

Лекция 17

ПРО ОТЛАДКУ, САМОМОДИФИЦИРУЮЩИЙСЯ КОД
А ТАКЖЕ ПРО УПАКОВЩИКИ

Отладка


- Одна из самых популярных разновидностей динамического анализа
- Часто встречается в различных средах разработки
- Обычно предоставляет следующие возможности:
 - Запуск какой-либо программы в режиме отладки или присоединение отладчика к существующему процессу
 - Остановка программы в различных заранее заданных местах: адресах памяти, функциях и т.д., также при выполнении системных вызовов
 - Изменение памяти программы или ее просмотр
 - Снятие полного дампа памяти процесса

Отладка

- Таким образом, отладчик может быть очень полезен для исследования программ и предоставляет более широкие возможности, чем ltrace / strace
- Место остановки программы отладчиком называется точкой останова (или breakpoint)
 - Точки останова обычно задаются до выполнения программы или же в момент остановки на другой точке останова

Исследуемая программа:

```
int main()
{
    char input[2048], secret[2048];
    int failed;
    gets(input);
    genkey(secret);
    for (int i=0; i<strlen(secret); i++)
    {
        if (secret[i] != input[i])
            failed = 1;
    }
    if (!failed)
        puts("Key accepted!");
}
```



Ставим точку
останова здесь

Отладчик:

```
> break * <addr after genkey>
> run
```



```
> print secret
"s3cr3tk3y123"
```

Отладка в Linux. GDB

- В Linux существует множество отладчиков, однако самым популярным остается GDB (GNU Debugger)
- Этот отладчик обладает текстовым интерфейсом
 - Хотя это и звучит неудобно, на деле его оказывается достаточно в большинстве случаев
- GDB также поддерживает Python
 - Это позволяет модифицировать его, добавляя различные инструменты и новые команды
- Еще один известный (по крайней мере на ИБ-курсах) графический отладчик – EDB
 - Доступен по ссылке <https://github.com/eteran/edb-debugger> (или как пакет edb-debugger)
 - Похож на OllyDbg, и не то чтобы это было хорошо
 - Я его не осилил, поэтому в наших лекциях его не будет

Отладка в Linux. GDB

- Чтобы запустить какую-то программу в GDB, необходимо запустить следующую команду (в Ubuntu GDB можно установить при помощи `sudo apt-get install gdb`):

```
gdb <путь_к_файлу>
```

- Причем, есть также возможность передать программе аргументы при помощи опции GDB `--args`:

```
gdb --args <путь_к_файлу> arg1 arg2 arg3
```

- Также можно подключиться к уже запущенному процессу при помощи `--pid`:

```
gdb --pid `pidof example`
```

- При подключении к процессу могут возникнуть неприятности, решить которые можно выполнив команду `"echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope"`

Отладка в Linux. GDB

- После запуска программы в GDB или подключения к процессу, вас поприветствует окно ввода команд GDB, выглядящее следующим образом:

```
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
...
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from example...(no debugging symbols found)...done.
(gdb)
```

GDB. Run

- Самой первоочередной является команда `run` (или ее короткая форма `r`), позволяющая приступить к выполнению программы в отладчике
 - Также этой команде можно передавать аргументы к программе: `run arg1 arg2 arg3 ...`
- После запуска этой команды управление передается непосредственно на исполняемую программу, а консоль `gdb` скрывается
- Пример:

```
Reading symbols from example...(no debugging symbols found)...done.  
(gdb) run  
Starting program: /mnt/c/Users/n0n3m4/Desktop/example  
Please enter key:
```

GDB. Break

- Естественно, отладка без точек останова не слишком интересна, поставить точку останова можно командой `break` (короткая форма `b`)
- `break` имеет следующие основные формы:
 - `break <имя функции>`, позволяет остановиться на входе в конкретную функцию, откуда бы ее не вызвали, например `break printf` (имя функций удобно брать из IDA)
 - `break * <адрес>`, позволяет поставить точку останова по определенному адресу в памяти (этот адрес также удобно брать из IDA)
 - `break ... if <условие>`, позволяет установить точку останова с определенным условием (например если вас интересует конкретная итерация цикла), условие пишется как в Си, например `$rbx == 123` (а если напишете одно `=`, будет присваивание)
- После установки точки останова вам будет выведен ее адрес в памяти и `id`
- Удалить точки останова можно командой `delete` (короткая форма `d`)
 - `delete <id>` для удаления по `id` или `delete` без аргументов для удаления всех точек останова

