ИТМО

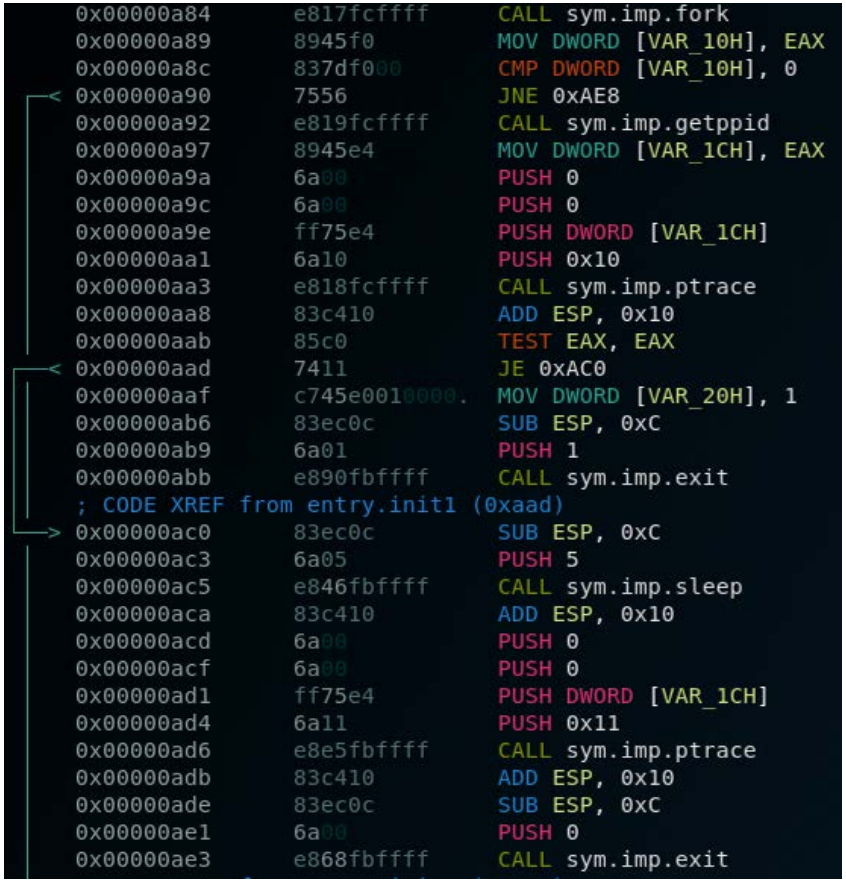
Лекция №7. Введение в обратную разработку

Содержание

- **Что такое реверс**
- **Архитектура ЭВМ**
- **Подробнее о процессоре**
- **Инструкции процессора**
- **Архитектура x86-64**
- **Исполняемые файлы**
- **Формат ELF**
- **Утилиты для реверса**
- **Как учиться реверсу**
- **Полезные источники**

Что такое Реверс?

Реверс – процесс анализа скомпилированного кода для того, чтобы понять его работу. В рамках соревнований CTF необходимо понять принцип работы программы, чтобы получить флаг. Для этого чаще всего используются такие инструменты, как дизассемблер (IDA Pro, Radare2, Ghidra, …), декомпилятор (IDA Pro, dnSpy, JDProject, …) и дебаггер (x64dbg, WinDbg, GDB, ...).



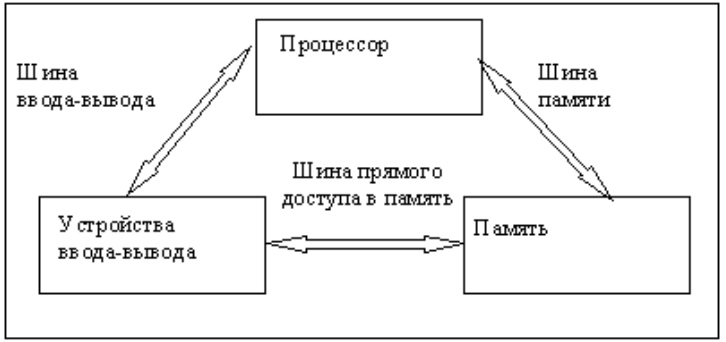
Архитектура ЭВМ

Любой современный компьютер состоит из 3 ключевых абстрактных компонентов:

Процессор, память, устройства ввода-вывода.

