

Обратная разработка программного обеспечения. IDA

Игорь Черватюк
Александр Трифанов
Андрей Басарыгин

Москва, 2018

Терминология

- Обратная разработка программ (или реверс-инжиниринг, или reverse engineering, в контексте CTF – просто Reverse).

Стадии разработки ПО

- Техническое задание
- Разработка архитектуры приложения
- Написание кода
- Компиляция/Сборка
- Результат – исполняемый файл

Стадии обратной разработки ПО

- Исполняемый файл
- Дизассемблер/Отладчик /Декомпилятор
- Восстановление логики работы приложения
- Результат – исходный код

План лекции

Форматы файлов:

PE (Windows) и ELF (Linux)

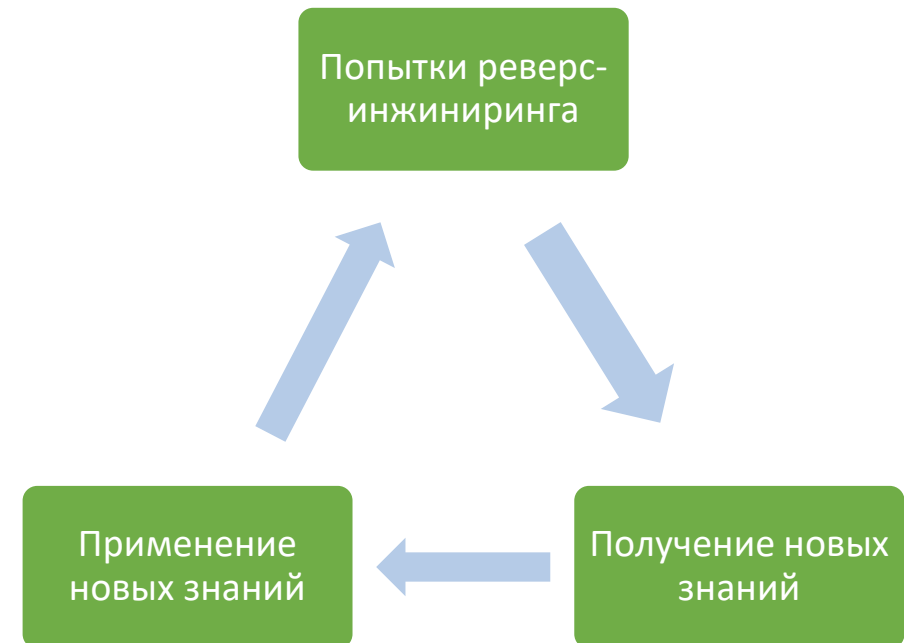
Статический (IDA) vs

Динамический анализ (Отладка)

Что мы охватим:

- Ассемблер
- Регистры
- Стек
- Функции
- Некоторые особенности компиляции

Обучение обратной разработке



PE и ELF

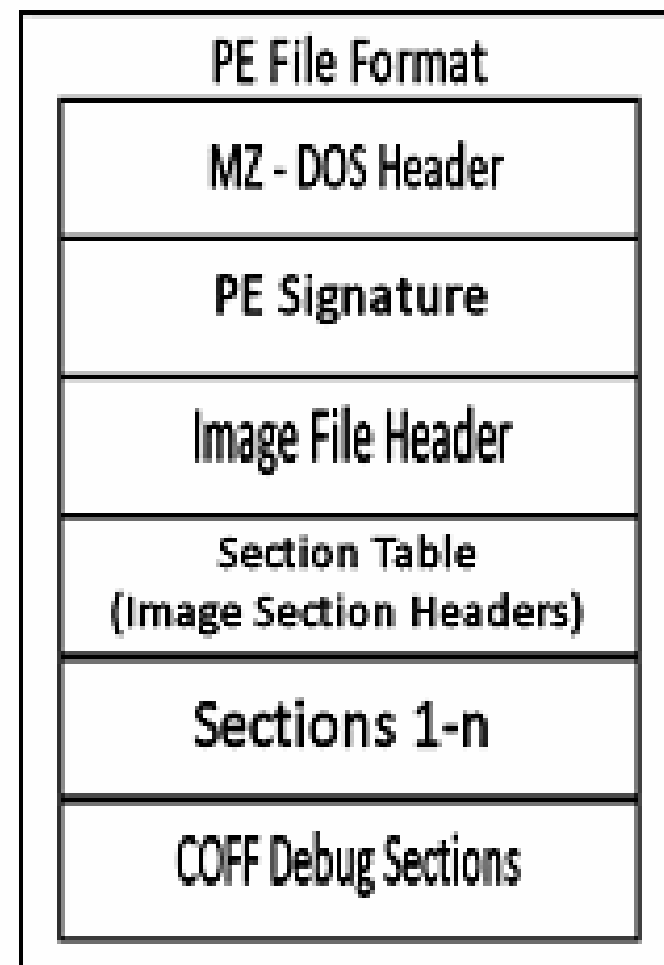
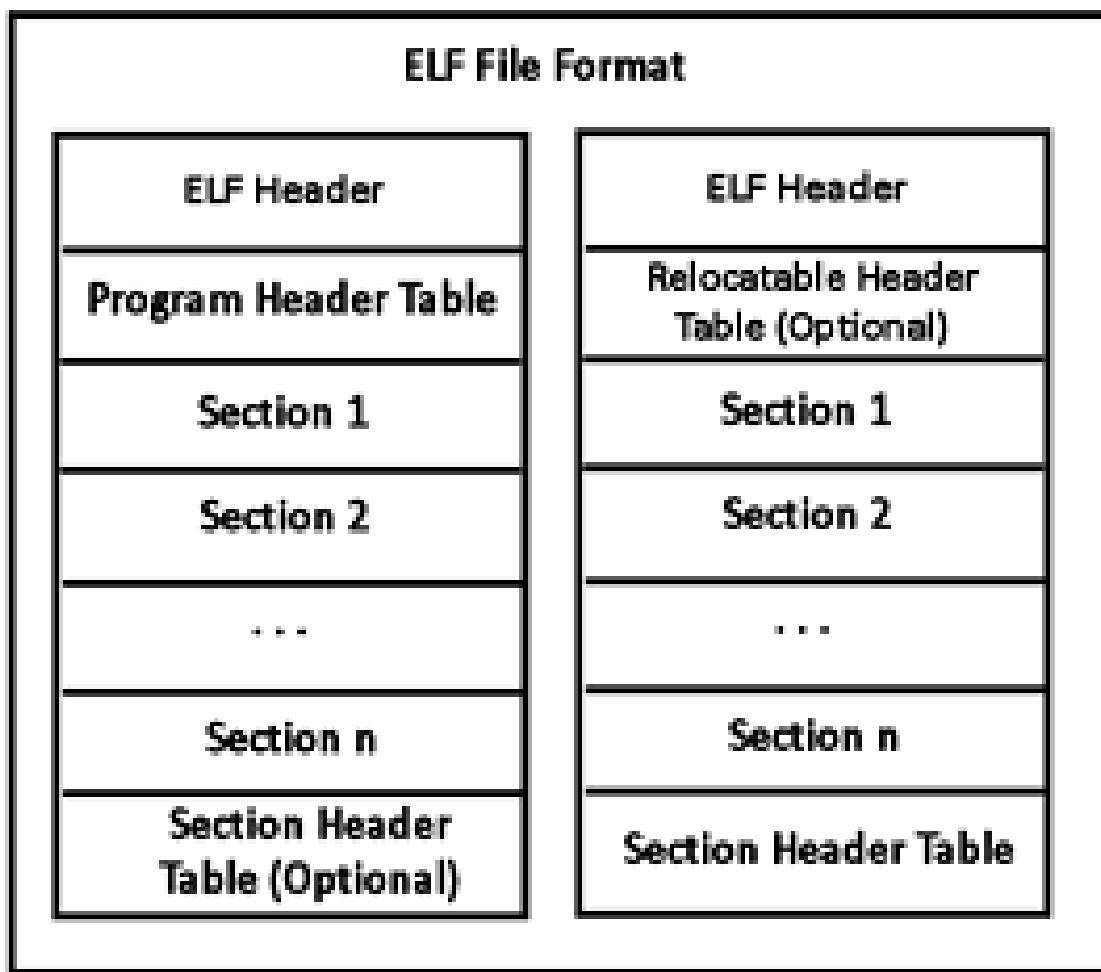
- PE (Portable Executable)

