Лекция 15

ПРО СТЕК, СОГЛАШЕНИЕ О ВЫЗОВЕ И СИСТЕМНЫЕ ВЫЗОВЫ
А ТАКЖЕ НЕМНОГО ПРО ДИЗАССЕМБЛЕРЫ

rev. 4

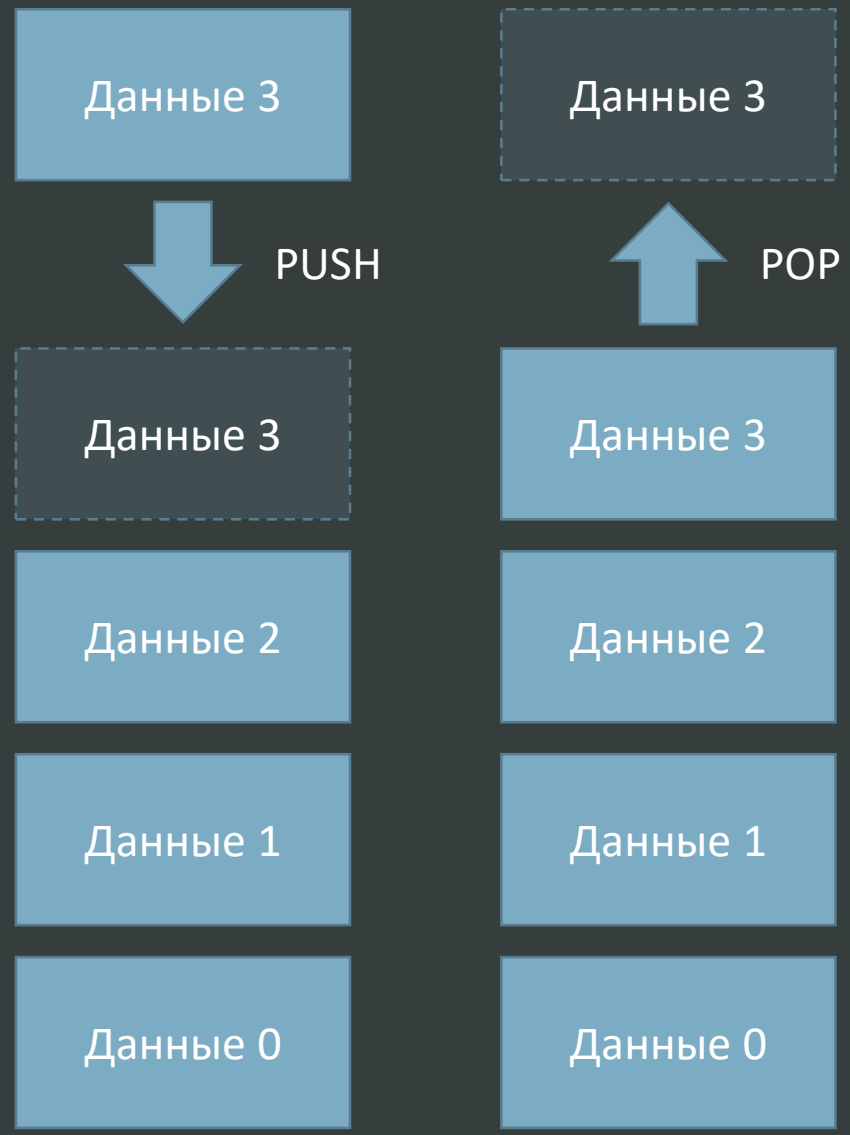
Так и куда же все-таки девается адрес возврата?

- Тот самый, который сохраняется функцией CALL и переход по которому осуществляется при помощи RET
- Он сохраняется в специальную область памяти, называемую стеком

Стек

Что такое стек

- Стек в общем смысле слова – структура данных, в которой соблюдается принцип LIFO (last in – first out, последним пришел, первым ушел)
- У типичного стека в мире информатики есть операции push и pop
  - При помощи этих операций можно класть что-то в стек и доставать что-то из него
  - В ассемблере x86 эти операции называются PUSH и POP соответственно