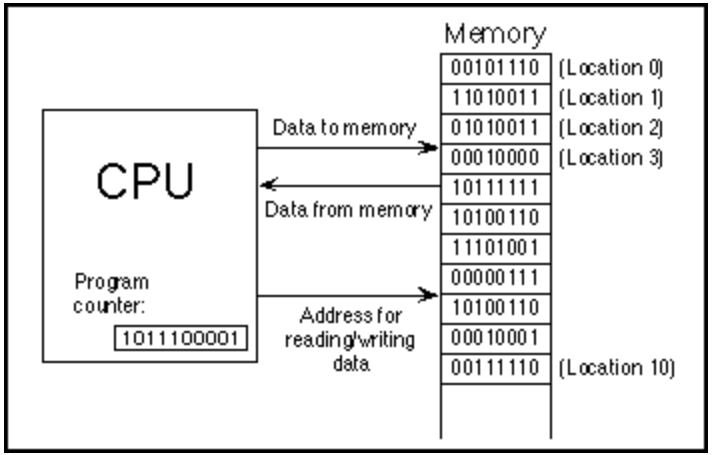
Основные задачи компонентов:

● Процессор — берет из памяти команду и исполняет её
● Память — просто массив ячеек, способных хранить числа
● Уст-ва ввода-вывода — позволяют заносить данные в компьютер



Подробнее о процессоре

Процессор — просто исполнитель команд
Регистры процессора — ячейки памяти **внутри** процессора. Они позволяют управлять процессором и организовывать алгоритм вычислений. Каждый регистр имеет своё имя.

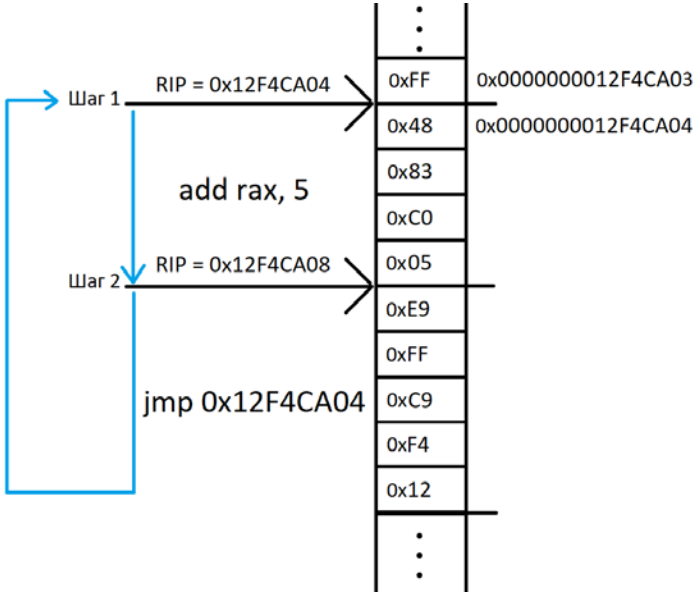


Простая аналогия:
память — массив
регистры — переменные

Архитектура процессора — это некий набор свойств и качеств, присущий целому семейству процессоров. Архитектура определяет как регистры, так и множество команд, которые «понимает» процессор.

Инструкции процессора

Для процессора абсолютно всё - это просто байты. Он может интерпретировать их либо как инструкцию, либо как данные. Если специальный регистр — счётчик команд (RIP) указывает на байты то процессор исполнит эти байты как инструкцию.



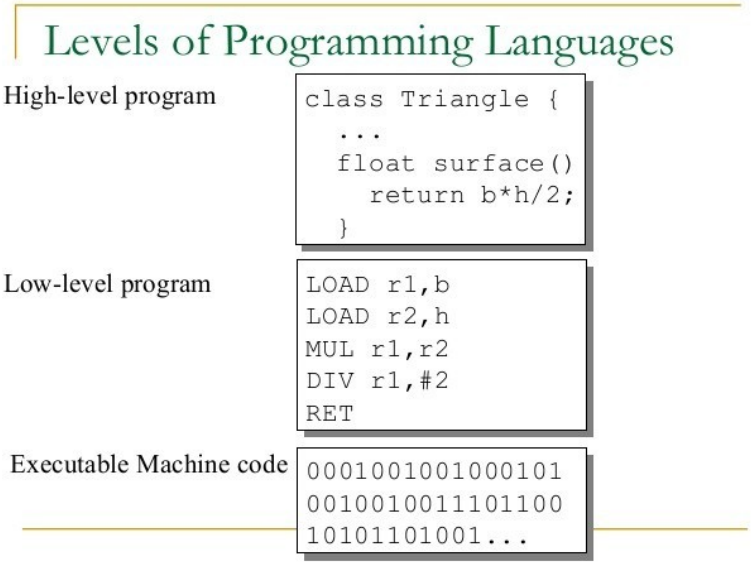
Пример:

