Лекция 18

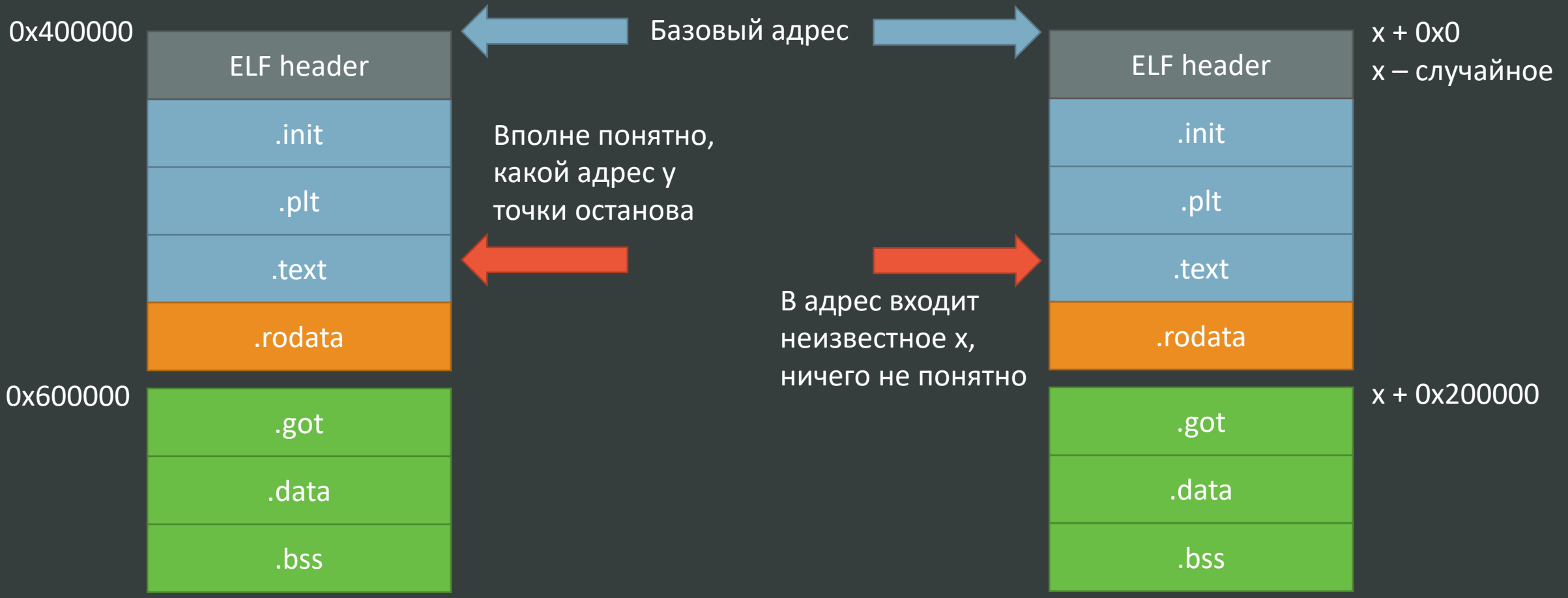
ЕЩЕ НЕМНОГО ПРО ОТЛАДКУ, ПРО АНТИОТЛАДКУ
А ТАКЖЕ ПРО МОДИФИКАЦИЮ МАШИННОГО КОДА

rev. 5

Нюансы отладки PIE исполняемых файлов

- PIE – Position-Independent Executable
  - Или позиционно-независимый исполняемый файл, мы сталкивались с этим термином на одной из прошлых лекций
- Главной особенностью этих исполняемых файлов является тот факт, что они не имеют постоянного адреса в виртуальной памяти (он может меняться от запуска к запуску)
  - В этом отношении они очень похожи на разделяемые библиотеки
- С установкой точек останова по адресу возникает очевидная проблема: если нет адресов, куда ставить точку останова?
  - С функциями, доступными по именам, таких проблем нет
- К сожалению, многие ОС (например Ubuntu начиная с 16.10) имеют PIE включенным по умолчанию