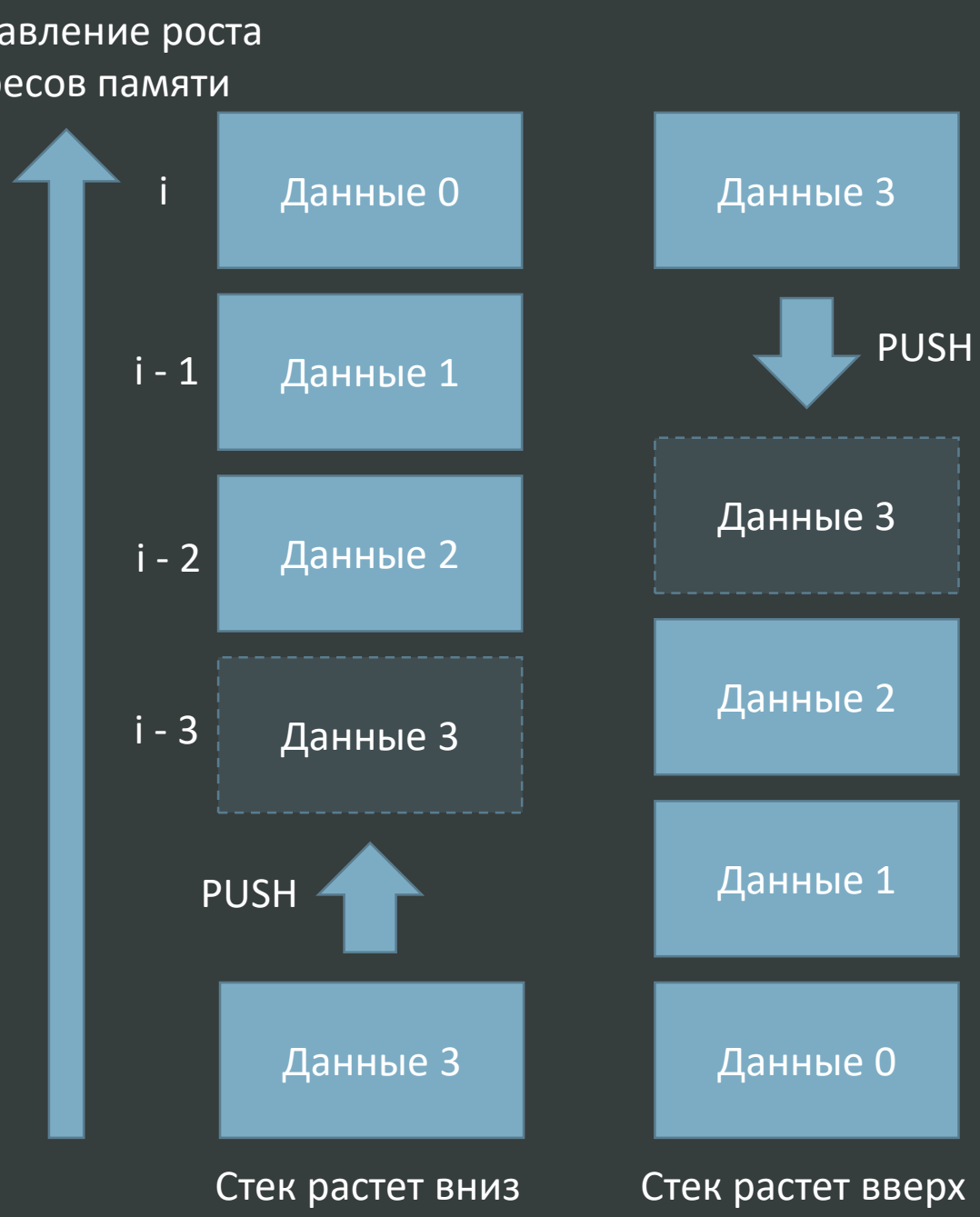


Что обычно хранят на стеке

- Адреса возврата функций
  - Очень полезное применение, мы с ним встретимся подробнее во второй половине семестра
- Локальные переменные, включая динамические массивы
- Аргументы функций
- В принципе, все что вы пожелаете, благодаря функции Си alloca()
  - Это как malloc(), только на стеке
- Все эти переменные живут до выхода из функции
  - Что логично, поскольку адрес возврата кладется в стек первым и к моменту, когда его достают, все остальное тоже нужно вытащить

Как выглядит стек в памяти

- Стек является непрерывной областью в памяти
- Начинается обычно далеко от области памяти кода и данных программы
- В Linux стек растет автоматически в пределах, установленных ulimit -s
  - Ядро само выделяет новую память под стек
- Стек в зависимости от архитектуры растет в разных направлениях:
  - Вниз – данные кладутся по меньшему адресу, чем предыдущие (x86, ARM, многие другие)
  - Вверх – наоборот (в теории – тоже ARM)



Что такое программа v2

- Обновим схему с прошлой лекции, используя знания о существовании стека
  - Данная схема актуальна для систем с ростом стека вниз

Стек на x86-64

- На архитектуре x86-64 текущий указатель стека хранится в регистре RSP
  - Также для нужд, связанных со стеком, используется RBP, но о нем позже
- Работу со стеком в основном реализуют следующие инструкции:
  - PUSH и POP
  - CALL и RET
- Стек, как было сказано ранее, растет вниз

Инструкции PUSH и POP

- Позволяют выполнить соответствующие операции со стеком
- Могут использовать в качестве аргументов регистр, адрес по регистру и константу (только PUSH)

Код ассемблера	Альтернатива на Си
PUSH 123	RSP -= 8; *((long long*)RSP) = 123;
PUSH RAX	RSP -= 8; *((long long*)RSP) = RAX;
PUSH [RAX]	RSP -= 8; *((long long*)RSP) = *((long long*)RAX);
POP RAX	RAX = *((long long*)RSP); RSP += 8;
POP [RAX]	*((long long*)RAX) = *((long long*)RSP); RSP += 8;

Инструкции PUSH и POP

- Оперируют аргументами размером с текущее машинное слово (или 64 бита, или 32 бита), но также позволяют использовать 16-битные аргументы
  - Для указания таких аргументов используется word ptr в случае использования адреса или 16-битные регистры в случае регистров, константы так указывать нельзя

Код ассемблера	Альтернатива на Си
PUSH AX	RSP -= 2; *((short*)RSP) = AX;
PUSH word ptr [RAX]	RSP -= 2; *((short*)RSP) = *((short*)RAX);
POP AX	AX = *((short*)RSP); RSP += 2;
POP word ptr [RAX]	*((short*)RAX) = *((short*)RSP); RSP += 2;

Инструкции CALL и RET

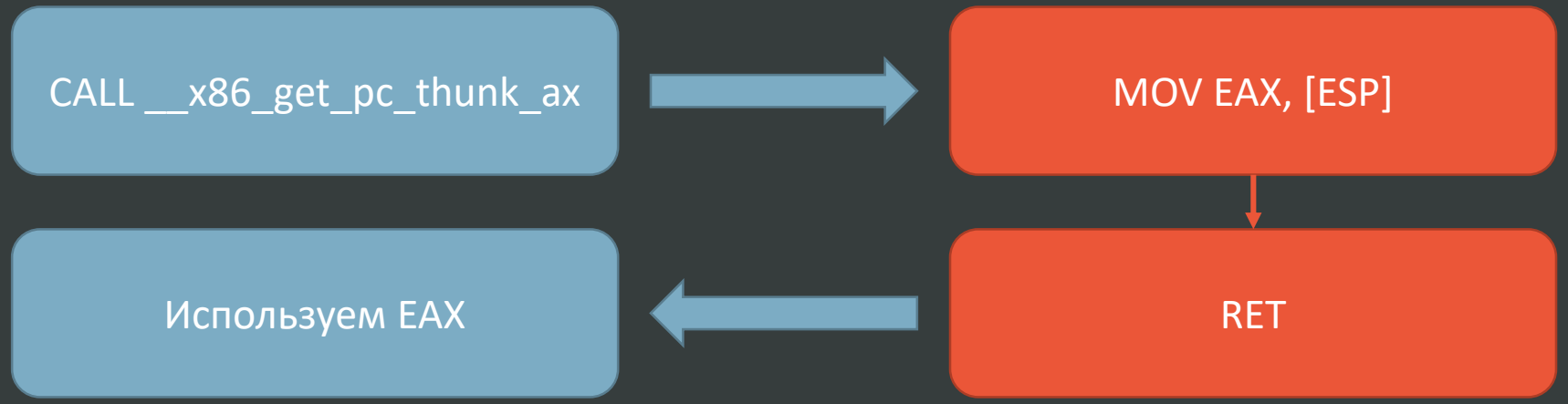
- CALL кладет на стек адрес инструкции, куда программа должна будет вернуться
  - То есть, кладет адрес следующей инструкции после CALL
- RET берет со стека адрес возврата и переходит на него