Для процессора архитектуры X86-64 (Intel) последовательность байтов: 48 83 C0 05 E9 FF C9 F4 12 означает 2 инструкции:

1) Прибавить 5 к регистру rax
2) Начать выполнять инструкции по адресу 0x12F4CA04

Ассемблер

Писать программы байтами не удобно
Поэтому люди придумали записывать инструкции процессора понятным для человека способом. Запись машинной инструкции в читаемом для человека виде называется **мнемоникой**



Ассемблер — всего лишь определенный способ записи машинных кодов
Языков ассемблера существует много. Для каждой архитектуры свой.

Также существует 2 синтаксиса записи мнемоник: AT&T и Intel синтакис

Архитектура X86-64: Введение

Отрицательные числа в памяти компьютера
Представляются в т. н. дополнительном коде

Числа со старшим битом 1 считаем
отрицательными, причём число с единицей
в последнем разряде и нулём в остальных
наименьшее, а число с 1 во всех разрядах есть
обратное единице.

```
00000000 =  0
00000001 =  1
     ...
     ...
01111111 =  127
10000000 = -128
     ...
     ...
11111111 = -1
```

Для нахождения обратного числа надо изменить
все биты на противоположное значение и прибавить
единицу
Пример для байта: b00000001 = 1; b11111110 + 1 = -2 + 1 = -1;

Little и Big Endian

Little и Big Endian обозначают порядок записи **байтов**. Big Endian – привычный для нас вариант записи: от старшего к младшему. Little Endian – наоборот от младшего к старшему.

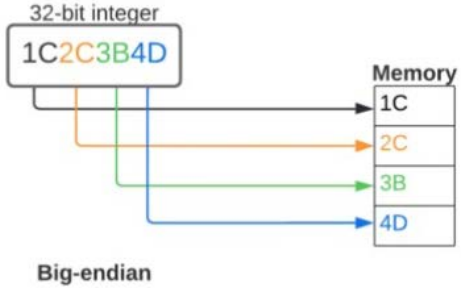
Пример:

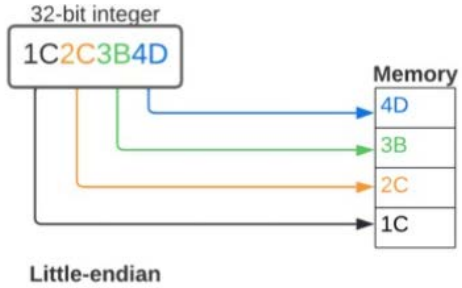
Возьмем шестнадцатеричное число 0x123456.

Поделим его по байтам: 0x12, 0x34, 0x56.

В Big Endian число так и запишется: 0x123456

В Little Endian байты будут записаны наобортот: 0x563412



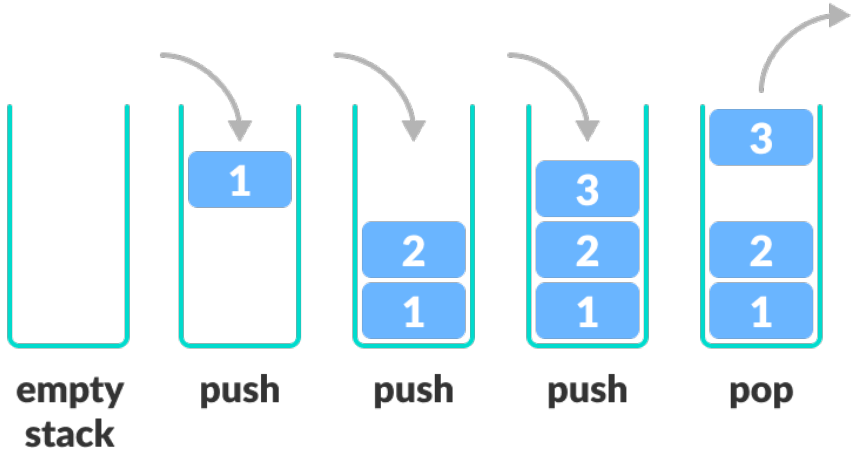


Архитектура X86-64: Регистры

Size (in Bits)			
64	32	16	8
RAX	EAX	AX	AH/AL
RBX	EBX	BX	BH/BL
RCX	ECX	CX	CH/CL
RDX	EDX	DX	DH/DL
RDI	EDI	DI	DIL
RSI	ESI	SI	SIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8∼R15	R8D∼R15D	R8W∼R15W	R8L∼R15L