«формат исполняемых файлов, объектного кода и динамических библиотек, используемый в 32- и 64-битных версиях операционной системы Microsoft Windows» – *wikipedia*

- ELF (Executable and Linkable Format)

«формат двоичных файлов, используемый во многих современных UNIX-подобных операционных системах, таких как FreeBSD, Linux, Solaris и др» – *wikipedia*

PE и ELF



PE и ELF

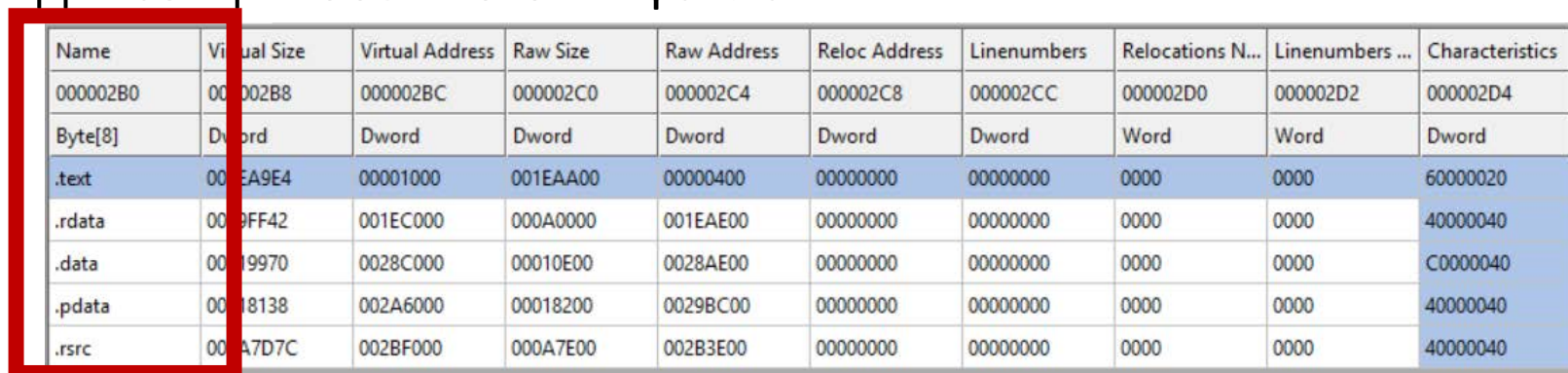
- Сильно глубоко форматы обсуждать не будем (на это могут уйти годы)
- Каждый из форматов – коллекция полей
- Поля хранят определенные значения
 - дата создания, дата последней модификации, количество секторов...
- Секции содержат код или данные
 - имеют разрешения (read/write/execute – помните NX-bit?)
 - имеют имя (.text, .bss, ...)

PE и ELF

- Зачем нам это надо?
- Понять что делает исполняемый файл
 - Какие библиотеки загружает
 - Какие функции из библиотек использует
 - Находить уязвимости
 - Искать строки в данных
 - Очень полезно на CTFs и что немаловажно – в реальной жизни 😊
 - Может помочь узнать запакован ли исполняемый файл
- Как их анализировать?
 - PE : CFF Explorer, IDA, pefile (python library), ...
 - ELF : *readelf, objdump, file*, ...