Код ассемблера	Альтернатива на Си
CALL RAX	RSP -= 8; *((long long*)RSP) = RIP + sizeof(call_instr); RIP = RAX;
RET	RIP = *((long long*)RSP); RSP += 8;

- RIP в этих примерах – регистр счетчика команд

Получаем значение RIP при помощи стека

- Костыль, ставший для 32-битного варианта x86 классикой
  - В GCC функции, реализующие такой костыль, называются вроде __x86_get_pc_thunk_ax
- Выглядит это следующим образом:



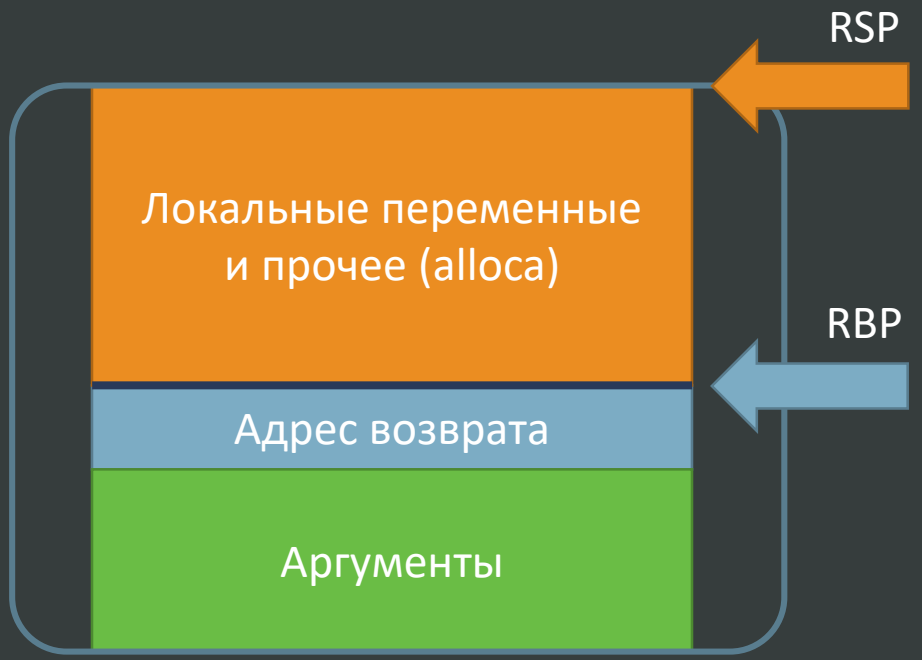
- В результате выполнения, правда, мы получим адрес следующей инструкции, а не текущей на момент CALL
- На x86-64 это можно по-нормальному сделать через LEA RAX, [RIP]

Понятие стекового кадра

- Стековый кадр (stack frame) создается при каждом вызове функции
- В стековый кадр обычно включают следующие области стека:
  - Переданные функции параметры
  - Адрес возврата
  - Локальные переменные и все, что функция создала на стеке в процессе работы
- Справа на схеме приведен пример вызова функции B из функции A

Регистр указателя кадра стека

- Более известен как frame pointer
- Указывает на место между локальными переменными и адресом возврата
  - Можно считать неизменным в процессе работы функции, в отличие от указателя стека (RSP)
  - Относительно него удобно получать доступ к адресам возврата и аргументам
- Строго говоря, этот регистр не является обязательным для использования в случае, если компилятор в состоянии считать все отступы относительно указателя стека
  - В GCC есть флаг -fomit-frame-pointer для неиспользования этого регистра



Минутка относительной адресации

- Как получить какие-то данные «относительно» регистра?
  - Как было указано на предыдущем слайде
- Оказывается, в x86 многие инструкции, принимающие адреса, могут осуществлять операции не только по адресу регистра, но и по адресу с некоторым смещением от него, выглядит синтаксис такой конструкции следующим образом:

```
[base + index*scale + displacement]
```

- В этой записи:
  - Base – регистр (может быть даже RIP, но тогда index = 0)
  - Index – регистр (кроме RSP)
  - Scale – 1, 2, 4 или 8
  - Displacement – константа размером до 32 бит

Минутка относительной адресации

- Благодаря вышеописанной записи, мы можем получать значения в памяти относительно регистра RBP для легкого получения аргументов функции
- Существует инструкция LEA, которая позволяет просто вычислить адрес получившегося значения, без разыменования указателя
  - В ней было бы мало смысла, не будь возможности писать такие сложные конструкции в качестве адреса

Код ассемблера	Альтернатива на Си
MOV RAX, [RBP + 8]	RAX = *((long long*)(RBP + 8));
MOV RAX, [RAX + 4 * RCX + 1337]	RAX = *((long long*)(RAX + 4 * RCX + 1337));
LEA RAX, [RAX + 4 * RCX + 1337]	RAX = RAX + 4 * RCX + 1337;

Инструкции для работы со стековыми кадрами

- Нетрудно видеть, что при входе в функцию (или непосредственно перед ее вызовом) значение указателей стека было бы хорошо сохранять и восстанавливать при выходе
  - В противном случае любой вызов функции испортил бы стек (RSP) и адреса аргументов текущей функции (RBP)
- Типичные способы это сделать, которые вы можете встретить во многих программах x86-64 выглядят следующим образом:

Создание стекового кадра с сохранением RBP:

```
push   rbp
mov    rbp, rsp
sub    rsp, 0x20
```

Уничтожение стекового кадра (восстановление RSP и RBP):

```
mov    rsp, rbp
pop    rbp
ret
```

Инструкции для работы со стековыми кадрами

- Чтобы каждый раз не писать одни и те же инструкции, были созданы следующие инструкции, объединяющие вышеописанные операции:
  - ENTER – инструкция для создания стекового кадра, вы ее вряд ли встретите
  - LEAVE – достаточно часто встречающаяся инструкция для удаления стекового кадра
- ENTER принимает два параметра, объем области для локальных переменных и «уровень вложенности», второй из них у нас будет всегда нулевым для простоты