Регистры общего назначения в x86_64 имеют длину 64 бита.
Также есть возможность адресовать разные части регистра.
Также есть 2 регистра:
RIP — указатель инструкции
RSP — указатель стека

Архитектура X86-64: Стек



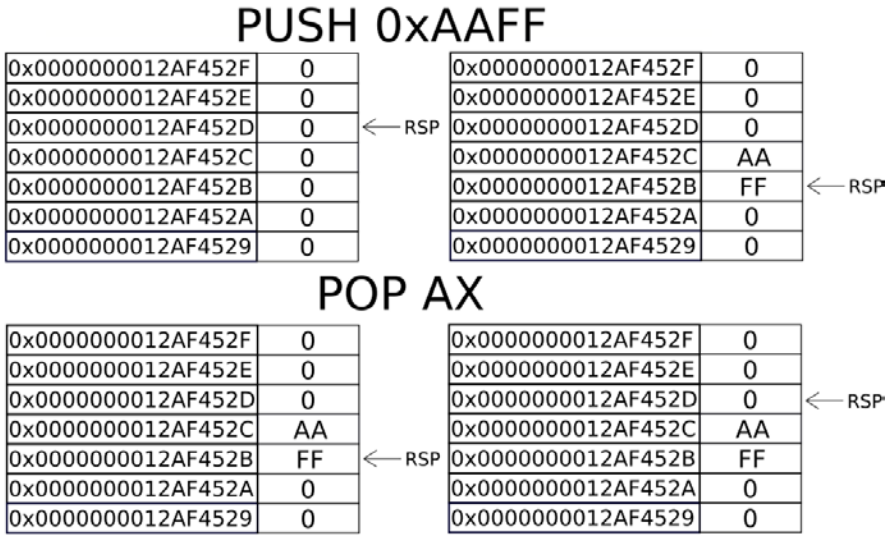
Стек — фундаментальная структура в компьютерных науках
Принцип прост: Положить на верх либо взять с вершины
Стек удобен для организации вычислений. В x86-64 стек используется для работы функций, хранения локальных вычислений, передачи параметров и даже для работы с дробными числами

Команды работы со стеком

RSP указывает на байт, который является вершиной стека

PUSH arg — уменьшает RSP на размер аргумента r и записывает по адресу в RSP аргумент arg

POP reg — считывает значение по адресу в RSP в регистр reg затем уменьшает RSP на размер reg



Обратите внимание — стек растёт в сторону меньших адресов.
Поскольку RSP уменьшается при каждой инструкции PUSH

Архитектура X86-64: Основные команды

Синтаксис мнемоник таков:

Intel: операция назначение, источник

AT&T: операция источник, назначение

Примеры:

addq %rax, %rbp — записать в rbp значение rbp + rax

add rbp, rax — тоже самое в синтаксисе Intel

movl $4, %eax — занести число 4 в eax

mov eax, 4 — тоже самое в синтаксисе Intel

Пересылка данных

Основная команда ассемблера — **mov**
Она напоминает операцию присваивания в языках программирования

Синтаксис: mov источник, назначение (для AT&T)
Примеры: см. предыдущий слайд

Стоит отметить что mov не умеет перемещать данные из памяти
в память. Поэтому один из операндов всегда должен быть либо регистром
либо константой. Также можно использовать указатели:

AT&T: movq 5(%rbx), %rax – записать 8 байт по адресу в (rbx + 5) в rax
Intel: mov rax, [rbx + 5]

Работа с указателями

Команда **lea** — Load Effective Address
помещает адрес источника в назначение

Синтаксис: lea источник, назначение (для AT&T)
Примеры:

AT&T: leaq (%rsp), %rax – записать в rax адрес, в rsp
Intel: lea rax, [rsp]

Команда выше аналогична mov %rsp, %rax

AT&T: leaq b, %rdx – записать в rax адрес переменной b
Intel: lea rdx, b

Арифметика

В архитектуре x86 достаточно арифметических команд (AT&T):

inc операнд
dec операнд

Увеличивает/уменьшает значение операнда на 1

add источник, приёмник
sub источник, приёмник

Сложение/вычитание

mul операнд
div операнд

Умножение/Деление

Побитовые инструкции

В архитектуре x86 достаточно арифметических команд:
Синтаксис основных команд (в AT&T)

and источник, приемник
or источник, приемник

Побитовые логические операции “И” и “ИЛИ”

xor источник, приёмник
not операнд

Побитовое исключающее “ИЛИ”, инвертирование битов в регистре

Архитектура X86-64: Флаги

Регистр **EFLAGS** используется для хранения состояния процессора. Состояние представлено битами в этом регистре. Нам интересны флаги **CF ZF SF OF**
Они устанавливаются в результате арифметических или логических операций.