PE – CFF Explorer – пример обычного файла

- Так выглядят секции обычного PE-файла

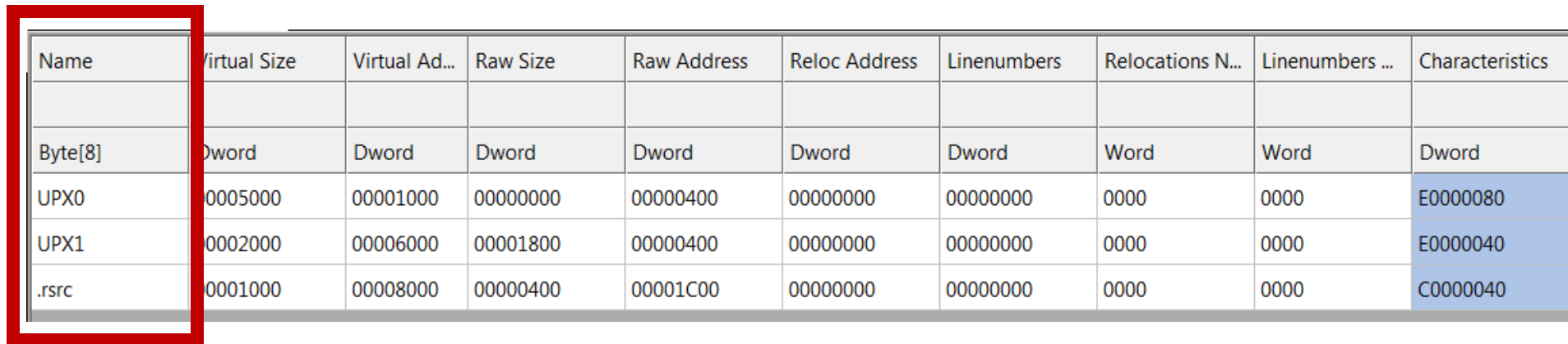


Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
000002B0	000002B8	000002BC	000002C0	000002C4	000002C8	000002CC	000002D0	000002D2	000002D4
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	000EA9E4	00001000	001EAA00	00000400	00000000	00000000	0000	0000	60000020
.rdata	0009FF42	001EC000	000A0000	001EAE00	00000000	00000000	0000	0000	40000040
.data	00099970	0028C000	00010E00	0028AE00	00000000	00000000	0000	0000	C0000040
.pdata	00018138	002A6000	00018200	0029BC00	00000000	00000000	0000	0000	40000040
.rsrc	000A7D7C	002BF000	000A7E00	002B3E00	00000000	00000000	0000	0000	40000040

- .text – это «executable code» - код программы
- .rdata – это «read-only initialized data» - это инициализированные константы
- .data – «initialized data» - это инициализированные данные
- .pdata – «exception information» - обработка исключений происходит здесь
- .rsrc – «resource directory» - здесь хранятся ресурсы
- Более подробно можно посмотреть здесь [https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547\(v=vs.85\).aspx#section_table_section_headers](https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx#section_table_section_headers)

PE – CFF Explorer – пример с UPX

- Так выглядят секции упакованного файла



Name	Virtual Size	Virtual Ad...	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
UPX0	0005000	00001000	00000000	00000400	00000000	00000000	0000	0000	E0000080
UPX1	0002000	00006000	00001800	00000400	00000000	00000000	0000	0000	E0000040
.rsrc	0001000	00008000	00000400	00001C00	00000000	00000000	0000	0000	C0000040

- Обратите внимание на имена секций – UPX – это весьма распространённый упаковщик, в большинстве случаев команда `upx -d имя_файла` распакует файл (это в Linux, для Windows можно использовать PE-explorer и плагин UPX unpacker).

ELF - readelf

Использование *readelf* для просмотра заголовков секций

```
:~$ readelf -S a.out
```

There are 8 section headers, starting at offset 0x70:

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00000000	000034	00000a	00	AX	0	0	4
[2]	.rel.text	REL	00000000	000208	000008	08		6	1	4
[3]	.data	PROGBITS	00000000	000040	000000	00	WA	0	0	4
[4]	.bss	NOBITS	00000000	000040	000000	00	WA	0	0	4
[5]	.shstrtab	STRTAB	00000000	000040	000030	00		0	0	1
[6]	.symtab	SYMTAB	00000000	0001b0	000050	10		7	4	4
[7]	.strtab	STRTAB	00000000	000200	000005	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

PE and ELF - Imports

- В IDA можно просматривать импортируемые функции:
 - **View-> Open subviews->Imports**
 - Справа пример →→→
- Внизу можно заметить хитрый фильтр `.*str.*`












011CC4D8		FreeEnvironmentStringsA	KERNEL32
011CC550		IsBadStringPtrA	KERNEL32
011CC554		IsBadStringPtrW	KERNEL32
011CC558		lstrcpyA	KERNEL32
011CC564		lstrcpyW	
011CC56C		lstrcpmA	
011CC57C		lstrcmpW	
011CC598		lstrcmpiW	
011CC5A0		GetStringTypeExW	
011CC5C0		lstrcmpA	
011CC5C4		lstrlenA	
011CC5D4		lstrcatW	KERNEL32
011CC644		GetProfileStringW	KERNEL32
011CC674		WritePrivateProfileStringW	KERNEL32
011CC6A0		lstrcpyW	KERNEL32
011CC6B4		GetPrivateProfileStringW	KERNEL32
011CC714		lstrlenW	KERNEL32
011CC724		OutputDebugStringW	KERNEL32
011CC840	38	SafeArrayDestroyDescriptor	OLEAUT32
011CC844	39	SafeArrayDestroyData	OLEAUT32

Возможно стоит обратить на них внимание ;)

WR
WX `.*str.*`

PE and ELF - Строки

- **View-> Open subviews->Strings**

Address	Lenath	Type	String
 .rdata:004020D6	00000004	unico...	@
 .rdata:004020E6	00000004	unico...	@
 .rdata:0040210C	00000009	C	HoppaKey
 .rdata:00402118	00000028	C	Ups, some calls are wrong or missing =\\
 .rdata:00402140	00000012	C	Get your flag %s\\n ←
 .rdata:00402154	00000008	C	load_me
 .rdata:0040215C	0000000D	C	Kernel32.dll
 .rdata:0040216C	0000000D	C	LoadLibraryA
 .rdata:0040217C	0000000F	C	GetProcAddress
 .rdata:00402360	0000000D	C	KERNEL32.DLL
 .rdata:0040236D	0000000C	C	MSVCR90.dll

- Обычно стоит запустить программу и посмотреть что она нам пишет и искать соответствующие строки здесь.