Код ассемблера	Альтернатива на ассемблере

ENTER x, 0	PUSH rbp
	MOV rbp, rsp
	SUB rsp, x

LEAVE	MOV rsp, rbp POP rbp

Соглашение о вызове

Соглашение о вызове

- Ранее мы выяснили, что аргументы функции могут лежать и в регистрах и на стеке
- Возникает закономерный вопрос, когда где и чем это вообще регламентируется?
- Для описания всего процесса вызова функций существует понятие «соглашение о вызове» (calling convention), которое включает следующее:
  - Способ передачи аргументов (в регистрах, на стеке, в каком порядке)
  - Способ возврата значения из функции (в регистре, на стеке и т.д.)
  - Какие регистры вызываемая функция обязана оставить неизменными к моменту выхода
  - Некоторые другие понятия, которые нам не пригодятся

Соглашение о вызове

- Наиболее популярными соглашениями о вызове в Linux являются:
  - cdecl (32-битный вариант x86)
  - System V AMD64 ABI (x86-64)
- Почитать про другие соглашения о вызове можно в интернете, эта информация широко доступна
  - https://www.agner.org/optimize/calling_conventions.pdf

cdecl (Linux)

- Передача аргументов:
  - Справа налево, на стеке (последний аргумент пушится первым)
  - Каждый аргумент размером менее 4 байт расширяется до 4 байт
- Возвращаемое значение:
  - В регистре EAX если число до 4 байт (или указатель) или в паре EDX:EAX если число 8 байт
  - В случае структур данных вызывающая функция выделяет память под структуру и передает на стеке скрытый параметр с указателем на нее, а вызываемая функция сдвигает потом стек обратно через особую форму инструкции RET (RET x)
- Сохранение регистров:
  - Вызываемая функция должна сохранять все кроме EAX, ECX и EDX
  - За очистку стека от аргументов отвечает вызывающая функция
- Выравнивание адреса стека на 16 байт (на момент вызова функций ESP % 16 == 0)

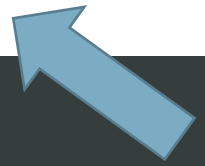
Пример вызова функции с cdecl

Код на Си:

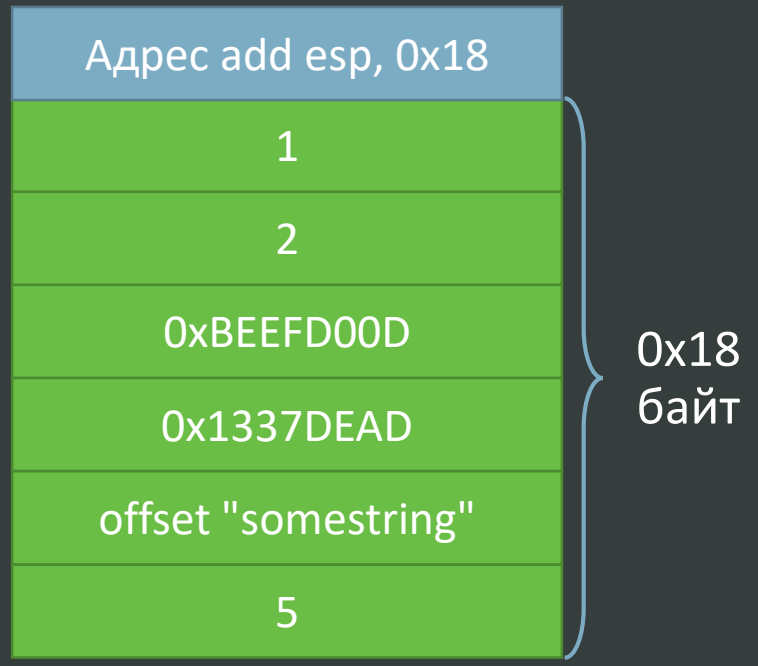
```
int somefunction(int a,
int b, long long c,
char* d, char e);

int main()
{
somefunction(1, 2,
0x1337DEADBEEFD00D,
"somestring", 5);
}
```

Код вызова на ассемблере:

```
push    5
mov     eax, offset somestring
push    eax
push    0x1337DEAD
push    0xBEEFD00D
push    2
push    1
call    somefunction
add     esp, 0x18
```



Очистка стека вызывающей функцией



System V AMD64 ABI

- Передача аргументов:
  - В регистрах RDI, RSI, RDX, RCX, R8, R9, кто не влез – далее на стеке справа налево
  - Каждый аргумент размером менее 8 байт расширяется до 8 байт
- Возвращаемое значение:
  - В регистре RAX если число до 8 байт (или указатель) или в паре RDX:RAX если число 16 байт
  - В случае структур данных вызывающая функция выделяет память под структуру и передает адрес на эту память в регистре RDI (все остальные аргументы сдвигаются на 1). Вызываемая функция возвращает это значение указателя в регистре RAX
- Сохранение регистров:
  - Вызываемая функция должна сохранять регистры RBX, RSP, RBP, R12, R13, R14 И R15
  - За очистку стека от аргументов отвечает вызывающая функция
- Выравнивание адреса стека на 16 байт (на момент вызова функций RSP % 16 == 0)

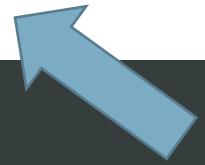
Пример вызова функции с System V AMD64 ABI

Код на Си:

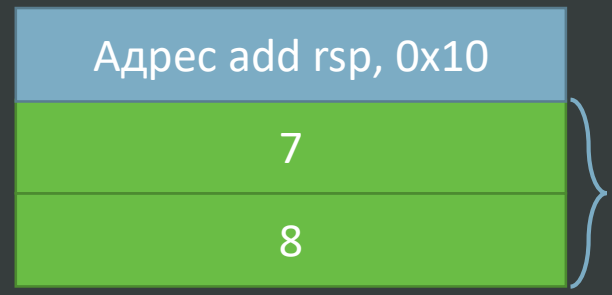
```
int somefunction(int a,
int b, long long c, char*
d, char e, char f, char g,
int h);

int main()
{
somefunction(1, 2,
0x1337DEADBEEFD00D,
"somestring", 5, 6, 7, 8);
}
```

Код вызова на ассемблере:

```
push    8
push    7
mov     r9d, 6
mov     r8d, 5
lea     rcx, offset somestring
mov     rdx, 0x1337DEADBEEFD00D
mov     esi, 2
mov     edi, 1
call    somefunction
add     rsp, 0x10
```