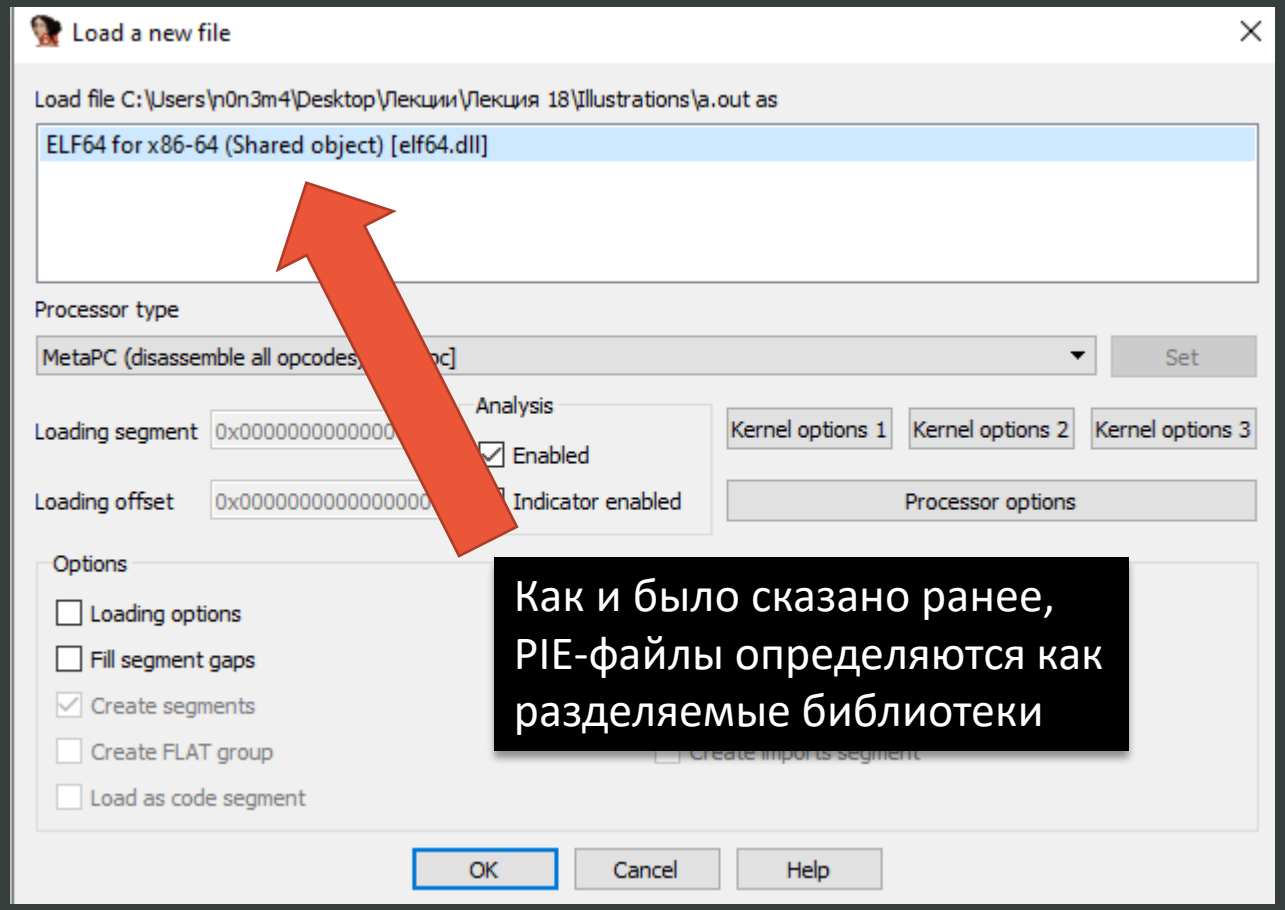
Нюансы отладки PIE исполняемых файлов



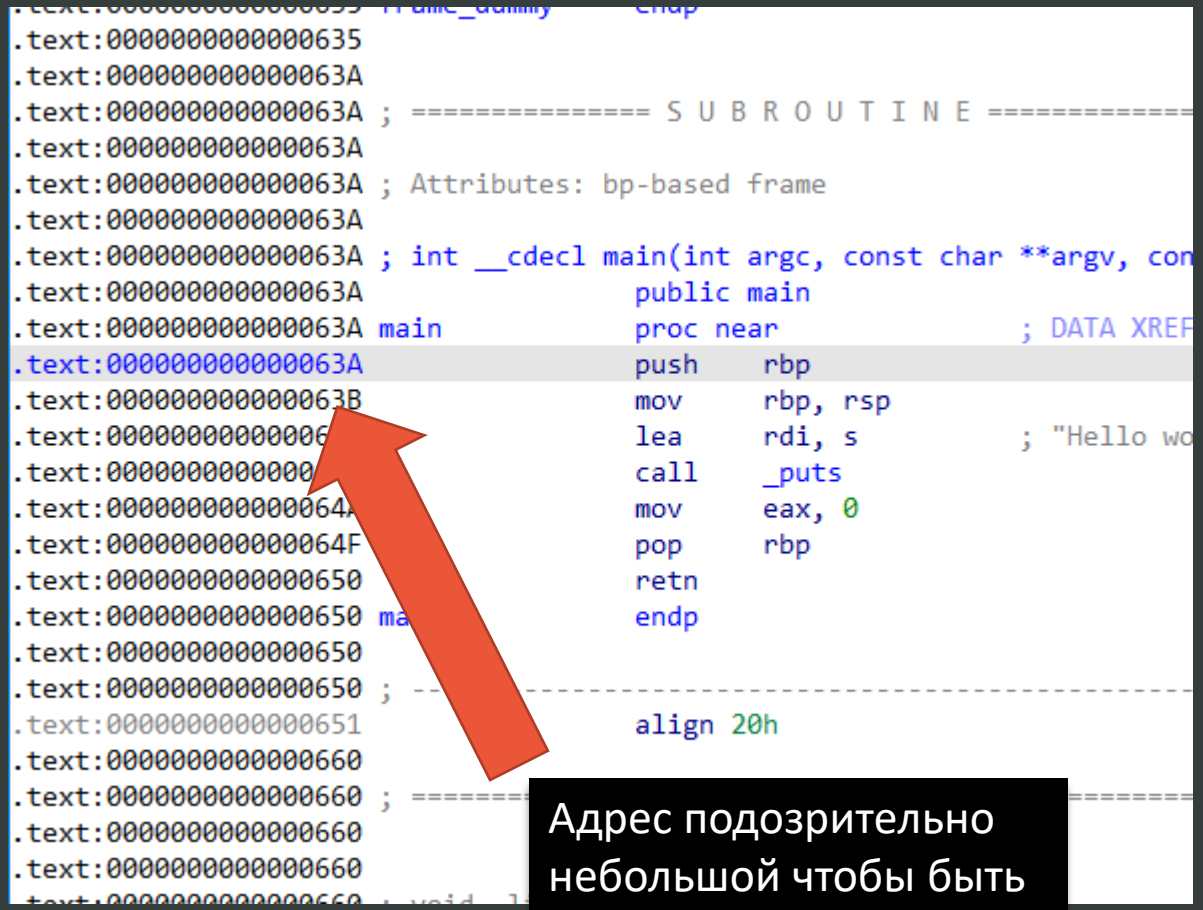
Нормальный исполняемый файл в памяти

Позиционно-независимый исполняемый файл в памяти

Нюансы отладки PIE исполняемых файлов



Как и было сказано ранее, PIE-файлы определяются как разделяемые библиотеки



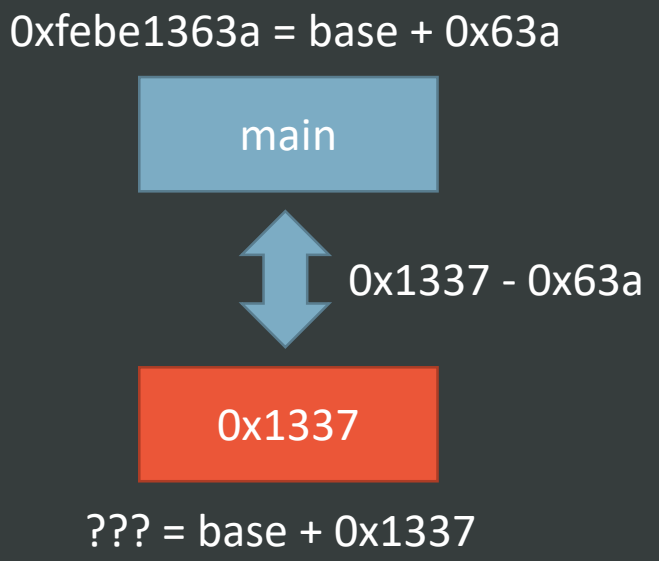
Адрес подозрительно небольшой чтобы быть реальным виртуальным адресом

Нюансы отладки PIE исполняемых файлов

- Поскольку GDB по умолчанию выключает рандомизацию адресного пространства, наш исполняемый файл каждый раз попадает своим началом на постоянный «случайный» адрес
  - В Ubuntu – 0x555555554000 (x86-64)
  - В WSL 1 – 0x8000000
- Отсюда возникает самый очевидный способ справиться с проблемой – просто прибавлять к желаемому адресу нужное значение руками
- У метода есть очевидный недостаток – нужно складывать числа руками (особенно 0x555555554000)
- Также метод не сработает в случае, если вы подключитесь к уже запущенному процессу (тогда базовый адрес придется смотреть через /proc/…/maps или "info proc mappings")

Нюансы отладки PIE исполняемых файлов

- Другой, чуть более простой вариант – найти точку опоры в лице какой-нибудь функции, у которой есть название, например main
- После этого адрес в PIE-файле можно рассчитать как разницу между нужным адресом и адресом известной функции в исполняемом файле, сложенную с реальным адресом этой функции
- Проще говоря, можно написать следующую команду:
  - break * (&main - 0x63a + <нужный_адрес>), где 0x63a – адрес main в исполняемом файле (можно посмотреть в IDA)
- Этот подход работает всегда (по крайней мере в свежих версиях GDB), но для него требуется какая-нибудь известная функция или переменная, за которую можно зацепиться



Нюансы отладки PIE исполняемых файлов

- Еще удобнее поставить точку останова можно при помощи ранее упомянутых плагинов для GDB
- В PEDA и pwndbg для этого можно использовать команду breakrva
  - Эта команда имеет вид breakrva <адрес> <файл> (файл – опциональный аргумент)
  - Адекватно работает эта команда только когда программа уже запущена
  - Для того, чтобы запустить программу, выполнив только первую инструкцию, можно использовать команду starti (это будет удобным местом для использования breakrva)
- В GEF для этого используется команда pie break, которая работает как обычный break, но с поддержкой PIE
  - В сочетании с ней нужно использовать pie run и pie attach, не очень удобно

Нюансы отладки PIE исполняемых файлов

```
(gdb) break * 0x645
Breakpoint 1 at 0x645
(gdb) r
Starting program: /tmp/test
Warning:
Cannot insert breakpoint 1.
Cannot access memory at
address 0x645
(gdb)
```

Без плагинов

```
gdb-peda$ starti
Starting program: /tmp/test

Program stopped.
...
gdb-peda$ breakrva 0x645
Breakpoint 1 at 0x8000645
gdb-peda$ c
Continuing.
...
Breakpoint 1,
0x0000000008000645 in main ()
gdb-peda$
```

PEDA / pwndbg

```
gef➤  pie break * 0x645
gef➤  pie run
Stopped due to shared library
event (no libraries added or
removed)

Breakpoint 1,
0x0000000008000645 in main ()
[+] base address 0x8000000
...
[#0] Id 1, Name: "a.out",
stopped 0x8000645 in main (),
reason: BREAKPOINT
...
gef➤
```

GEF

Противодействие дизассемблированию

Противодействие дизассемблированию

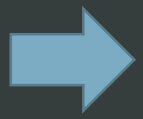
- Имеет главной целью запутывание кода, получаемого аналитиком в дизассемблере
- Может осуществляться не только при помощи ранее рассмотренных техник, связанных, прежде всего, с динамической модификацией кода
- Мы рассмотрим некоторые из многочисленных способов:
  - Переход на переменный адрес
  - Переход через RET, переполнение стека
  - Неоднозначное дизассемблирование (x86)

Переход на переменный адрес

- Подход, похожий на самомодифицирующийся код в самом бедном исполнении
  - Тут «самомодифицируется» только адрес перехода, а код остается как есть
- Суть подхода состоит в том, что вместо конструкций вида jmp function вы используете jmp RAX, где в RAX лежит динамически рассчитанный адрес функции, равный в конечном итоге тому же function (но дизассемблер об этом не знает)
- Этот подход может быть реализован случайно в рамках таблицы виртуальных функций C++
- Главным способом противодействия подходу является отладка (в момент исполнения адрес становится известен)
  - В простых случаях может быть эффективен и ручной анализ (например, можно пристально посмотреть на конструктор объекта в случае виртуальных функций)
  - Также можно попробовать плагин https://github.com/murx-/devi/ который трейсит все за вас при помощи Frida (но немного кривоват)

Переход на переменный адрес

```
class Parent {
  public:
  virtual void function() {
    cout << "Parent Function\n";
  }
};
class Child: public Parent {
  public:
  void function() {
    cout << "Child Function\n";
  }
};
int main() {
  Parent *x = new Parent();
  x->function();
  Parent *y = new Child();
  y->function();
}
```



IDA View-A | Pseudocode-A

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  _QWORD *v3; // rax
  void (__fastcall ***v4)(_QWORD); // rbx
  _QWORD *v5; // rax
  void (__fastcall ***v6)(_QWORD); // rbx

  v3 = (_QWORD *)operator new(8uLL);
  v4 = (void (__fastcall ***)(_QWORD))v3;
  *v3 = 0LL;
  sub_AD8(v3, a2);
  (**v4)(v4);
  v5 = (_QWORD *)operator new(8uLL);
  v6 = (void (__fastcall ***)(_QWORD))v5;
  *v5 = 0LL;
  sub_AF2(v5);
  (**v6)(v6);
  return 0LL;
}
```

Вызов Parent::function

Вызов Child::function

Переход через RET

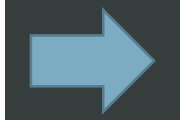
- До этого декомпилятор / дизассемблер по крайней мере понимал, что происходит вызов функции (правда, мог не знать, какой в ней код или какой у нее адрес)
- В рамках данного подхода появляется возможность осуществлять переходы, при которых дизассемблер в принципе не будет отображать код как переход куда-либо
- Для этого можно использовать конструкцию из PUSH и RET
  - В этом случае вы положите на стек «адрес возврата», после чего сразу же вернетесь на него, осуществив переход
- Код такого рода также используется в Retpoline – технологии защиты от уязвимостей Spectre, поэтому такое можно увидеть и в нормальных программах
  - Правда там используется конструкция из CALL; CALL; MOV [RSP], addr; RET, но суть та же
- Главный способ противодействия – знание того, что такие переходы бывают

Переход через RET