Статический и динамический анализ

- Статический
 - Смотрим на код, пытаемся понять что он делает
 - Там есть всё, но это порой довольно сложно
 - Более безопасный подход (особенно если речь идёт об исследовании вредоносного программного обеспечения)
- Динамический
 - Исследуем процесс во время выполнения
 - Можем просматривать значения в реальном времени
 - Регистры, содержимое памяти, и т.д.
 - Мы можем управлять процессом
 - Безопасней делать это в виртуальной машине!
- Комбинированный подход: всегда можно запустить IDA и подключить к ней отладчик 😊

Инструменты

- Дيزассемблеры – статический анализ
 - IDA Pro, Binary ninja, objdump, etc.
 - Есть ещё radare2, но там очень хороший графический интерфейс, поэтому использовать мы его не будем 😊
- Отладчики – динамический анализ
 - Windows
 - WinDBG, Immunity, OllyDBG, x64dbg
 - Отладчики можно подключать к IDA
 - Linux
 - GDB

Инструменты

- Хороший дизассемблер умеет:
 - Комментарии
 - Переименование переменных
 - Изменение прототипов функций
 - Подсветка, группировка (IDA)
 - ...
- Хороший отладчик умеет:
 - Устанавливать точки останова (breakpoints)
 - Пошаговое выполнение (Step into / over)
 - Показ загруженных модулей
 - Поиск в памяти
 - ...

Регистры процессора – глобальные переменные

- RAX, RBX, RDX, R8, R9, R10..R15 – регистры общего назначения
- RIP – указатель на следующую команду, которая будет исполнена
- RSP – указатель на вершину стека
- RBP – указатель на базовый адрес стека (я поясню чуть позже)
- RSI, RDI – «source index», «destination index» - источник и приёмник для некоторых вызовов функций, также могут использоваться как регистры общего назначения.

RAX		64 бита
EAX		32 бита
AX		16 бит
AH	AL	По 8 бит

Регистр флагов – RFLAGS (EFLAGS для 32 бит)

- Здесь важны биты
- CF - Carry Flag - Флаг переноса. Устанавливается в 1, если результат предыдущей операции не уместился в приёмнике и произошёл перенос из старшего бита или если требуется заём (при вычитании).
- PF - Parity Flag - Флаг чётности. Устанавливается в 1, если младший байт результата предыдущей команды содержит чётное количество битов, равных 1. Если количество единиц в младшем байте нечётное, то этот флаг равен 0.
- ZF - Zero Flag - Флаг нуля. Устанавливается 1, если результат предыдущей команды равен 0.
- SF - Sign Flag - Флаг знака. Этот флаг всегда равен старшему биту результата.

Отрицательные числа

- Чтобы из N получить -N нужно:
- Инвертировать все биты в N
- Прибавить единицу
- В обратную сторону – то же самое
- Сделано ради удобного вычитания (оно сводится к сложению)

0	0	0	0	0	1	0	1	
								=5
+	1	1	1	1	1	0	1	0
								1
1	1	1	1	1	0	1	1	
								= -5
знаковый бит		информационные биты						

Assembly варианты синтаксиса

AT&T

инструкция источник, назначение

`mov %eax, %edx`

“Move eax into edx”

`mov %rax, %rdx`

“Move rax into rdx”

Intel

инструкция назначение, источник

`mov edx, eax`

“Move into edx, eax”

`mov rdx, rax`

“Move into rdx, rax”

Далее будем использовать только синтаксис Intel

Базовые команды:

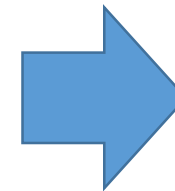
- `mov eax, ecx`
 - Поместить в `eax` содержимое `ecx`
- `mov eax, [ecx]`
 - Поместить в `eax` содержимое на которое `ecx` указывает
 - Скобки [...] означают разыменование указателя
 - В С это как `eax = *ecx`
- Можно использовать значения из памяти или константы
- `mov eax, 5`
 - Поместить в `eax` значение 5

Косвенная адресация:

- `mov edx, [0x12345678]`
 - Поместить в `edx` то, на что указывает `0x12345678`

```
eax 0x1234567D
ebx 0x1234567F
ecx 0x1234567A
edx 0x00000000
ebp 0xfffff00D
esp 0xfffff00A
esi 0x00000000
edi 0x00000000
eip 0x40000000
```

1	0x12345677	0x00000004
2	0x12345678	0x00000008
3	0x12345679	0x0000000F
4	0x1234567A	0x00000010
5	0x1234567B	0x00000017
6	0x1234567C	0x0000002A
7	0x1234567D	0x00000004
8	0x1234567E	0x00000008
9	0x1234567F	0x0000000F
10	0x12345680	0x00000010



```
eax 0x1234567D
ebx 0x1234567F
ecx 0x1234567A
edx 0x00000008
ebp 0xfffff00D
esp 0xfffff00A
esi 0x00000000
edi 0x00000000
eip 0x40000000
```

Assembly: call, mov, cmp, jmp

Очень небольшое количество инструкций позволят многое понять:

- `call 0x12345678` – Вызов функции по адресу `0x12345678`
- `cmp eax, 8` - Сравнить `eax` с 8
 - Сравнение идет слева-направо
 - На самом деле из `eax` вычитается 8 и по результатам выставляются флаги нуля (ZF) и переноса (CF)
- `jmp 0x12345678` – Безусловный переход по адресу `0x12345678`
- `jle 0x12345678` – Перейти по адресу `0x12345678` если `eax` ≤ 8
- `jg 0x12345678` – Перейти по адресу `0x12345678` если `eax` > 8
- `jz 0x41040300` – Перейти если установлен флаг нуля

Переход по адресу vs Вызов функции

jmp 0x12345678

- В регистр `еір` записывается `0x12345678`
- Управление передаётся следующей команде (т.е. команде, которая находится по адресу `0x12345678`)

call 0x12345678

- В стек записывается адрес возврата на инструкцию, следующую за инструкцией `call 0x12345678`
- В регистр `еір` записывается `0x12345678`
- Управление передаётся следующей команде