CF (carry flag) — устанавливается при переносе из знакового бита результата или при заеме в знаковый бит

ZF (zero flag) — устанавливается если результат последнего действия ноль

SF (sign flag) — знак результата последнего действия

OF (overflow flag) — выставляется если в результате сложения двух положительных чисел получилось отрицательное и наоборот (арифметическое переполнение)

```
  11111111
+ 00000001
----------
 100000000
```

carry flag

```
  01111111 = 127
+ 01111111 = 127
----------
  11111110 = -2 (SF = 1 OF = 1)

  01111111 =  1
+ 11111110 = -2
----------
  11111111 = -1 (SF = 1 OF = 0)
```

Архитектура X86-64: Ветвление

В ассемблере нет оператора if. Эта операция делается в 2 этапа:

1) Команда **cmp** операнд2, операнд1 выполняет вычитание:
операнд_1 – операнд_2 , но не заносит результат в операнд1
Её задача — выставить флаги в EFLAGS без изменения значений Регистров.

2) Семейство команд условного перехода jX проверяют условие X и совершают переход по нужному адресу. Условие X определяется состоянием флагов в EFLAGS

Инструкции перехода

Примеры инструкций условного перехода:

В данной таблице Выше/Ниже означает больше/меньше в отношении беззнаковых чисел
В самом деле:

A = 01111111
B = 11111111

Если мы рассматриваем A,B как числа со Знаком, то A = 127 > B = -1

Но если A,B беззнаковые, то
A = 127 < B = 255

команда	отношения между операндами операнд1, операнд2 cmp	флаги
ja/jnbe	Выше	CF=0 и ZF=0
jae/jnb	Выше или равно	CF=0
jb/jnae/jc	Ниже	CF=1
jbe/jna	Ниже или равно	CF=1 и ZF=1
je/jz	Равно	ZF=1
jg/jnle	Больше	ZF=0 и SF=OF
jge/jnl	Больше или равно	SF=OF
jge/jnl	Больше или равно	SF=OF
jl/jnge	Меньше	SF≠ OF
jle/jng	Меньше или равно	ZF=1 или SF≠ OF
jne/jnz	Не равно	ZF=0
jno	Не переполнение	OF=0
jns	Не знак	SF=0
jo	Переполнение	OF=1
js	Знак	OF=1
jmp	Переход происходит всегда	---

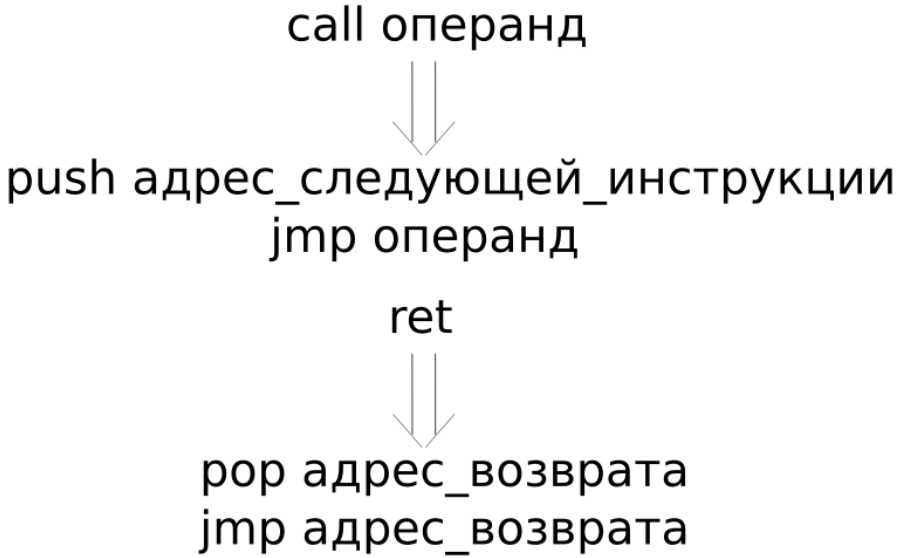
Вызов подпрограмм

Большой код программы надо как то структурировать

Удобнее всего структурировать вычисления, деля программу на подпрограммы или функции
Для этого есть команды **call** и **ret**