```
int somefunction()
{
  printf("I'm a function\n");
}

int callfun(void* addr);
asm("   callfun:\n\
        push %rdi\n\
        ret\n\
");

int main()
{
  callfun(&somefunction);
}
```



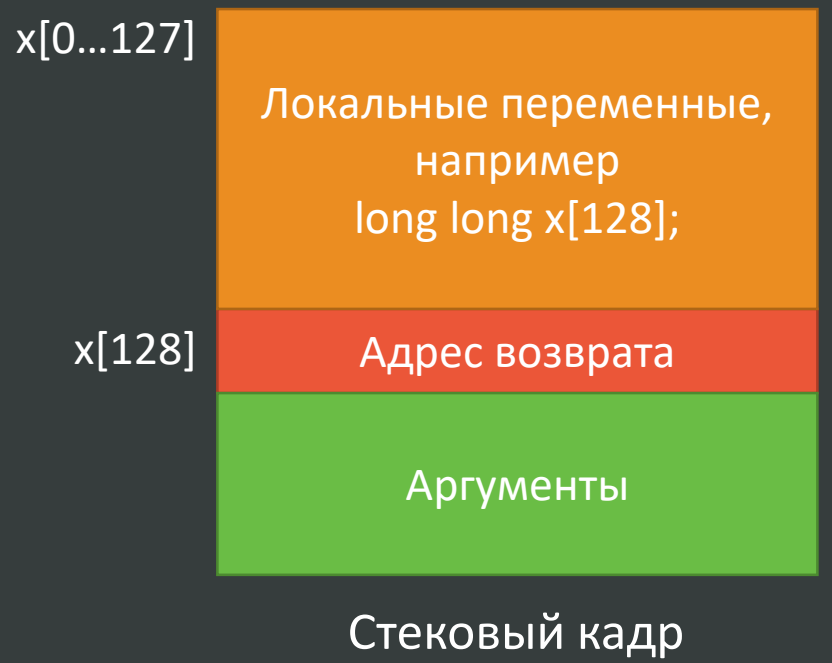
IDA View-A | Pseudocode-B

```
__int64 __fastcall main(__int64 a1, char **a2, char
{
  sub_64D((__int64)sub_63A);
  return 0LL;
}
```

```
void __fastcall sub_64D(__int64 a1)
{
  __asm { retn }
}
```

Переход через переполнение стека

- Мы подробнее изучим переполнение стека в следующем разделе, однако со структурой стека мы уже знакомы достаточно хорошо, чтобы понять этот подход
- Что если не класть адрес возврата на стек прямо перед возвратом, а подредактировать адрес возврата текущей функции?
  - Это можно сделать невзначай обратившись за пределы массива, лежащего на стеке
- Стек после таких фокусов будет в печальном состоянии, но это решаемая проблема
- Главный способ противодействия – отладка



Стековый кадр

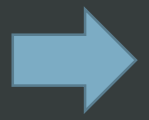
Переход через переполнение стека

```c
// gcc -s retovflw.c -o retovflw
// -fomit-frame-pointer -fno-stack-protector

int somefunction() {
  // Stack is not aligned,
  // got to use asm to align it
  asm("push $0");
  printf("I'm function\n");
  // Stack is broken, just exit
  exit(0);
}

int jumpto(void* addr) {
  long x[0];
  x[1] = addr;
}

int main() {
  jumpto(&somefunction);
}
```



IDA View-A | Pseudocode-A

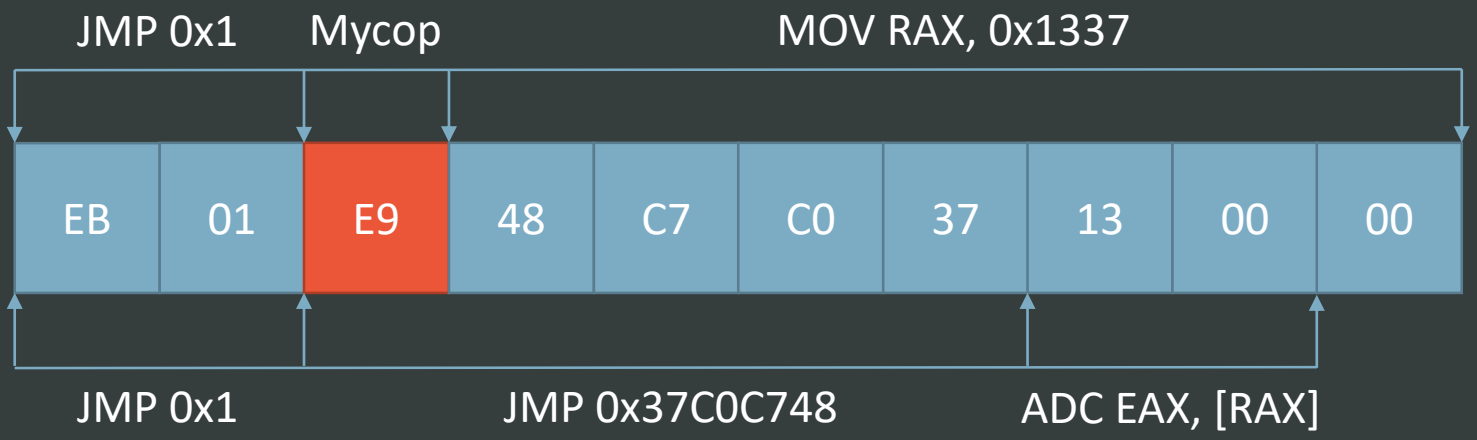
```
1 __int64 __fastcall main(__int64 a1, char **a2, char **a3)
2 {
3   sub_6A6((__int64)sub_68A);
4   return 0LL;
5 }

__int64 __fastcall sub_6A6(__int64 a1)
{
  return a1;
}
```

Неоднозначное дизассемблирование (x86)

- Поскольку у команд x86 переменная длина, может возникнуть вопрос: где именно заканчивается текущая инструкция и начинается следующая?
- Соответственно, иногда дизассемблеры могут ошибиться с определением начала инструкции
  - Особенно, если им с этим специально помочь, добавив лишних байтов перед исполняемым кодом
- Со стороны наблюдателя такой способ защиты кода может выглядеть как прыжок в середину процессорной инструкции
- С этим способом защиты можно бороться при помощи отладчика (в тяжелых случаях), при помощи IDA (с помощью кнопок C / D) или просто сменив дизассемблер (например на Ghidra)

Неоднозначное дизассемблирование (x86)



- Продвинутый дизассемблер (IDA, Ghidra) сообразит, что в данном случае байт 0xE9 вообще не исполняется и просто пропустит его при дизассемблировании
- Более простые дизассемблеры (objdump, GDB), в свою очередь, декодируют инструкции линейно и поведутся на этот трюк

Неоднозначное дизассемблирование (x86)

IDA

```
seg000:0000000000000000                 assume es:nothing, ss:nothing, ds:nothin
seg000:0000000000000000                 jmp     short loc_3
seg000:0000000000000000 ; ---------------------------------------------------------
seg000:0000000000000002                 db 0E9h
seg000:0000000000000003 ; ---------------------------------------------------------
seg000:0000000000000003
seg000:0000000000000003 loc_3:                                  ; CODE XREF: seg
seg000:0000000000000003                 mov     rax, 1337h
seg000:0000000000000003 seg000          ends
```

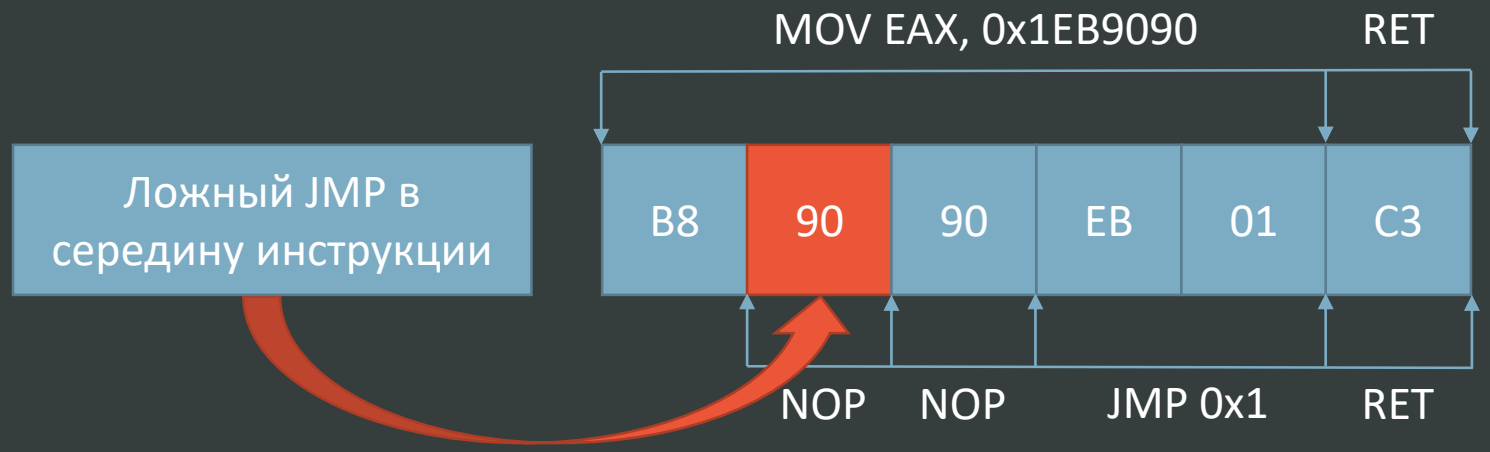
Objdump

```
Disassembly of section .data:

0000000000000000 <.data>:
   0:   eb 01                   jmp    0x3
   2:   e9 48 c7 c0 37          jmpq   0x37c0c74f
   7:   13 00                   adc    (%rax),%eax
        ...
```

Неоднозначное дизассемблирование (x86)

- Впрочем, иногда может возникнуть обратная ситуация: слишком умный дизассемблер может проследовать по ссылкам на середины инструкций, в то время как настоящий адрес перехода будет вычислен динамически
  - В первую очередь от этого страдает IDA, хотя с Ghidra такое тоже бывает
- Лечится это нажатием кнопок C / D (в IDA) в нужных местах
  - Как ни странно, в Ghidra кнопки те же, но наоборот (D там значит не Data, а Disassemble)

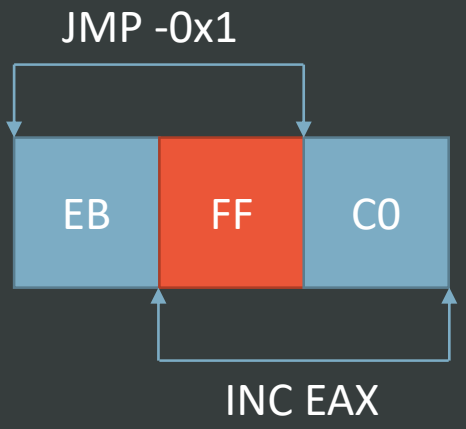


Неоднозначное дизассемблирование (x86)

```
int faketrigger();
asm("faketrigger:\n\
     jmp .+3\n\
    ");
int fun();
asm("fun:\n\
     mov $0x01EB9090, %eax\n\
     ret\n\
    ");
static int zero;
int main()
{
    if (1 == zero)
        faketrigger();
    else
        printf("Fun: %x\n", fun());
}
```

```
.text:000000000000064A sub_64A         proc near
.text:000000000000064A                 jmp     short loc_64D
.text:000000000000064A sub_64A         endp
.text:000000000000064A
.text:000000000000064A ; ---------------------------------------
.text:000000000000064C byte_64C        db 0B8h
.text:000000000000064D ; ---------------------------------------
.text:000000000000064D
.text:000000000000064D loc_64D:
.text:000000000000064D                 nop
.text:000000000000064E                 nop
.text:000000000000064F                 jmp     short main
.text:0000000000000651 ; ---------------------------------------
.text:0000000000000651                 retn
.text:0000000000000652
.text:0000000000000652 ; =============== S U B R O U T I N E =
.text:0000000000000652
.text:0000000000000652 ; Attributes: bp-based frame
.text:0000000000000652
.text:0000000000000652 ; int __fastcall main(int, char **, cha
.text:0000000000000652 main            proc near
.text:0000000000000652
.text:0000000000000652                 push    rbp
```

Невозможное дизассемблирование (x86)



Примечание:
JMP -0x1 переходит на один байт назад относительно следующей инструкции, то есть на инструкцию INC EAX

- Расширенный вариант предыдущего трюка: один байт вообще может принадлежать сразу нескольким инструкциям
- Такой код вообще невозможно нормально дизассемблировать
  - Впрочем, по нему все еще можно пройтись отладчиком
- В качестве метода борьбы можно поредактировать машинный код и переписать его по-нормальному (в этом случае – заменить ненужный JMP на NOP)

Время задач

Горе от ума

Категория: Lesson 18 / Antidebug + Patching

Решивших: 0

Время: 00:00:02

- Доступ к задачам можно получить как всегда на nsuctf.ru

Противодействие отладке

Противодействие отладке

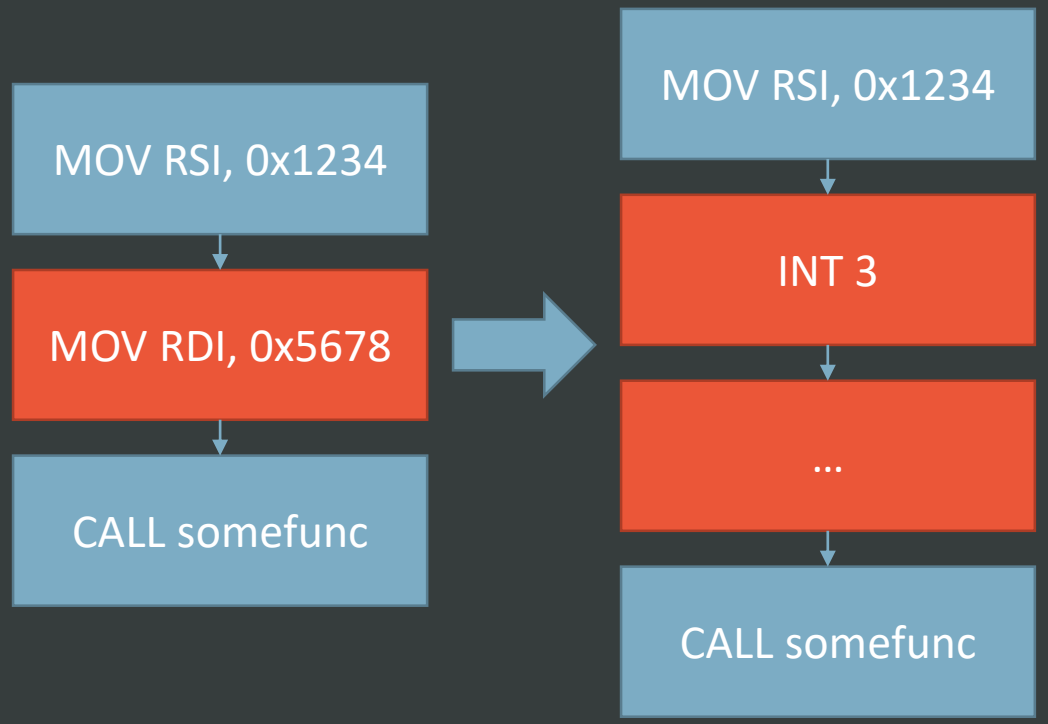
- Нетрудно видеть, что отладка является достаточно мощным инструментом, чтобы авторы программного обеспечения захотели от нее избавиться
- Как показывает практика, существует множество возможностей это сделать, мы рассмотрим следующие из них:
  - Использование официальных API
  - Противодействие установке точек останова и сопутствующему сигналу SIGTRAP
  - Поиск процессов отладчиков
  - Измерение времен исполнения
  - Подключение второго отладчика
  - Проверка ASLR
- Неплохую подборку методов для Windows можно найти здесь: https://anti-debug.checkpoint.com/ (перечисленные методы там тоже есть)

Использование официальных API

- Иногда достаточно просто спросить операционную систему, не подвергается ли процесс отладке
- В Windows это функция IsDebuggerPresent (и она там далеко не одна)
- В Linux это файл /proc/<pid>/status, в котором присутствует поле TracerPid
  - В нормальном состоянии (без отладки), в этом поле указан 0
  - В случае, если процесс подвергается отладке, в этом поле указан PID отладчика
- Самый радикальный способ справиться с такими проверками – написать драйвер или модуль ядра для операционной системы, который будет перехватывать соответствующий API
  - Для Linux такой модуль ядра доступен по ссылке https://github.com/LWSS/TracerHid

Противодействие установке точек останова

- Для начала рассмотрим, как вообще ставятся обычные точки останова, которые доступны в GDB по команде break
- Для того, чтобы поставить точку останова, GDB (на x86) заменяет инструкцию, на которую вы ее ставите, на INT 3 (кодируется одним байтом 0xCC)
  - При этом он, конечно, запоминает оригинальную инструкцию
- В результате вызова инструкции INT 3 ядро Linux генерирует сигнал SIGTRAP, который перехватывается и обрабатывается GDB



Противодействие установке точек останова

- Отсюда следует сразу два эффективных способа противодействия:
  - Проверка целостности исполняемого кода: установленные 0xCC меняют код, что достаточно легко заметить
  - Отправка и обработка сигнала SIGTRAP внутри программы в процессе нормального исполнения: в случае если сигнал SIGTRAP где-то потеряется можно начинать бить тревогу

Проверка целостности кода

- Можно сделать поведение кода (например, какие-то ключи шифрования) зависимым от его содержимого
  - Это более надежный и предпочтительный способ, хотя и более сложный и зависимый от конкретного компилятора
- Можно просто поискать в коде лишние 0xCC и завершить выполнение при их обнаружении
  - Это значительно проще, но также и легче в поиске
- В обоих случаях наиболее удобным способом получения доступа к коду является приведение его к массиву байтов
  - Поскольку в большинстве нормальных операционных систем (не Android) код доступен и для исполнения и для чтения, это не будет проблемой

Проверка целостности кода

```
static char int3 = 0xCC;
int funend();
int fun(int x)
{
  for (char* i=fun;i<funend;i++)
    if (i[0] == int3)
    {
      printf("BREAKPOINT DETECTED\n");
      return 0;
    }
  int secret = 0x1337;
  secret ^= 0xBEEF;
  return x == secret;
}
int funend(){}
```

```
Reading symbols from ./nobreak...(no
debugging symbols found)...done.
(gdb) break fun
Breakpoint 1 at 0x78e
(gdb) r
Starting program: /tmp/nobreak
Enter secret number: 123

Breakpoint 1, 0x000000000800078e in fun ()
(gdb) c
Continuing.
BREAKPOINT DETECTED BYE

(gdb)
```

Аппаратные точки останова

- Для решения этой проблемы можно использовать аппаратные точки останова
- Такие точки останова можно установить при помощи команды hbreak в GDB
  - Она используется так же как обычная команда break
- Их количество ограничено примерно четырьмя (на x86), однако они не требуют модификации кода, что позволяет обходить вышеупомянутые проверки
- GDB может не захотеть ставить такие точки останова до запуска программы
  - Тогда рекомендуется воспользоваться инструкцией starti, чтобы выполнить первую инструкцию программы, а уже затем ставить аппаратные точки останова
- Кстати, если когда-нибудь вам придется отлаживать ядро, нужно тоже использовать hbreak
- Такие точки останова могут быть обнаружены в Windows через чтение регистров DR0…DR3 из текущего контекста

Фокусы с SIGTRAP

- Можно добавить в программу обработчик сигналов SIGTRAP, который при исполнении будет менять какую-либо переменную
- После этого в произвольном месте программы добавить код, вызывающий сигнал SIGTRAP (INT 3) и убедиться, что переменная изменилась
- Дело в том, что по умолчанию GDB предлагает игнорировать полученные сигналы SIGTRAP и не передавать их программе
  - В этом случае сигнал SIGTRAP молча потеряется и переменная не изменится

Фокусы с SIGTRAP