Assembly – Пример

```
080483b4 <main>:
80483b4:      55                push    ebp
80483b5:      89 e5            mov     ebp,esp
80483b7:      83 ec 10         sub     esp,0x10
80483ba:      c7 45 fc 04 00 00 00 mov     DWORD PTR [ebp-0x4],0x4
80483c1:      c7 45 f8 0a 00 00 00 mov     DWORD PTR [ebp-0x8],0xa
80483c8:      8b 45 fc         mov     eax,DWORD PTR [ebp-0x4]
80483cb:      3b 45 f8         cmp     eax,DWORD PTR [ebp-0x8]
80483ce:      7d 07           jge     80483d7 <main+0x23>
80483d0:      b8 01 00 00 00   mov     eax,0x1
80483d5:      eb 05           jmp     80483dc <main+0x28>
80483d7:      b8 00 00 00 00   mov     eax,0x0
80483dc:      c9              leave
80483dd:      c3              ret
```

Пример 1

- $[ebp-0x4] = 0x4$
- $[ebp-0x8] = 0xa$
- $eax = [ebp-0x4]$
- Два значения, по адресу $ebp-0x4$ и $ebp-0x8$ получили значения $0x4$ и $0xa$
- Регистру было присвоено значение

```
080483b4:
80483b4: push    ebp
80483b5: mov     ebp,esp
80483b7: sub     esp,0x10
80483ba: mov     DWORD PTR [ebp-0x4],0x4
80483c1: mov     DWORD PTR [ebp-0x8],0xa
80483c8: mov     eax,DWORD PTR [ebp-0x4]
80483cb: cmp     eax,DWORD PTR [ebp-0x8]
80483ce: jge     80483d7 <main+0x23>
80483d0: mov     eax,0x1
80483d5: jmp     80483dc <main+0x28>
80483d7: mov     eax,0x0
80483dc: leave
80483dd: ret
```

Пролог

Эпилог

Пример 1

- $[ebp-0x4] = 0x4$
- $[ebp-0x8] = 0xa$
- $eax = [ebp-0x4]$
- `cmp eax, [ebp-0x8]`
 - $eax == [ebp-0x8]$?
 - $4 == 10$?
- `jge 0x80483d7`
 - если $4 \geq 10$, перейти по адресу ...
 - иначе, продолжить выполнение

```
080483b4: push    ebp
80483b5: mov     ebp,esp
80483b7: sub     esp,0x10
80483ba: mov     DWORD PTR [ebp-0x4],0x4
80483c1: mov     DWORD PTR [ebp-0x8],0xa
80483c8: mov     eax,DWORD PTR [ebp-0x4]
80483cb: cmp     eax,DWORD PTR [ebp-0x8]
80483ce: jge     80483d7 <main+0x23>
80483d0: mov     eax,0x1
80483d5: jmp     80483dc <main+0x28>
80483d7: mov     eax,0x0
80483dc: leave
80483dd: ret
```

Пример 1

- ...
- `eax = [ebp-0x4]`
- `cmp eax, [ebp-0x8]`
 - `eax == [ebp-0x8] ?`
 - `4 == 10 ?`
- `jge 0x80483d7`
 - если `4 >= 10`, перейти по адресу ...
 - иначе, продолжить выполнение

```
080483b4:
80483b4: push    ebp
80483b5: mov     ebp, esp
80483b7: sub     esp, 0x10
80483ba: mov     DWORD PTR [ebp-0x4], 0x4
80483c1: mov     DWORD PTR [ebp-0x8], 0xa
80483c8: mov     eax, DWORD PTR [ebp-0x4]
80483cb: cmp     eax, DWORD PTR [ebp-0x8]
80483ce: jge     80483d7 <main+0x23>
80483d0: mov     eax, 0x1
80483d5: jmp     80483dc <main+0x28>
80483d7: mov     eax, 0x0
80483dc: leave
80483dd: ret
```

`4 >= 10` не является истиной, просто переходим к следующей инструкции

Пример 1

- `[ebp-0x4] = 0x4`
- `[ebp-0x8] = 0xa`
- `eax = [ebp-0x4]`
- `cmp eax, [ebp-0x8]`
- `jge 0x80483d7`
- `mov eax, 0x1`
 - `eax = 1`
- `jmp` через `mov eax, 0`
- `leave` и `return`

```
080483b4:
80483b4: push    ebp
80483b5: mov     ebp, esp
80483b7: sub     esp, 0x10
80483ba: mov     DWORD PTR [ebp-0x4], 0x4
80483c1: mov     DWORD PTR [ebp-0x8], 0xa
80483c8: mov     eax, DWORD PTR [ebp-0x4]
80483cb: cmp     eax, DWORD PTR [ebp-0x8]
80483ce: jge     80483d7 <main+0x23>
80483d0: mov     eax, 0x1
80483d5: jmp     80483dc <main+0x28>
80483d7: mov     eax, 0x0
80483dc: leave
80483dd: ret
```

Пример 1

- Давайте копать дальше
- Всё что там есть имеет значение
- `mov DWORD PTR [ebp-0x4], 0x4`
 - Что значит DWORD PTR?
- Скобки [...] получить значение по адресу... но DWORD PTR?

DWORD PTR

DWORD = размер, их бывает несколько

PTR = Указание на то, что это указатель

Пример 1

На размеры важно обращать внимание – это может помочь понять с каким типом данных работал разработчик

Тип	Размер (байты)	Размер (биты)	Обозначение на ассемблере	Пример
char	1 byte	8 bits	BYTE	char c;
short	2 bytes	16 bits	WORD	short s;
int	4 bytes	32 bits	DWORD	int i;
long long	8 bytes	64 bits	QWORD	long long ago;

Пример 1

- `mov DWORD PTR [ebp-0x4], 0x4`
- Команда говорит заполнить адрес `[ebp-4]` значением 4, размер этого значения 4 байта.
- `[ebp-4]` целое число
- Значит наш исходный код имеет целочисленную переменную и записывает в неё значение 4

Пример 1

- `mov DWORD PTR [ebp-0x4], 0x4`
- `mov DWORD PTR [ebp-0x8], 0xa`
- Две целочисленные переменные:
 - `int x = 4;`
 - `int y = 10;`
- Но эти переменные локальные, глобальные или статические???
- Для того чтобы ответить, нужно понять как распределяется память в приложении.