Очистка стека вызывающей функцией



RDI	RSI	RDX
1	2	0x1337DEADBEEFD00D

RCX	R8	R9
offset "somestring"	5	6

Переменное число аргументов

- Некоторые функции имеют переменное число аргументов
- К таким функциям относятся, например, printf, scanf
  - Там количество аргументов предполагается зависящим от форматной строки
- Вызовы подобных функций практически не отличаются от вызова обычных:
  - В случае x86 (32) ничто не мешает положить больше аргументов на стек, а потом их убрать
  - В случае x86-64, казалось бы, тоже можно спокойно размещать в регистрах и на стеке лишние аргументы, однако есть одно примечание
- В случае System V AMD64 ABI в регистр EAX должно быть записано количество аргументов, размещенных в регистрах SSE (которые хранят дробные числа)
  - То есть, если у вас нет дробных аргументов, туда нужно написать 0

Немного о дробных числах

Немного о дробных числах

- Я старательно избегал дробных чисел, но нам пришлось с ними встретиться на прошлом слайде
- Изначально аппаратная поддержка дробных чисел в x86 обеспечивалась при помощи x87
  - Это набор инструкций математических сопроцессоров Intel
  - Также включает 8 регистров ST0-ST7 для дробных чисел (каждый по 80 бит размером)
  - Инструкции в основном работают с регистрами как со стеком
- Современные системы используют для дробных чисел SSE
  - Набор SIMD инструкций, также позволяет работать с дробными числами
  - Включает 8 (32-битный x86) или 16 (x86-64) регистров XMM0-XMM7(15) по 128 бит
  - AVX расширяет эти регистры до 256 бит (YMM0-YMM15)
  - AVX-512 расширяет регистры до 512 бит и увеличивает их число до 32 (ZMM0-ZMM31)

Дробные числа и соглашение о вызове

- В случае cdecl дробные числа передаются на стеке
  - Однако возвращаемое дробное значение прилетит в ST0
- В случае System V AMD64 ABI все чуть сложнее
  - Обычные дробные аргументы (float / double) передаются в регистрах XMM0-XMM7 независимо от целочисленных
  - 80-битные числа long double передаются все так же через ST0 и т.д. (хотя по моим данным через стек)
  - Что не влезло в регистры кладется на стек
  - Возвращаемое значение в зависимости от типа может лежать как в XMM0, так и в ST0

Примеры

```
fstp    dword ptr [esp]
call    _sinf
add     esp, 0x10
fstp    dword ptr [ebp+var_38]
...
fstp    qword ptr [esp]
call    _sin
add     esp, 0x10
fstp    [ebp+var_20]
...
fstp    tbyte ptr [esp]
call    _sinl
add     esp, 0x10
fstp    [ebp+var_38]
```

cdecl

```
movss   xmm0, cs:x
call    _sinf
movd    eax, xmm0
...
movsd   xmm0, qword ptr [rbp+var_30]
call    _sin
movq    rax, xmm0
...
fstp    tbyte ptr [rsp]
call    _sinl
add     rsp, 0x10
fstp    tbyte ptr [rbp-0x30]
```

System V AMD64 ABI

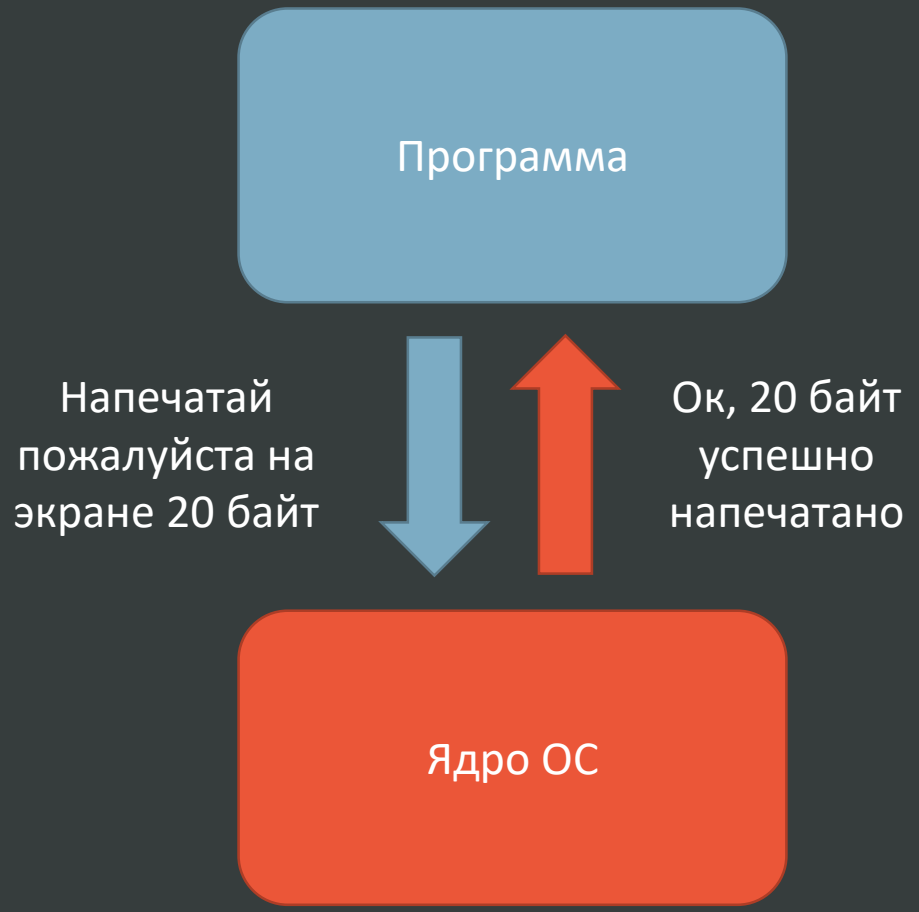
Системные вызовы

Как вывести текст на экран на ассемблере?