Команда call выполняет переход по адресу, который содержится в её операнде, сохранив перед эти адрес следующей команды (адрес возврата)

ret выполняет обратную операцию

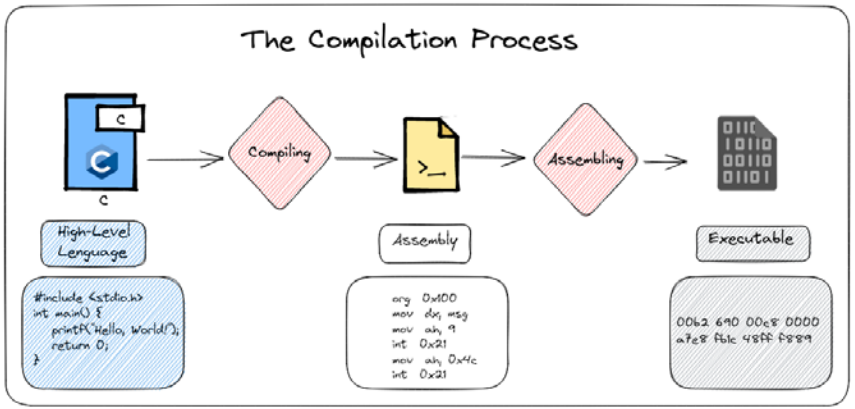


Исполняемые файлы

Фактически код любой программы открыт — просто этот код машинный

Это даёт нам возможность при должном желании и умении разобраться в том, как она работает

Этот код находится в исполняемом файле программы



Однако исполняемый файл - это не просто набор инструкций CPU помимо инструкций программе требуются данные, информация о библиотеках, от которых она зависит и так далее. Вся эта важная Информация и содержится в исполняемых файлах

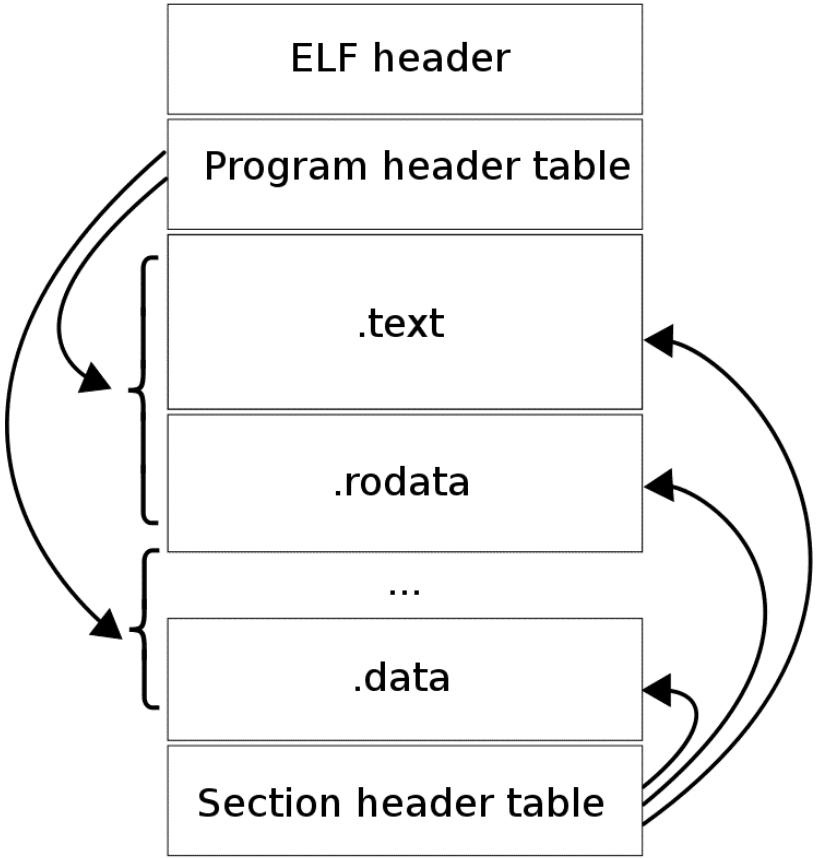
Формат ELF

ELF — Executable and Linkable Format

Формат elf предполагает разделение программы на части, а именно деление на заголовки программы и
на секции

Заголовки нужны при загрузке программы в память, они указывают как расположить части программы в памяти

Секции нужны для линковки программы и релокаций



Утилиты для реверса

Для анализа кода программ в первую очередь необходимы 2 утилиты: Дизасемблер и Отладчик.