Пример 1

- int x = 4;
- int y = 10;
 - Мы не знаем где они находятся
- if (4 >= 10)
 - goto main+0x23 (0x80483d7)
- eax = 1
- goto main+0x28
- main+0x23 (0x80483d7) :
 - eax = 0
- main+0x28 (0x80483dc):
 - ret (Выход из программы)
- Это начинает напоминать исходный код!

```
080483b4 .
80483b4: push    ebp
80483b5: mov     ebp,esp
80483b7: sub     esp,0x10
80483ba: mov     DWORD PTR [ebp-0x4],0x4
80483c1: mov     DWORD PTR [ebp-0x8],0xa
80483c8: mov     eax,DWORD PTR [ebp-0x4]
80483cb: cmp     eax,DWORD PTR [ebp-0x8]
80483ce: jge     80483d7 <main+0x23>
80483d0: mov     eax,0x1
80483d5: jmp     80483dc <main+0x28>
80483d7: mov     eax,0x0
80483dc: leave
80483dd: ret
```

Выделение памяти

- Мы хотим знать
 - Почему адреса задаются относительно esp/ebp?
 - Что делают push/pop инструкции?
 - Что насчет leave/ret инструкций?

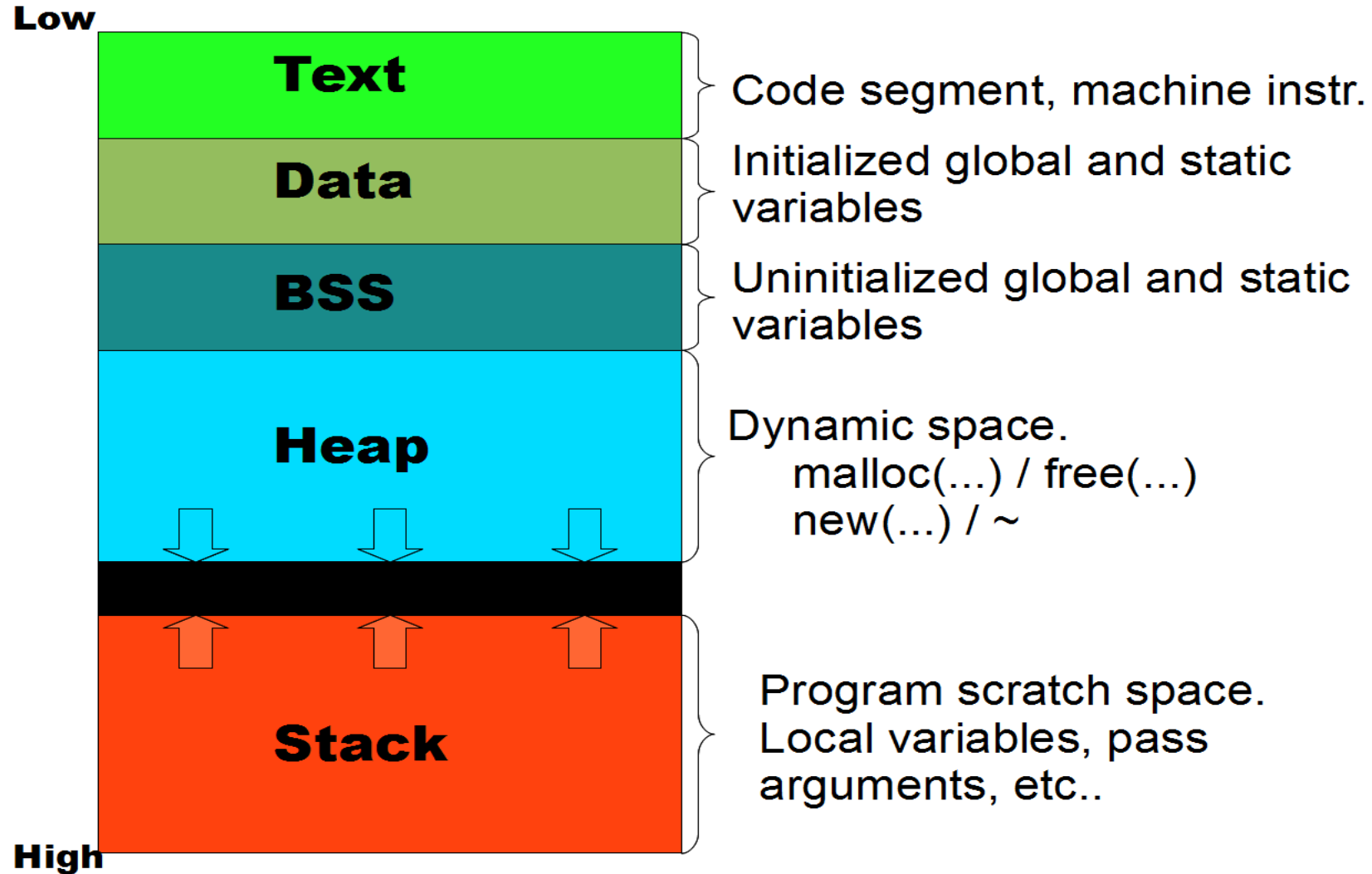
Распределение памяти Windows

0x00000000	Память ядра	Обычные программы не могут получить сюда доступ
	Стек (stack)	Когда что-то помещается в стек, вершина стека уменьшается
	Куча (heap)	Куча растет вниз (в сторону увеличения адресов)
0x00400000	Program image PE header .text (code) .rdata (imports) .data (data) .rsrc (resources)	
		Свободное пространство
	DLL	Подключаемые библиотеки
	DLL	Подключаемые библиотеки
...
0x7fffffff		