- Как вы ранее могли заметить, ранее не было упомянуто никаких инструкций для, например, вывода текста на экран
  - Это потому что у современных программ нет прав делать это напрямую, обращаясь к аппаратному обеспечению ПК, это сделано из соображений безопасности
- Одним из способов осуществления действий, связанных с системой (например, вывод текста или чтение файлов), является использование сторонних библиотек, например стандартной библиотеки языка Си (libc)
  - Пример такого вызова: CALL printf
- Однако библиотеки ведь тоже состоят из какого-то машинного кода?
  - А значит, в них есть инструкции, осуществляющие, в том числе, вывод текста на экран

Понятие системного вызова

- Системный вызов – обращение программы к ядру операционной системы
- Осуществляются вызовы при помощи прерываний процессора или специальных инструкций
- С точки зрения обратной разработки системный вызов можно рассматривать в первом приближении как просто вызов функции
- Подробнее о системных вызовах можно узнать на курсе по операционным системам



Системные вызовы Linux

- У каждого системного вызова Linux есть следующие параметры:
  - Номер системного вызова – параметр, позволяющий ядру понять, какой именно системный вызов вы у него попросили
  - Аргументы – практически то же самое, что и у обычных функций
- Со многими системными вызовами вы могли быть знакомы по функциям из библиотеки языка Си на Linux, многие из этих функций являются тонкими прослойками между системными вызовами и языком Си
  - В отличие от системных вызовов, эти функции используют для вывода ошибок errno
- С некоторыми такими функциями вы наверняка уже знакомы:
  - write(), read() – запись и чтение из файловых дескрипторов (в т.ч. терминала)
  - open(), close(), stat() – открытие и закрытие файлов, информация о файле
  - exit(), kill() – завершает свой или чужой процесс

Как осуществить системный вызов

- Для осуществления системного вызова на x86 существует несколько способов:
  - INT 0x80 – самый классический способ осуществления системных вызовов. Является основным способом на 32-битном Linux-x86, также может использоваться на 64-битном с оговоркой, что все переданные аргументы будут 32-битными. Поэтому там его использовать не стоит
  - SYSCALL – новый способ, доступный только на 64-битных системах. Является там основным способом осуществления системных вызовов
- Аргументы в обоих случаях (и x86, и x86-64) передаются в регистрах (что отличает системные вызовы от функций), да еще и не всегда в тех же

Платформа	№ Вызова	Аргумент 1	Аргумент 2	Аргумент 3	Аргумент 4	Аргумент 5	Аргумент 6	Результат
x86 (32 bit)	EAX	EBX	ECX	EDX	ESI	EDI		EAX
x86 (64 bit)	RAX	RDI	RSI	RDX	R10	R8	R9	RAX

- Красным отмечено отличие от System V AMD64 ABI, там четвертый аргумент лежит в RCX

Пример

- В качестве примера попробуем распечатать на экране строку "hello world"
- Для этого нам понадобится системный вызов write()
  - В случае x86-64 его номер будет равен 1
- Этот системный вызов принимает три параметра:
  - Файловый дескриптор fd
  - Указатель на буфер с текстом buf
  - Количество байт, которые следует вывести в файловый дескриптор count
- На Си этот пример выглядел бы следующим образом:

```
write(STDOUT_FILENO, "hello world\n", strlen("hello world\n"));
```

Пример

- Поскольку номер системного вызова SYS_write равен 1, поместим в регистр RAX единицу
- Первый параметр равен STDOUT_FILENO, эта константа равна одному, поэтому поместим в регистр RDI (первый аргумент) единицу
- Во второй параметр (регистр RSI) поместим указатель на нашу строку
- В третий параметр (регистр RDX) поместим ее длину, она равна 12
- Итоговый код приведен справа

```
.intel_syntax noprefix
.text
.global main
main:
    mov rax, 1
    mov rdi, 1
    mov rsi, offset hw
    mov rdx, 12
    syscall
    ret

hw:
    .asciz "hello world\n"
```

Системные вызовы Linux

- Естественно, учить все таблицы системных вызовов наизусть – плохая идея
- Удобно посмотреть таблицы можно:
  - На сайте https://filippo.io/linux-syscall-table/ (только x86-64, 32-битный x86 "Coming soon" уже несколько лет)
  - Как ни странно, в репозитории ChromeOS: https://chromium.googlesource.com/chromiumos/docs/+/HEAD/constants/syscalls.md
- Также некоторые дизассемблеры могут распознать системные вызовы самостоятельно