На самом деле инструментарий может быть сколь угодно велик:
от трассировщиков системных вызовов до декомпиляторов

На данный момент существует 4 фреймворка для реверса:
IDA
Ghidra
Radare2
BinaryNinja

IDA Pro

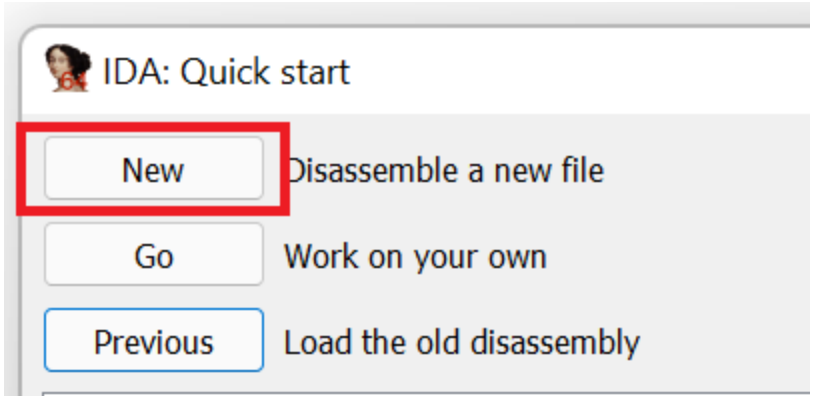
IDA Pro пожалуй самый популярный инструмент для статического анализа. Это **интерактивный** дизассемблер, что позволяет анализировать программы более эффективно. Также часто инструмент идет с дополнением от Hex-Rays – декомпилятором исходного кода, который значительно ускоряет анализ.

Подготовка

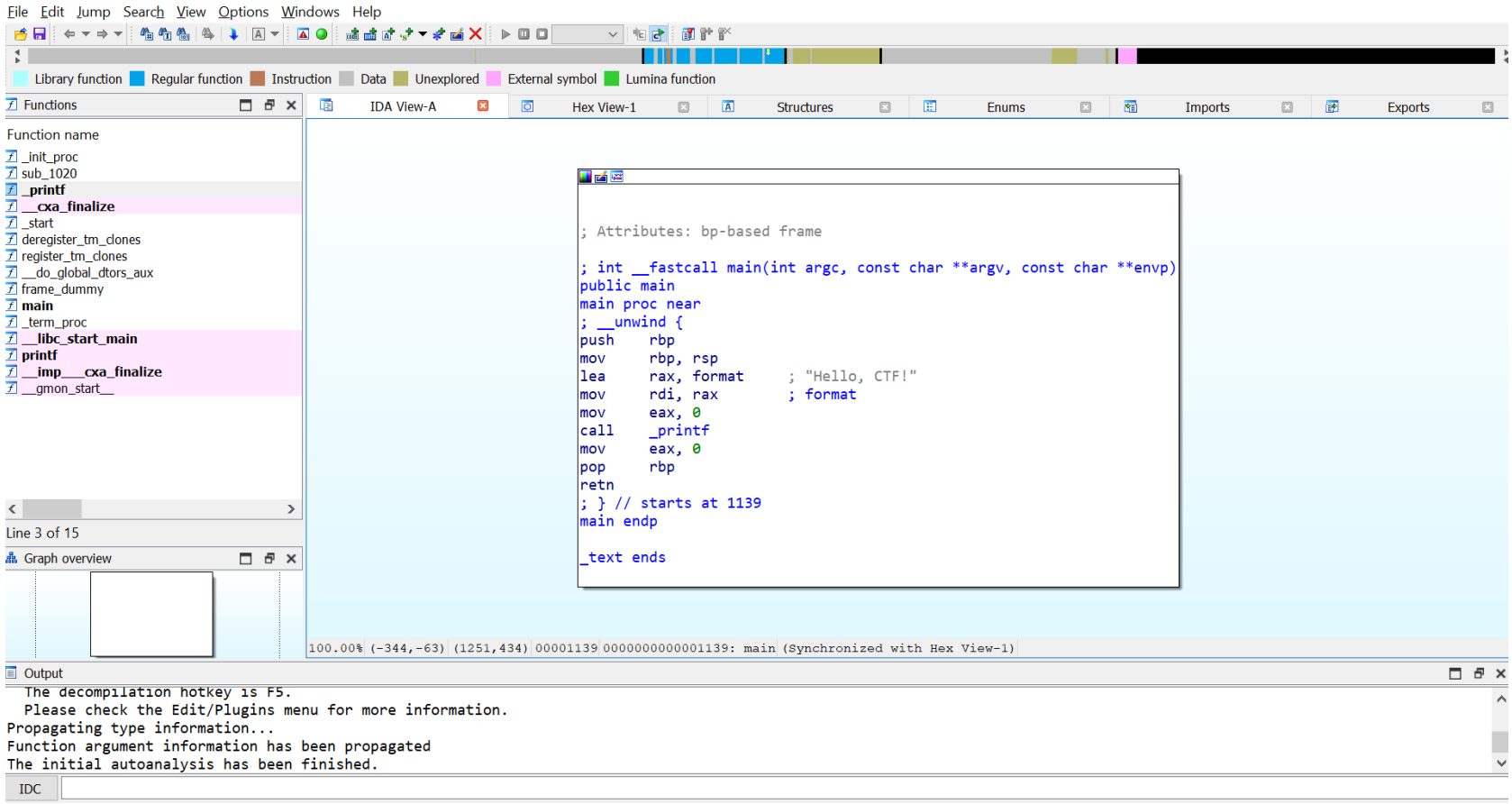
Напишем простую программу на C.
После скомпилируем ее и откроем исполняемый файл в IDA, нажав new и выбрав полученный исполняемый файл.

```c
#include <stdio.h>

int main()
{
        printf("Hello, CTF!");
        return 0;
}
```