```c
static int handled = 0;
void trap_handler(int num)
{
  handled = 1;
}

int main()
{
  signal(SIGTRAP, trap_handler);
  asm("int $3");
  if (!handled)
    printf("DEBUGGER DETECTED\n");
  else
    printf("All ok\n");
}
```

```
Reading symbols from ./notrap...(no debugging
symbols found)...done.
(gdb) r
Starting program: /tmp/notrap

Program received signal SIGTRAP,
Trace/breakpoint trap.
0x00000000080006b4 in main ()
(gdb) c
Continuing.
DEBUGGER DETECTED
[Inferior 1 (process 756) exited normally]
(gdb)
```

Обработка сигналов в GDB

- GDB предлагает команду handle для настройки поведения при получении сигналов
  - Посмотреть текущее состояние можно командой "info handle"
  - Можно заметить, что SIGTRAP является одним из немногих сигналов, которые не передаются дальше программе
- Команда handle используется следующим образом: handle <имя_сигнала> <поведение>
- Параметр «поведение» может содержать следующие слова:
  - stop / nostop – должен ли GDB останавливаться при получении сигнала
  - print / noprint – должен ли GDB печатать сообщение о получении сигнала
  - pass / nopass – должен ли GDB передать сигнал дальше

Обработка сигналов и SIGTRAP

- Увы, GDB использует SIGTRAP для работы (о чем он предупреждает) и по какой-то причине при установке обработчика в "pass" подопытная программа не переходит в обработчик сигнала, а падает
- Однако, при получении сигнала можно использовать команду "signal SIGTRAP", которая заново сгенерирует сигнал и направит его в программу
  - Конечно, программа может проверить и этот сигнал на подлинность, но может и не проверить

```
Reading symbols from ./notrap...(no
debugging symbols found)...done.
(gdb) r
Starting program: /tmp/notrap

Program received signal SIGTRAP,
Trace/breakpoint trap.
0x0000000080006b4 in main ()
(gdb) signal SIGTRAP
Continuing with signal SIGTRAP.
All ok
[Inferior 1 (process 838) exited normally]
(gdb)
```

Поиск процессов отладчиков

- Не слишком умный способ, основан на обычном просмотре списка процессов
  - Нормальные люди им не пользуются из-за большого числа ложноположительных результатов
- Можно встретить в различных античит-системах
  - Исследование типичного представителя доступно по ссылке https://habr.com/ru/post/483068/ (TL;DR !contains(window_title_report.window_title, "PUBG AIM"))
- Лечится сборкой отладчика из исходного кода с заменой всех имен
  - Впрочем, в некоторых особо тяжелых случаях достаточно просто переименовать отладчик в notepad.exe / gedit
  - От особо любопытных глаз можно собрать отладчик с использованием пакера или обфускатора

Измерение времен исполнения

- Обычные отладчики (без виртуализации) не замораживают время при исполнении отлаживаемой программы
- Поэтому, если кто-нибудь решит посидеть на breakpoint (пусть даже и аппаратном), это неминуемо отразится на времени исполнения
- Таким образом, измеряя время можно делать вывод о наличии отладчика
  - Кстати говоря, как можно было заметить при решении задач, strace тоже может здорово тормозить исполнение, когда в программе много системных вызовов, так что его тоже можно так обнаружить
- Измерять время можно как при помощи системных вызовов, так и инструкции RDTSC
- Единственным надежным способом противодействия (без виртуализации) является поиск таких моментов в программе и их удаление

Подключение второго отладчика

- Как в Windows так и в Linux процесс может отлаживаться только одним отладчиком одновременно
- Отсюда следуют два способа защиты:
  - Запуск второго процесса, который будет отлаживать первый (и, возможно, как-то менять его код)
  - Проверка принципиальной возможности подключения отладчика, без реальной отладки
- Первый подход не в пример более эффективен, его можно встретить в некоторых играх на платформе Android, с ним в общем случае можно справиться только вручную, поскольку он может быть смешан с модификацией кода
- В Linux, однако, гораздо более популярным (вероятно, из-за простоты реализации) является второй подход, поэтому рассмотрим его

Проверка возможности подключения отладчика

- В Linux для отладки используется системный вызов ptrace
- У него есть команда PTRACE_TRACEME, позволяющая процессу попросить об отладке себя (обычно это делается перед запуском другого процесса)
  - После выполнения этой команды каждый вызов execve будет вызывать ожидание подключения отладчика, как и каждый полученный сигнал
- В случае, если отладчик уже подключен, этот системный вызов закончится неудачей
- Для того, чтобы обойти эту проверку, достаточно перехватить все вызовы функции ptrace / системного вызова ptrace и заменить возвращаемое значение на 0 (успех)
  - В PEDA для автоматического перехвата такого рода есть команда unptrace, ее лучше запускать когда программа уже запущена (после starti)

Проверка возможности подключения отладчика

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/ptrace.h>

int main()
{
  if (ptrace(PTRACE_TRACEME, 0, 1, 0) ==
-1)
    printf("DEBUGGER DETECTED\n");
  else
    printf("All ok\n");
}
```

```
(gdb) r
Starting program: /tmp/notrace
DEBUGGER DETECTED
(gdb)
```

```
gdb-peda$ starti
...
gdb-peda$ unptrace
Breakpoint 1 at 0x8000560
'ptrace' deactivated
...
gdb-peda$ c
Continuing.
All ok
gdb-peda$
```

Проверка ASLR

- Поскольку отладчик по умолчанию выключает ASLR (рандомизацию адресного пространства), можно проверять, что этот механизм не выключен
  - Это особенно применимо для PIE-исполняемых файлов
- Самый простой способ это сделать – взять указатель от какой-нибудь функции из программы и сравнить его с типичными для Linux значениями 0x555555554000 и 0x8000000
- Самый простой способ противодействия такой антиотладке – включить рандомизацию обратно :)
  - Отлаживать станет не очень удобно, но хоть как-то
  - Сделать это можно командой "set disable-randomization off"

Проверка ASLR

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

int main()
{
  uintptr_t ptr = (uintptr_t)&main;
  if ((ptr >> 20) == 0x5555555 || (ptr
>> 20) == 0x80)
    printf("DEBUGGER DETECTED\n");
  else
    printf("All ok\n");
}
```

```
(gdb) r
Starting program: /tmp/noaslr
DEBUGGER DETECTED
[Inferior 1 (process 345) exited normally]
(gdb)
```