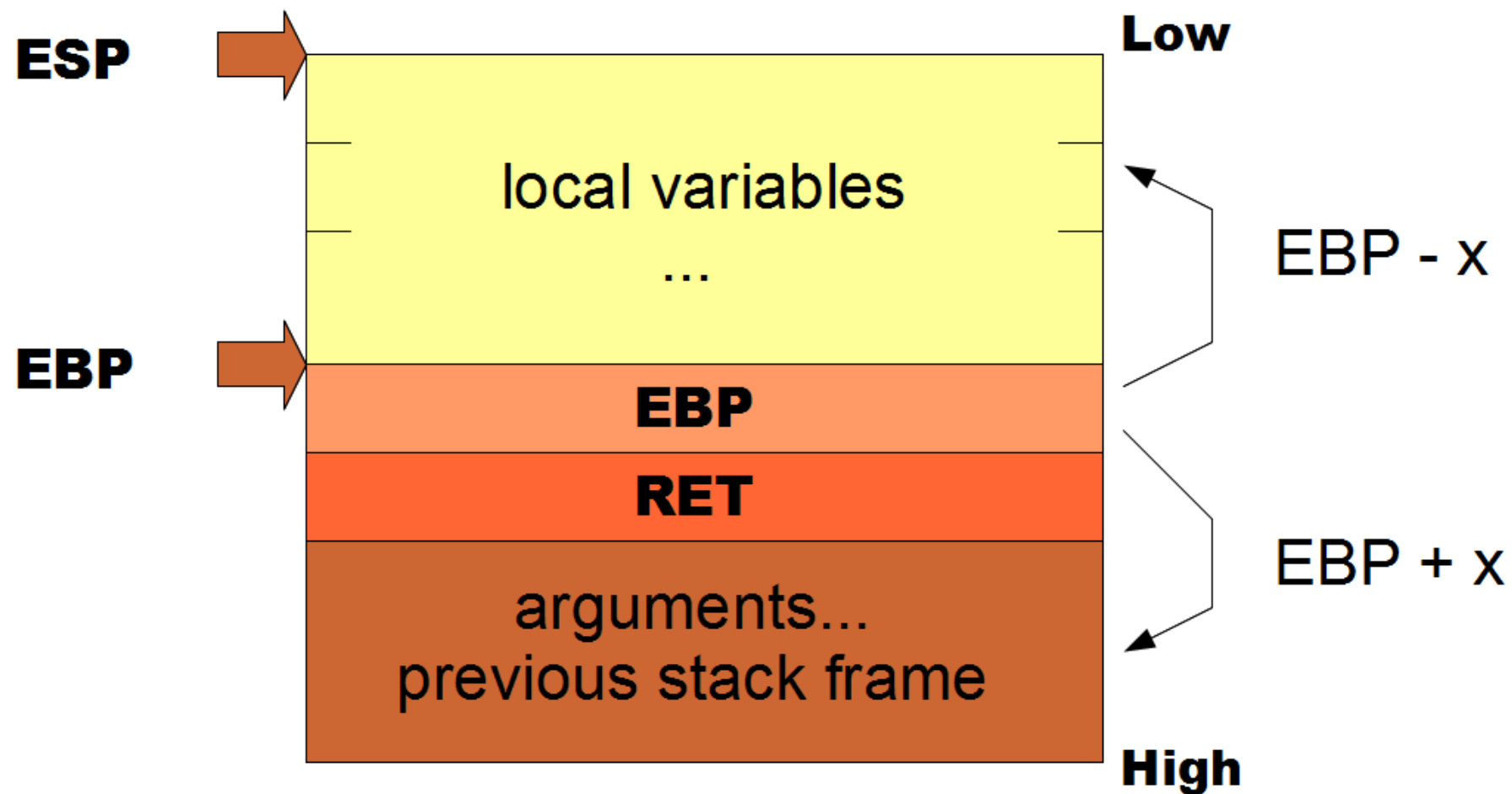
Распределение памяти Linux

...
	Память ядра	
	Стек (stack)	Когда что-то помещается в стек, вершина стека уменьшается
	DLL	Подключаемые библиотеки
		Свободное пространство
	Куча (heap)	Куча растет вниз (в сторону увеличения адресов)
	data	Глобальные переменные
0x00400000	Program image	
...