Окно с дизассемблером



Что находится в окне IDA

Посередине находится большое окно, внутри которого расположен блок с дизассемблером. Это IDA нашла и открыла функцию main в виде графа. Если бы main был больше, здесь было бы несколько соединенных между собой блоков с дизассемблером, которые описывали бы переходы внутри функции с помощью инструкций jump. Нажав на пробел, мы выйдем из графа и увидим сплошной дизассемблер. Слева находится окно со списком функций внутри исполняемого файла.

Разбор функции main

То что написано после ; - комментарии иды. Она нам показала прототип функции и где main начинается и заканчивается. Также есть комментарий напротив строки **lea rax, format**, в котором написана наша строка. Инструкция lea вычисляет адрес строки и помещает в регистр **rax**. В следующей строке: **mov rdi, rax** адрес помещается в регистр **rdi**, который обозначает 1 аргумент, передаваемой функции. После обнуления rax (mov eax, 0), вызывается printf: **call _printf**.

```
; int __fastcall main(int argc, const char **argv, const char **envp)
                public main
main            proc near               ; DATA XREF: _start+14↑o
; __unwind {
                push    rbp
                mov     rbp, rsp
                lea     rax, format     ; "Hello, CTF!"
                mov     rdi, rax        ; format
                mov     eax, 0
                call    _printf
                mov     eax, 0
                pop     rbp
                retn
; } // starts at 1139
main            endp
```

Декомпилятор

Нажав на F5, откроется новое окно “Pseudocode-A”. В нем находится декомпилированная функция main в C формате. Декомпилятор понял, что в функцию printf передается строка “Hello, CTF!“ и написал это, что серьезно улучшило читаемость кода.

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  printf("Hello, CTF!");
  return 0;
}
```

Как учиться реверсу

Реверс — сложная область, она требует много знаний и навыков

Примерный алгоритм таков:
Первым делом стоит выучить C/C++

Затем выучить asm и как языки выше в него компилируются

Знать устройство операционных систем

Практикуясь получать новые знания
И опыт

Полезные источники

- https://beginners.re/
- https://www.reddit.com/r/ReverseEngineering
- https://godbolt.org/
- https://yutewiyof.gitbook.io/intro-rev-ida-pro/
- https://ru.wikibooks.org/wiki/Ассемблер_в_Linux_для_программистов_C
- https://en.wikibooks.org/wiki/X86_Assembly
- https://www.youtube.com/playlist?list=PLLguubeCGWoZIZBfJ1ZfJytjHLZyFFrMk
- https://ctf101.org/reverse-engineering/overview/
- https://www.nandgame.com/

ИТМО

Спасибо за внимание!