```
(gdb) set disable-randomization off
(gdb) r
Starting program: /tmp/noaslr
All ok
[Inferior 1 (process 345) exited normally]
(gdb)
```

Модификация машинного кода

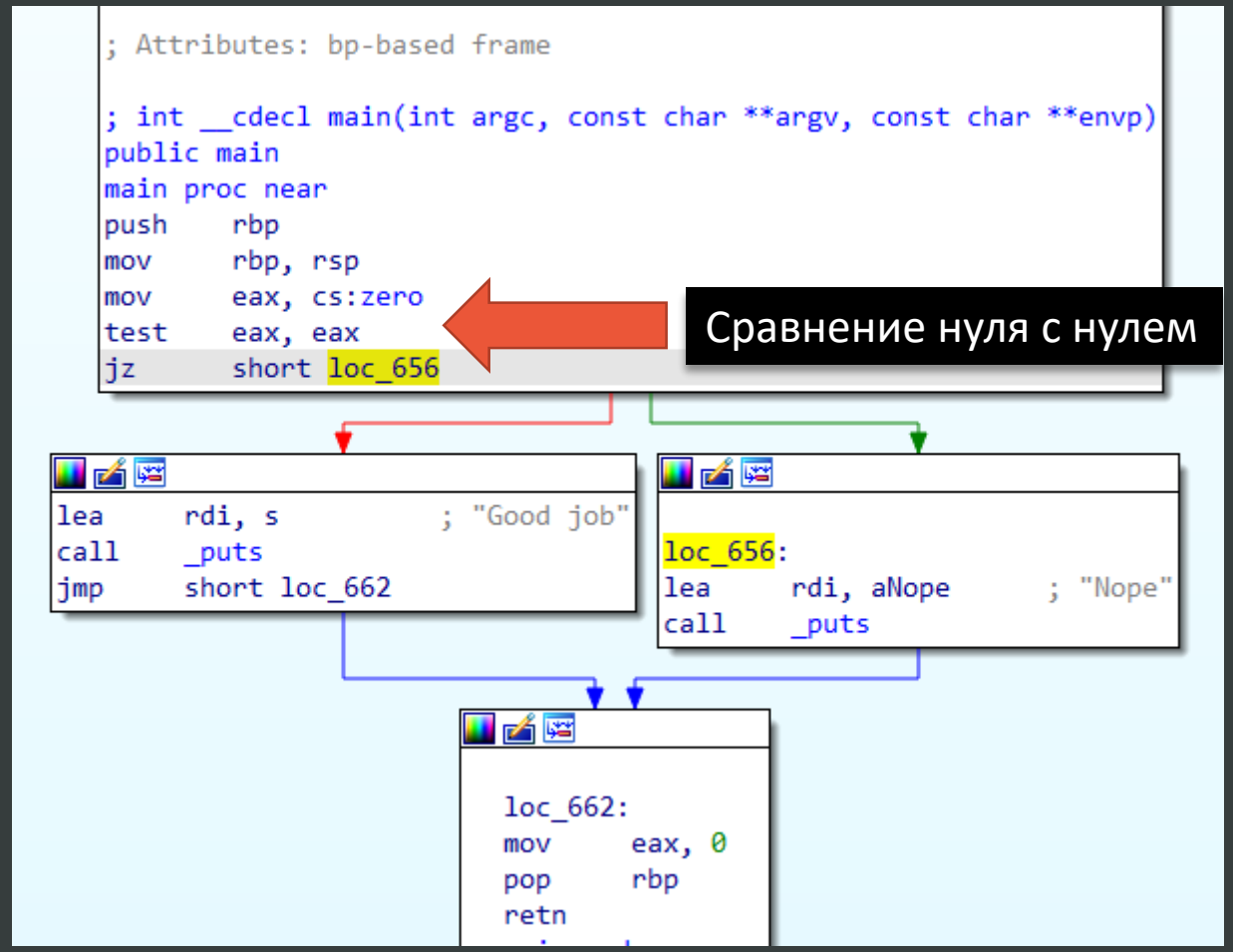
Модификация машинного кода

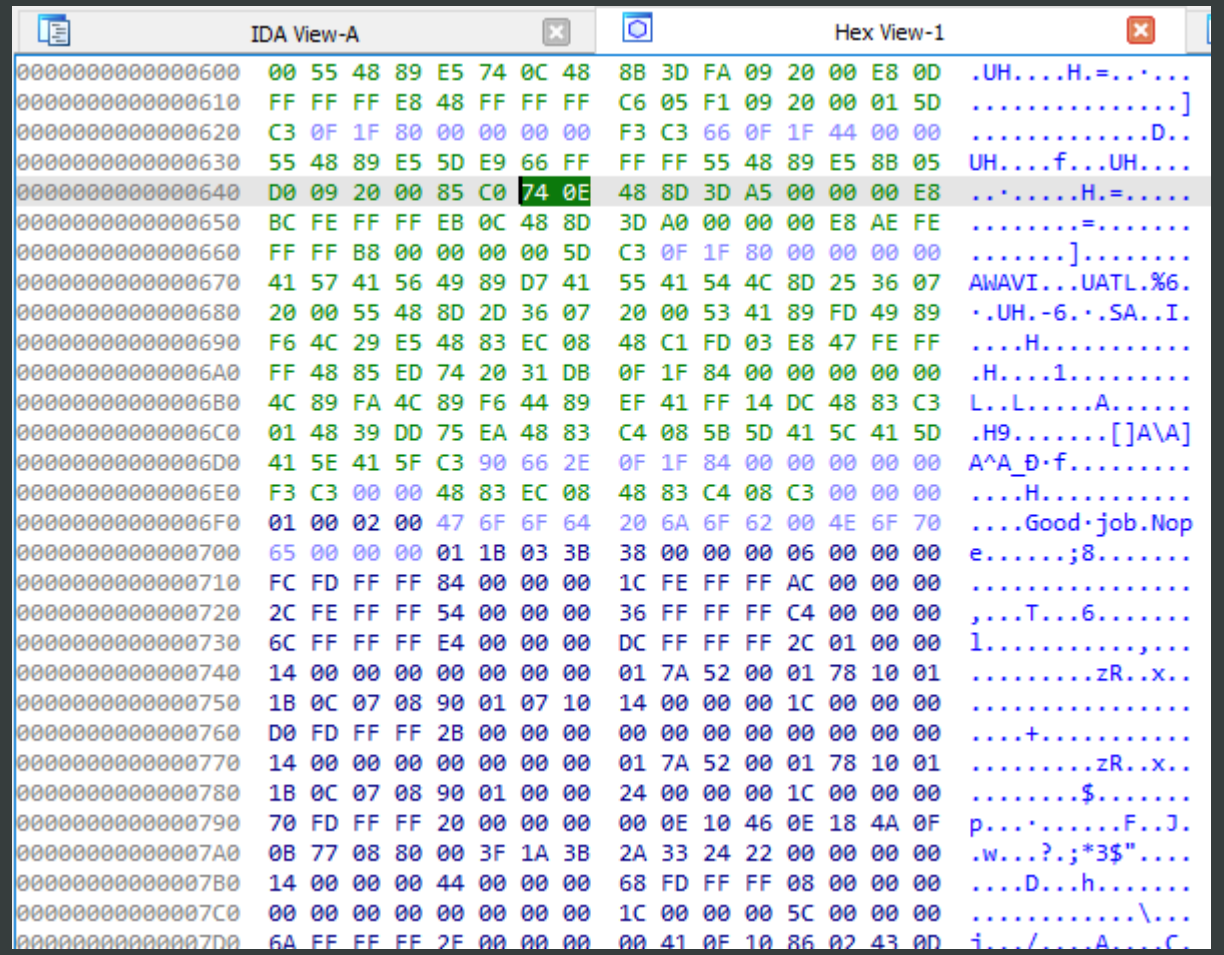
- Иногда возникает необходимость вынести из исполняемого файла лишнее
  - Например антиотладочные проверки
- Для этого было бы неплохо научиться редактировать исполняемые файлы
- Мы рассмотрим следующие способы редактирования исполняемых файлов:
  - Шестнадцатеричным редактором
  - При помощи плагина для IDA
- Осторожно, любой способ проверки целостности исполняемого файла может испортить вам все веселье, поэтому модификация исполняемого файла это последняя мера
  - Также она применима если вы целиком понимаете как работает файл, но его антиотладка вас утомляет

Редактирование файлов HEX-редактором

- Поскольку машинный код состоит из байтов, самой очевидной идеей является редактирование его именно в таком представлении
- Для этого вам понадобится узнать, как выглядит в машинном представлении код, который вы заменяете, а также код, на который вы хотите его заменить
- С первым вам очень поможет IDA – при переходе в Hex View она по умолчанию подсвечивает текущую выделенную в IDA View инструкцию
  - Ghidra тоже так умеет, достаточно нажать кнопку "Display bytes", инструкцию правда целиком не подсвечивает, но ее машинный код в окне дизассемблера и так видно

Редактирование файлов HEX-редактором



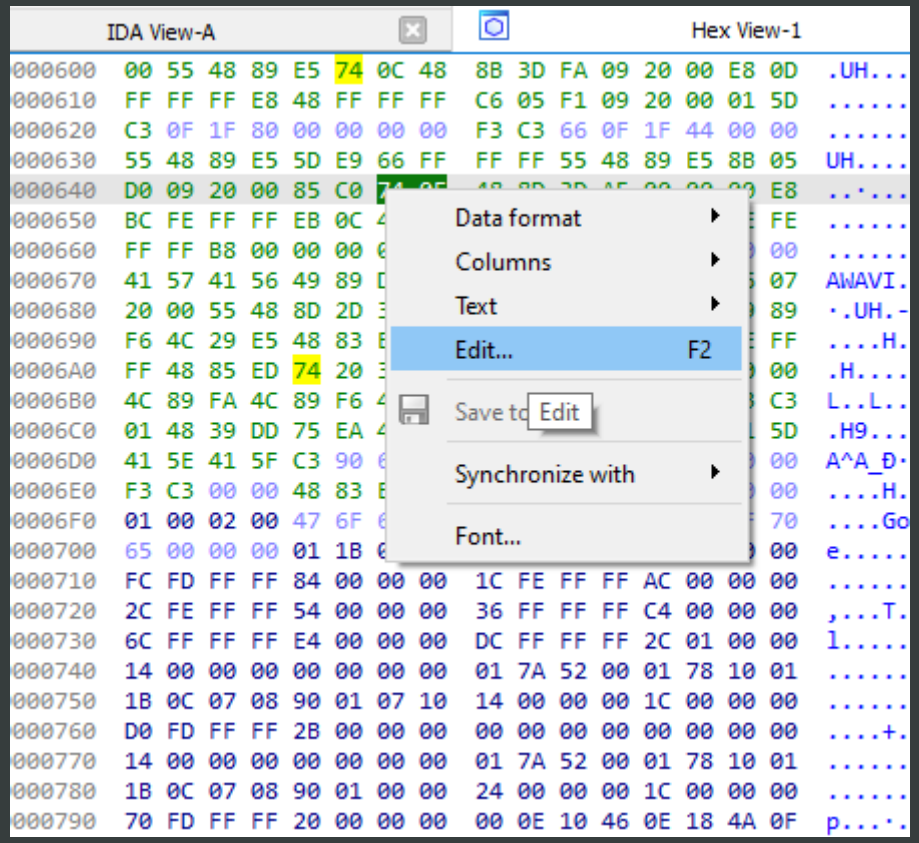