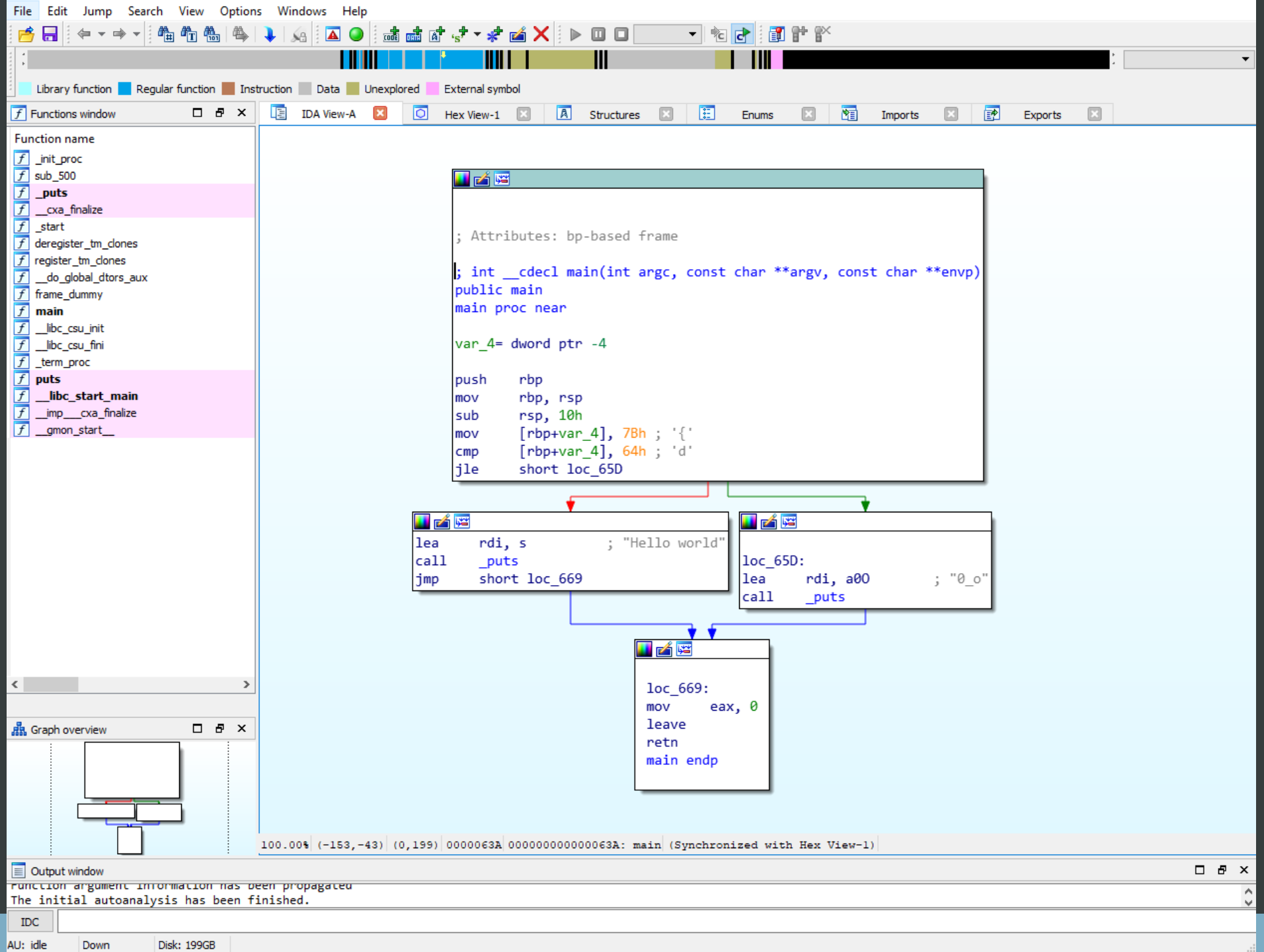
Дизассемблеры

Дизассемблеры

- Дизассемблер – утилита, преобразующая машинный код в программу на языке ассемблера
- Широко распространены в среде реверсеров следующие дизассемблеры:
  - IDA
  - Ghidra
  - Radare2

IDA

- IDA – Interactive DisAssembler
  - Интерактивный – потому что позволяет вам переименовывать функции, превращать данные в код и т.д.
- Самый классический из дизассемблеров, имеет очень широкую поддержку сообщества
- Очень платный ($4.7k за IDA Pro и x86-64 декомпилятор)
  - Купить его из-за политики автора / компании было целым приключением (т.к. авторы опасаются пиратства): https://habr.com/ru/post/124054/
  - Несмотря на параноидальные меры защиты, все равно периодически подвергается пиратству
- Также имеет встроенный отладчик


File Edit Jump Search View Options Windows Help
Library function Regular function Instruction Data Unexplored External symbol
Functions window
Function name
_init_proc
sub_500
_puts
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
main
__libc_csu_init
__libc_csu_fini
_term_proc
puts
__libc_start_main
__imp___cxa_finalize
__gmon_start__
Graph overview
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
; Attributes: bp-based frame
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near
var_4= dword ptr -4
push rbp
mov rbp, rsp
sub rsp, 10h
mov [rbp+var_4], 7Bh ; '{'
cmp [rbp+var_4], 64h ; 'd'
jle short loc_65D
lea rdi, s ; "Hello world"
call _puts
jmp short loc_669
loc_65D:
lea rdi, a0O ; "0_o"
call _puts
loc_669:
mov eax, 0
leave
retn
main endp
100.00% (-153,-43) (0,199) 0000063A 000000000000063A: main (Synchronized with Hex View-1)
Output window
The initial autoanalysis has been finished.
IDC
AU: idle
Down
Disk: 199GB

IDA

- Чтобы открыть исполняемый файл в IDA, достаточно просто перетащить его в окно и нажать ОК
- Чтобы файл закрыть, рекомендую нажать "DON'T SAVE the database", иначе IDA сохранит вам файл со всеми результатами переименований функций (или просто с результатом своего автоматического анализа), что зачастую не нужно