Virtual Memory



Стек



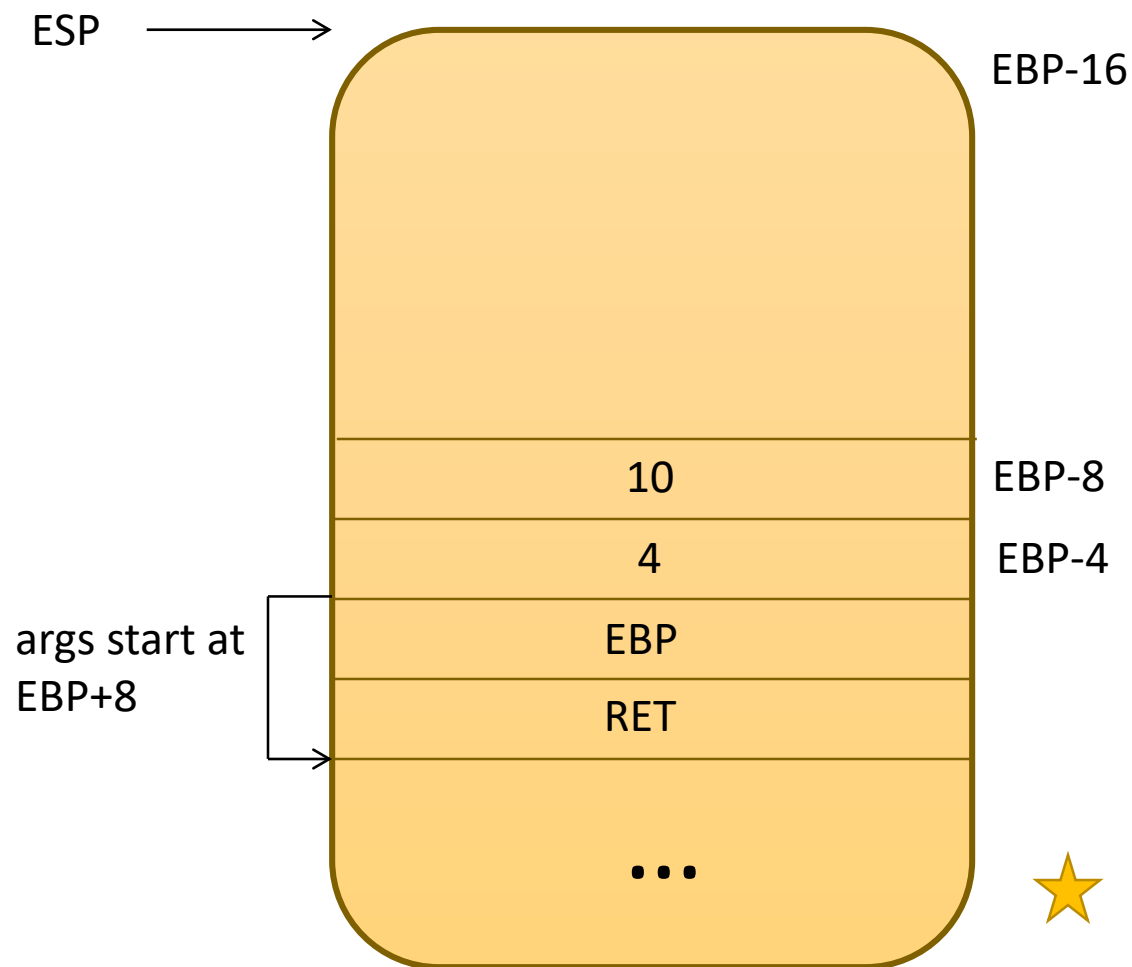
Пример 1 – Часть 2

- `sub esp, 0x10`
 - Комната размером 16 байт для локальных переменных, или 4 ints
- `[ebp-4]` локальная переменная
- `[ebp-8]` локальная переменная
- Возвращаемое значение, `eax`, это 1 или 0 зависит от результатов сравнения

Нам достаточно 8-ми байт скажете вы?
Да, но стек в Windows по умолчанию выравнивается кратно 16-ти байтам

```
080483b4:
80483b4: push    ebp
80483b5: mov     ebp, esp
80483b7: sub     esp, 0x10
80483ba: mov     DWORD PTR [ebp-0x4], 0x4
80483c1: mov     DWORD PTR [ebp-0x8], 0xa
80483c8: mov     eax, DWORD PTR [ebp-0x4]
80483cb: cmp     eax, DWORD PTR [ebp-0x8]
80483ce: jge     80483d7 <main+0x23>
80483d0: mov     eax, 0x1
80483d5: jmp     80483dc <main+0x28>
80483d7: mov     eax, 0x0
80483dc: leave
80483dd: ret
```


Пример 1 - Стек



```
push    ebp
mov     ebp,esp
sub     esp,0x10
mov     DWORD PTR [ebp-0x4],0x4
mov     DWORD PTR [ebp-0x8],0xa
mov     eax,DWORD PTR [ebp-0x4]
cmp     eax,DWORD PTR [ebp-0x8]
jge     80483d7 <main+0x23>
mov     eax,0x1
jmp     80483dc <main+0x28>
mov     eax,0x0
leave
ret
```



Нет $[ebp+x]$ – значит у функции нет аргументов

Пример 1 – Часть 2

```
int someFunction() {  
    int x = 4;  
    int y = 10;  
    if (4 >= 10)  
        goto main+0x23  
    eax = 1  
    goto main+0x28  
main+0x23 :  
    eax = 0  
main+0x28:  
    return  
}
```

```
080483b4  
80483b4: push    ebp  
80483b5: mov     ebp,esp  
80483b7: sub     esp,0x10  
80483ba: mov     DWORD PTR [ebp-0x4],0x4  
80483c1: mov     DWORD PTR [ebp-0x8],0xa  
80483c8: mov     eax,DWORD PTR [ebp-0x4]  
80483cb: cmp     eax,DWORD PTR [ebp-0x8]  
80483ce: jge     80483d7 <main+0x23>  
80483d0: mov     eax,0x1  
80483d5: jmp     80483dc <main+0x28>  
80483d7: mov     eax,0x0  
80483dc: leave  
80483dd: ret
```

Небольшая заметка об условиях

- Сравнение 'if' изменяется на противоположное
- Исходный код: `if x > y`
- Становится:
 - `cmp x, y`
 - `jle 0x12345678` (переход если x меньше либо равен y)
 - Если условие ***not true***, перепрыгиваем этот код
- `If x <= y`
- Становится
 - `cmp x, y`
 - `ja 0x12345678` (`jmp above`)

Пример 1 – Часть 2

```
int someFunction() {  
    int x = 4;  
    int y = 10;  
    if (4 < 10)  
        return 1  
    return 0  
}
```

- Это ж исходный код!

```
080483b4:  
80483b4: push    ebp  
80483b5: mov     ebp,esp  
80483b7: sub     esp,0x10  
80483ba: mov     DWORD PTR [ebp-0x4],0x4  
80483c1: mov     DWORD PTR [ebp-0x8],0xa  
80483c8: mov     eax,DWORD PTR [ebp-0x4]  
80483cb: cmp     eax,DWORD PTR [ebp-0x8]  
80483ce: jge     80483d7 <main+0x23>  
80483d0: mov     eax,0x1  
80483d5: jmp     80483dc <main+0x28>  
80483d7: mov     eax,0x0  
80483dc: leave  
80483dd: ret
```

5 минутное упражнение

- Понять что делает функция, написать её исходный код:

```
push    ebp
mov     ebp, esp
mov     eax, DWORD PTR [ebp+0xc]
mov     edx, DWORD PTR [ebp+0x8]
lea     eax, [edx+eax*1]
pop     ebp
ret
```

- Сколько локальных переменных, сколько параметров у функции?
- Подсказка: `lea eax, [edx+eax*1]` то же что и `eax = edx+eax`

Упражнение - Решение

- Просто сумма двух чисел.
- Компилятор использует `lea edx+eax` ради эффективности
- Мог бы просто использовать инструкцию `add`
- `eax` хранит возвращаемое значение
- Нет локальных переменных (нет `[ebp-x]`), только аргументы (`[ebp+x]`)

```
sum(int x, int y) {  
    return x + y;  
  
main(void) {  
    return sum(5, 7);  
}
```

Вопросы?

Игорь Черватюк

Александр Трифанов

Андрей Басарыгин

Москва, 2018

Demo time

Полезные ссылки

- <https://beginners.re/>
- <https://www.nostarch.com/idapro2.htm>

Вопросы?

Игорь Черватюк

Александр Трифанов

Андрей Басарыгин

Москва, 2018