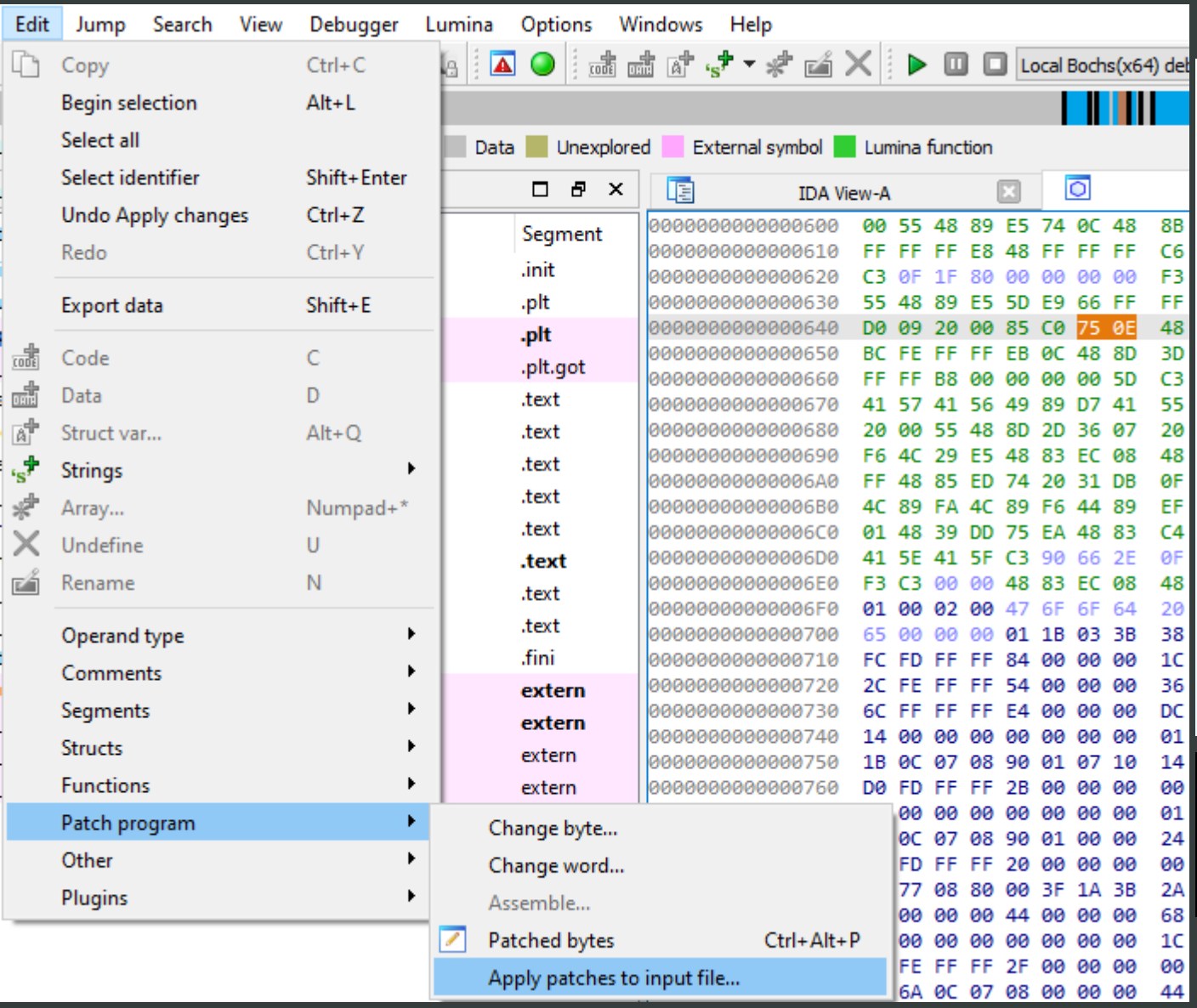
Редактирование файлов HEX-редактором

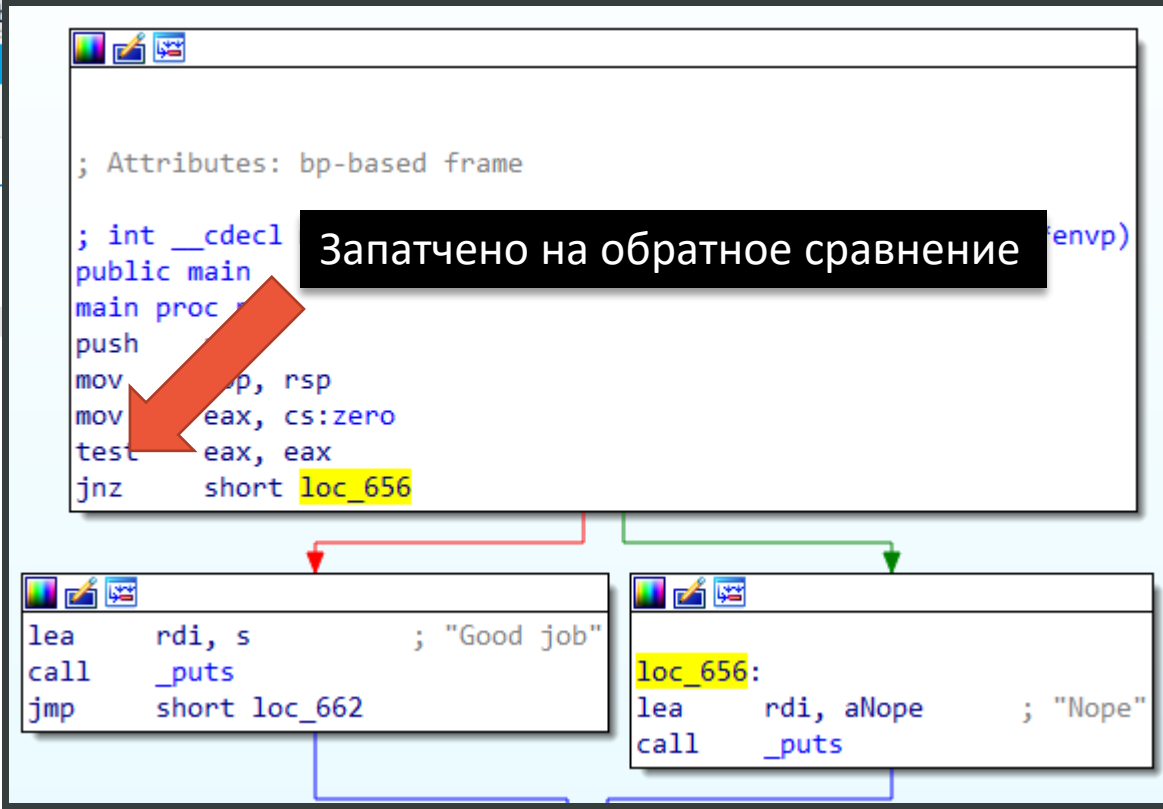
- Теперь, когда мы знаем, что за инструкцию мы хотим заменить, нужно выяснить, на что ее менять
- Один из вариантов – собрать свой код при помощи онлайн-сервиса https://defuse.ca/online-x86-assembler.htm, он выведет машинное представление вашего кода
- Однако, если вы хотите совсем немного подредактировать код (например заменить JZ на JNZ) – вам может пригодиться таблица опкодов (кодов операций), например http://ref.x86asm.net/coder64.html
  - Допустим, мы хотим заменить JZ из примера выше на JNZ, находим JZ в таблице опкодов и выясняем, что для него первым байтом является 0x74 (как мы и видели в HEX-представлении), а для JNZ – 0x75, соответственно достаточно просто заменить первый байт с 74 на 75

Редактирование файлов HEX-редактором

- В IDA чтобы редактировать HEX-представление, нужно нажать ПКМ – Edit или F2
  - После редактирования нужно обязательно нажать ПКМ – Apply (F2)
- По умолчанию редактируется только содержимое базы данных IDA (изменения отобразятся в IDA, но не в программе), чтобы сохранить изменения в программе нужно нажать Edit → Patch program → Apply patches to input file (затем нажать OK)
  - Грабли: если вы решите отменить уже внесенные изменения, эта отмена не будет считаться за «патч» и они не отменятся. Храните бэкапы
  - Опция "restore original bytes" также работает очень оригинально – она применяет обратный текущим изменениям патч (а не использует резервные копии)