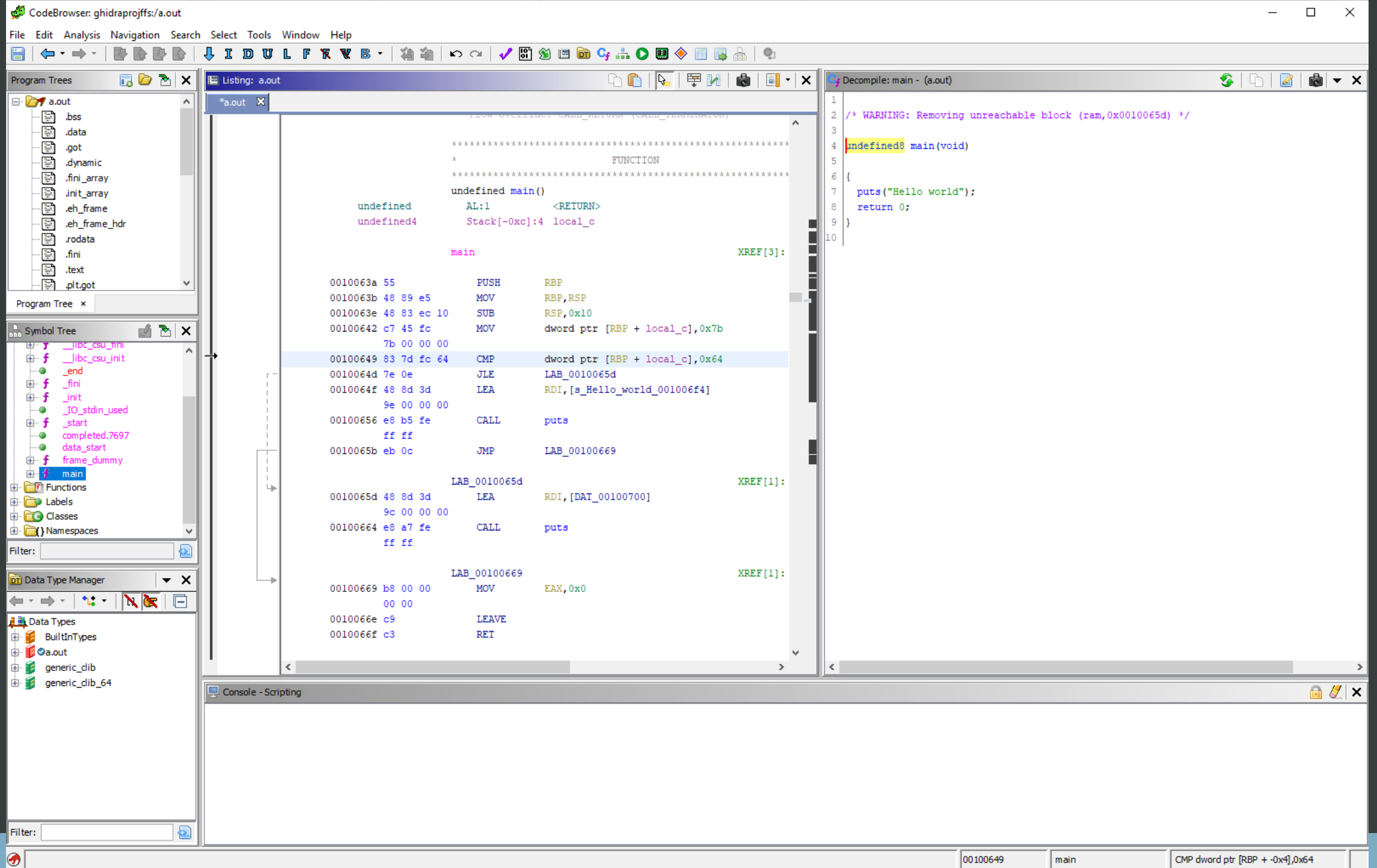
IDA

- Самые часто используемые (мной) операции в IDA:
  - Кнопка X (Cross References) на каком-либо объекте (функции, например) – показывает список использований этого объекта где-либо еще
    - Такой функционал часто встречается в средах разработки
  - Правая кнопка мыши -> Text view / Graph view – позволяет переключаться между режимами отображения в виде текста и графика
  - Кнопки D / C – позволяют преобразовать текущий фрагмент (выделенный или строку) в данные или код соответственно. На случай если IDA сама не смогла правильно определить
  - Кнопка N – переименовать текущий объект (опять же, например, функцию)
  - F5 – декомпилятор Hex-Rays (доступен только в платной версии, покупается отдельно)
  - Ну и еще в меню есть поиск, но там все довольно просто

Ghidra

"It sounds annoying but it is very quick and easy."

- Выпущенный в марте 2019 года инструмент для обратной разработки от АНБ
- Официальный сайт http://ghidra-sre.org/ при использовании российских IP адресов ~~кидает 403~~ притворяется мертвым ¯_(ツ)_/¯
  - Впрочем можно зайти через Tor, так что не очень понятно, на кого рассчитана эта блокировка
  - Собранные версии теперь размещаются на GitHub: https://github.com/NationalSecurityAgency/ghidra/releases
- С интерфейсом у этого инструмента примерно так же, как с защитой от загрузки из России
- Зато бесплатно и даже с открытым кодом
- С недавних пор также имеет отладчик


CodeBrowser: ghidraprojffs:/a.out
File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
a.out
.bss
.data
.got
.dynamic
.fini_array
.init_array
.eh_frame
.eh_frame_hdr
.rodata
.fini
.text
.plt.got
Program Tree
Symbol Tree
__libc_csu_fini
__libc_csu_init
_end
_fini
_init
_IO_stdin_used
_start
completed.7697
data_start
frame_dummy
main
Functions
Labels
Classes
Namespaces
Filter:
Data Type Manager
Data Types
BuiltInTypes
a.out
generic_clib
generic_clib_64
Filter:
Listing: a.out
*a.out
FUNCTION
undefined main()
undefined AL:1 <RETURN>
undefined4 Stack[-0xc]:4 local_c
main XREF[3]:
0010063a 55 PUSH RBP
0010063b 48 89 e5 MOV RBP,RSP
0010063e 48 83 ec 10 SUB RSP,0x10
00100642 c7 45 fc 7b 00 00 00 MOV dword ptr [RBP + local_c],0x7b
00100649 83 7d fc 64 CMP dword ptr [RBP + local_c],0x64
0010064d 7e 0e JLE LAB_0010065d
0010064f 48 8d 3d 9e 00 00 00 LEA RDI,[s_Hello_world_001006f4]
00100656 e8 b5 fe ff ff CALL puts
0010065b eb 0c JMP LAB_00100669
LAB_0010065d XREF[1]:
0010065d 48 8d 3d 9c 00 00 00 LEA RDI,[DAT_00100700]
00100664 e8 a7 fe ff ff CALL puts
LAB_00100669 XREF[1]:
00100669 b8 00 00 00 00 MOV EAX,0x0
0010066e c9 LEAVE
0010066f c3 RET
Decompile: main - (a.out)
/* WARNING: Removing unreachable block (ram,0x0010065d) */
undefined8 main(void)
{
puts("Hello world");
return 0;
}
Console - Scripting
00100649
main
CMP dword ptr [RBP + -0x4],0x64

Ghidra

- Чтобы открыть в нем исполняемый файл, нужно сначала создать проект через File -> New Project
- Затем нужно перетащить в этот проект исполняемый файл
  - Потыкав ОК, когда нужно
- После этого нужно нажать двойным щелчком на ваш файл
- Затем нужно нажать "Analyze" во всплывающем окне
- Отлично, вам удалось добиться того, что в IDA делается в одну кнопку

Radare2

- Свободный фреймворк для реверс-инжениринга
- Не просто дизассемблер, имеет множество утилит командной строки
  - Например radiff2, позволяющий посмотреть отличия между исполняемыми файлами, как diff в Linux
- Также имеет графический интерфейс Cutter
- Порог вхождения примерно на уровне Vim, поэтому его мы изучать не будем

Скачать бесплатно (или не очень)

- IDA
  - Бесплатная версия доступна на https://www.hex-rays.com/products/ida/support/download_freeware/
    - Ее достаточно, если вам нужен дизассемблер (и декомпилятор, но с ограничениями) для x86-64
  - Совершенно не стоит скачивать полную IDA с https://rutracker.org/, потому что там вы можете «стать жертвой подделки программного обеспечения», как говорят в Microsoft
- Ghidra
  - Доступен на https://github.com/NationalSecurityAgency/ghidra/releases
- Radare2
  - Можно скачать с откуда-то с https://rada.re или https://github.com/radareorg/cutter/releases

Время задач

Real CrackMe

Категория: Lesson 15 / Assembly + Disassembly basics

Решивших: 0

Время: 00:00:03

- Доступ к задачам можно получить как обычно на nsuctf.ru
- Вам может пригодиться IDA Free https://www.hex-rays.com/products/ida/support/download_freeware/ или Ghidra https://github.com/NationalSecurityAgency/ghidra/releases

Спасибо за внимание!
Задачи доступны на

nsuctf.ru

- Пожалуйста, используйте имя пользователя формата "Фамилия Имя"
  - e-mail можно забить любой, сервером он не проверяется
- Для вопросов по задачам рекомендую присоединиться к @NSUCTF в Telegram
  - Только, пожалуйста, без спойлеров