov       word ptr [eax+8], 3535h
        mov       eax, [ebp+this]
        mov       dword ptr [eax+20h], 0
        mov       eax, [ebp+this]
        mov       dword ptr [eax+24h], 0
        pop       ebp
retn MyClass::MyClass(void) endp
```

Очевидно, что команды

```
        mov       eax, [ebp+this]
        mov       dword ptr [eax], 1
```

соответствуют оператору this->a = 1. Таким образом, первое поле типа *int* хранится в первых 4 байтах объекта класса, так как в **eax** помещается адрес начала объекта и но этому адресу записываются 4 байта – **dword ptr**.

Команды

```
        mov       eax, [ebp+this]
        mov       dword ptr [eax+4], 2
```

соответствуют оператору this->b = 2 (*int* и *long* одинакового размера на *x86*). Значит, второе поле хранится по смещению 4 байта от начала объекта и также занимает 4 байта.

Команды

```
mov       edx, [ebp+this]
mov       eax, 40400000h
mov       [edx+8], eax
```

записывают 4 байта по смещению 0x8 байт от начала объекта; 0x40400000 – это шестнадцатеричное представление числа 3.0. То есть эти команды соответствуют выражению this->c =34.0, а поле *c* класса *MyClass* хранится в 4 байтах и смещено на 8 байт от начала объекта.

Команды

```
mov       eax, [ebp+this]
add       eax, 14h
mov       dword ptr [eax], 35353535h
mov       dword ptr [eax+4], 35353535h
mov       word ptr [eax+8], 3535h
```

соответствуют выражению memset(this->e, '5', sizeof(this->e)). Поле e расположено по смещению 0x14 от вершины объекта.

Команды

```
mov       eax, [ebp+this]
mov       dword ptr [eax+20h], 0
mov       eax, [ebp+this]
mov       dword ptr [eax+24h], 0
```

соответствуют операторам this->f = 0 и this->pNext = 0. То есть, поля *f* и *pNext* располагаются по смещениям 0x20 и 0x24, причём это указатели, а потому занимают по 4 байта. Необходимо отметить, что смещение поля *e* равно 0x14, а смещение поля *f* – 0x20, то есть между ними 12 байт, хотя поле *f* есть массив из 10 байтов (*char e[10]*). Поля располагаются таким образом потому, что по умолчанию для ускорения работы с памятью поля выравниваются на границу слова – в данном случае, на *4* байта. Ниже в комментариях справа от полей написаны найденные смещения.

```
class MyClass
{
public:
        int a;                  /*0x0*/
        long b;                 /*0x4*/
        float c;                /*0x8*/
        double d;               /*0xC*/
        char e[10];             /*0x14*/
        int* f;                 /*0x20*/
        MyClass* pNext;         /*0x24*/
        ...
}
```

Как можно видеть из второй части ассемблерного листинга функции *main*, доступ к полям структуры *MyStruct* осуществляется аналогично.

Задания

1. Напишите простейшую программу на *C*, в которой определена функция с аргументами, возвращаемым значением и локальными переменными, и эта функция вызывается из функции *main* (можно взять первый пример). Запустите скомпилированную программу под отладкой (например, с помощью ***IDA*** или ***gdb***) и пронаблюдайте за стеком во время вызова функции и возвращения после её выполнения на место вызова. Найдите на стеке передаваемые аргументы, сохраняемый адрес возврата и все её локальные переменные.
2. Напишите программу на ассемблере, в ней определите функцию, принимающую аргументы не в соответствии с каким-либо соглашением (например, через другие регистры) и возвращающую несколько значений в регистрах. Используйте ***IDA*** и дизассемблируйте программу, затем декомпилируйте вашу функцию. Поняла ли ***IDA***, как именно вы передаёте аргументы в функцию и возвращаете значения? Распознала ли она какое-нибудь соглашение по вызову функций?
3. Скомпилируйте все примеры из раздела "**Операторы ветвлений и цикл for"**. Запустите их на отладку (с помощью ***IDA*** или ***gdb***) и проследите за состоянием регистров **eip** и **eflags** при выполнении операций ветвления в программе.
4. Напишите и скомпилируйте программу на *C* с циклами *while* и *do-while*. Дизассемблируйте эту программу с помощью ***IDA*** и определите, как организованы эти циклы на ассемблере.
5. Скомпилируйте пример из раздела "**Структуры и объекты классов"**. Запустите на отладку (с помощью ***IDA*** или ***gdb***) и проследите за значениями полей объекта класса *MyClass* в процессе работы конструктора. найдите поля структуры *MyStruct* на стеке функции *main*.
6. В соответствии с номером варианта, дизассемблируйте с помощью ***IDA*** скомпилированный исполняемый файл *%N%*. Восстановите вид используемых структур/классов и логику работы приложения. Используйте полученный код для написания собственной программы, выполняющей те же действия, что и исходная программа.

Литература

1. https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html
2. https://gcc.gnu.org/onlinedocs/
3. https://developers.redhat.com/blog/2021/04/30/the-gdb-developers-gnu-debugger-tutorial-part-1-getting-started-with-the-debugger
4. https://developers.redhat.com/articles/2022/01/10/gdb-developers-gnu-debugger-tutorial-part-2-all-about-debuginfo
5. https://developers.redhat.com/articles/2022/11/08/introduction-debug-events-learn-how-use-breakpoints
6. https://developers.redhat.com/articles/2021/10/05/printf-style-debugging-using-gdb-part-1
7. https://developers.redhat.com/articles/2021/10/13/printf-style-debugging-using-gdb-part-2
8. https://developers.redhat.com/articles/2021/12/09/printf-style-debugging-using-gdb-part-3
9. http://gnu.ist.utl.pt/software/gdb/documentation/
10. https://hex-rays.com/ida-pro/
11. Chris Eagle. The IDA Pro Book, 2nd Edition: The Unofficial Guide to the World's Most Popular Disassembler
12. Касперски К. Искусство дизассемблирования. БХВ-Петербург, 2009
13. https://en.wikipedia.org/wiki/Function_prologue_and_epilogue
14. https://en.wikipedia.org/wiki/Calling_convention
15. https://godbolt.org/
16. Эриксон Д. Хакинг: искусство эксплойта. 2-е издание. Символ, 2010