```
n0n3m4@pc:/tmp$ ./patchme
Good job
```

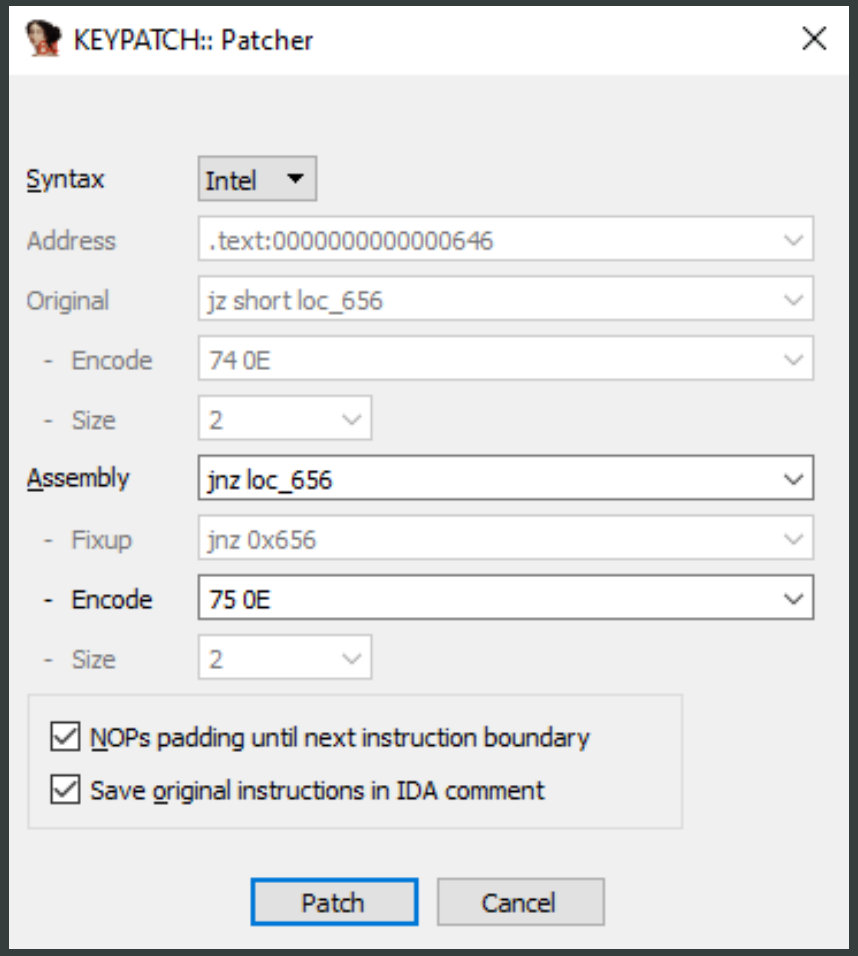
Редактирование файлов HEX-редактором

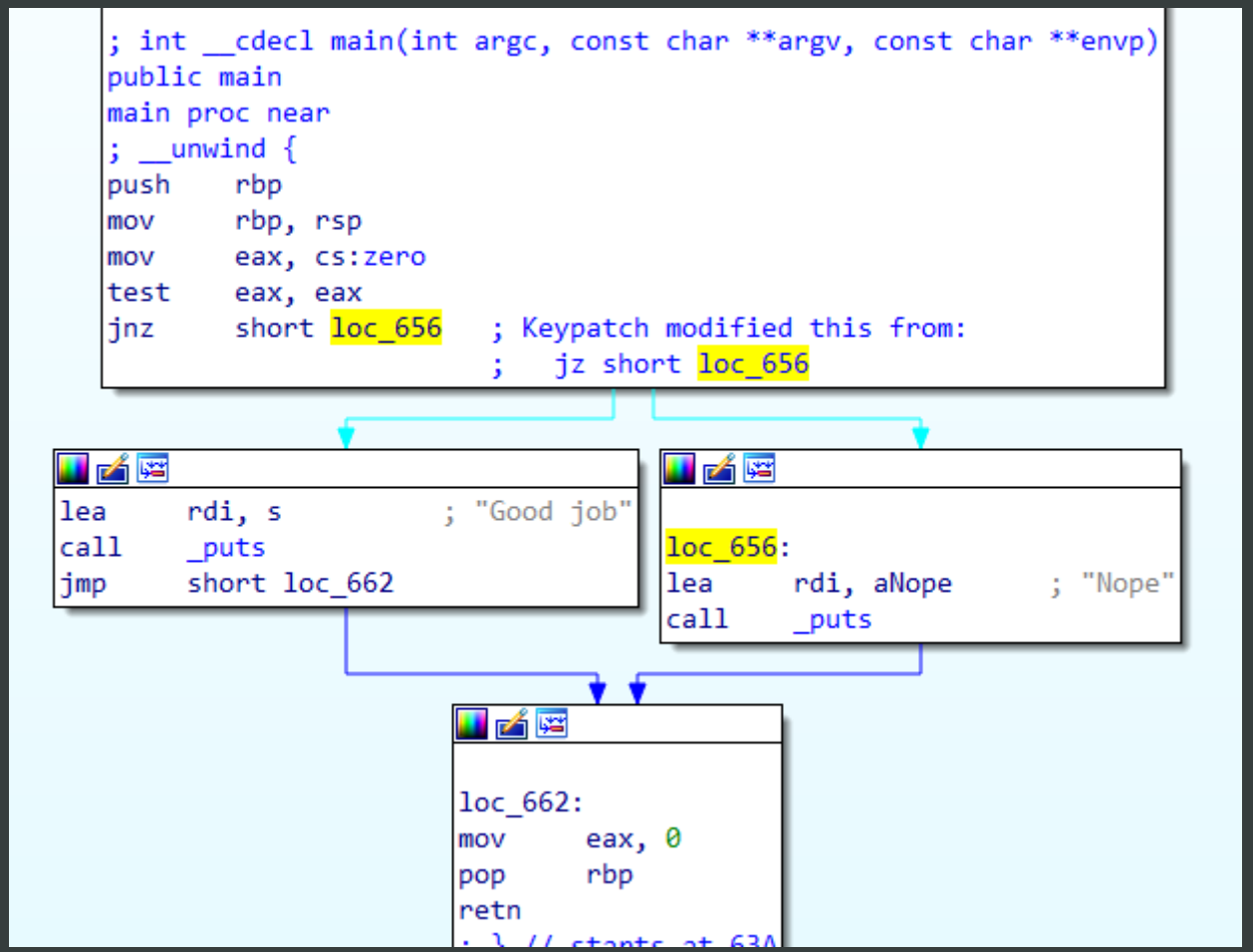
- Если вам нужно просто избавиться от какого-нибудь кода (будь это переход или что-то еще), вы можете просто забить его NOPами
  - NOP кодируется одним байтом 0x90, что очень удобно
- Если ваша инструкция получилась короче, чем та, которая была, вы также можете добавить в ее конец NOP
- Помните, что вы не сможете записать больше кода, чем уже было в программе, не пожертвовав соседними инструкциями
  - Это решаемый вопрос, но нетривиальными способами

Редактирование файлов плагином для IDA

- У полной версии IDA (не демо) есть поддержка IDAPython, что позволяет устанавливать в нее плагины
- Существует в том числе и плагин для редактирования кода – Keypatch
  - Его можно скачать по ссылке https://github.com/keystone-engine/keypatch, инструкции по установке представлены там же (только качайте код из репы, а не из вкладки Releases, он там старый)
- Этот плагин позволяет редактировать текущую инструкцию в виде кода на языке ассемблера
  - Он активируется при помощи ПКМ → Keypatch → Patcher или Ctrl-Alt-K
- После редактирования патчи также нужно применять к исходному файлу при помощи Edit → Patch program → Apply patches to input file (здесь все так же как и с ручным редактированием)

Редактирование файлов плагином для IDA

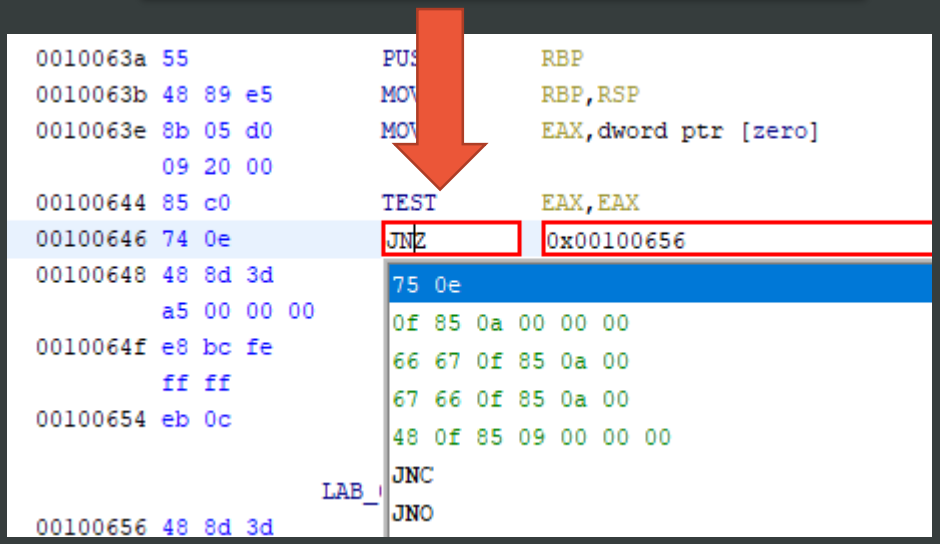


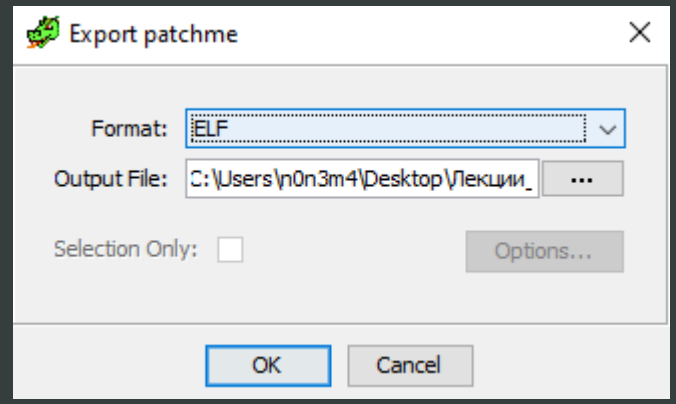


Редактирование файлов Ghidra

- С недавних пор Ghidra тоже научилась патчить исполняемые файлы, для этого можно нажать на нужной инструкции ПКМ → Patch instruction
- Сохранить запатченный файл можно при помощи File → Export program, а там выбрав формат ELF
- Тоже весьма неплохо работает, разве что NOPами автоматически не добивает

Пишем инструкцию прямо здесь, очень удобно





О помощи патчинга в отладке

- Иногда бывает, что искать все антиотладки в программе сложно, а вам очень хочется снять с программы дамп в определенный момент ее исполнения
- Здесь вам также может прийти на помощь патчинг: вы можете воспользоваться конструкцией типа EBFE для вашей целевой платформы, чтобы установить импровизированную точку останова
  - На x86 это бесконечный цикл, а именно JMP -0x2
- Тогда программа зависнет ровно в том месте, где нужно, а там вы можете или подключить отладчик (и заменить байты обратно через него, продолжив исполнение) или снять дамп
- Также для снятия дампа можно воспользоваться знаменитой инструкцией UD2, вызывающей падение программы, если у вас включено сохранение дампов через ulimit, но это менее кросс-платформенное решение (ну и мало ли, может процесс прячет часть страниц памяти от крашдампов)

Время задач

Ты не пройдешь

Категория: Lesson 18 / Antidebug + Patching

Решивших: 0

Время: 00:00:02

- Доступ к задачам можно получить как всегда на nsuctf.ru
- В этой задаче вам может пригодиться Ghidra или плагин IDA Keypatch (https://github.com/keystone-engine/keypatch)
- Или компилятор опкодов https://defuse.ca/online-x86-assembler.htm

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