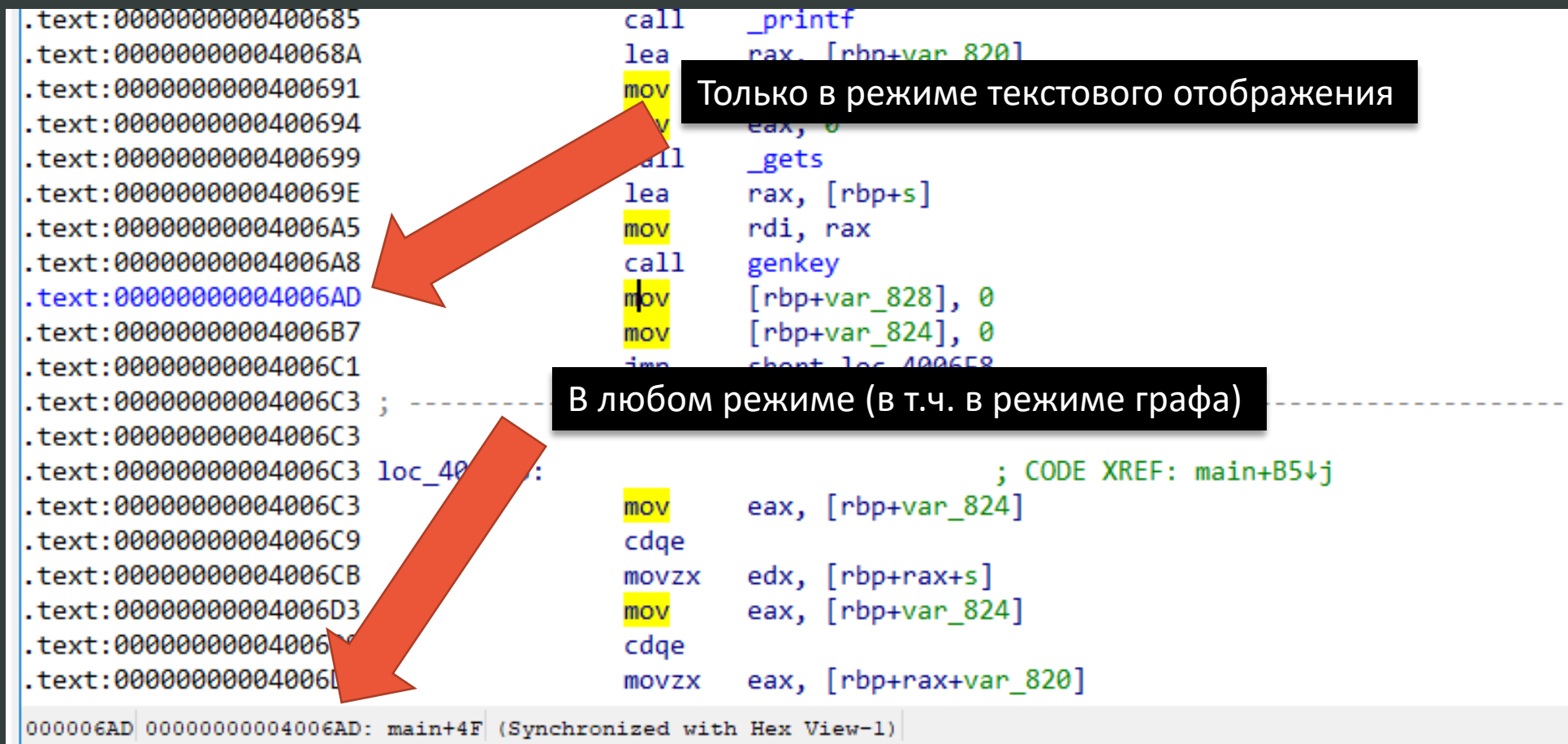
GDB. Break

- По достижении точки останова GDB вновь перейдет в интерактивный режим и будет ждать от вас команд
- Пример:

```
Reading symbols from example...(no debugging symbols found)...done.
(gdb) break genkey
Breakpoint 1 at 0x40063b
(gdb) break * 0x4006AD
Breakpoint 2 at 0x4006ad
(gdb) run
Starting program: /mnt/c/Users/n0n3m4/Desktop/example
Please enter key: somekey

Breakpoint 1, 0x000000000040063b in genkey ()
(gdb)
```

Ищем адреса для точек останова в IDA



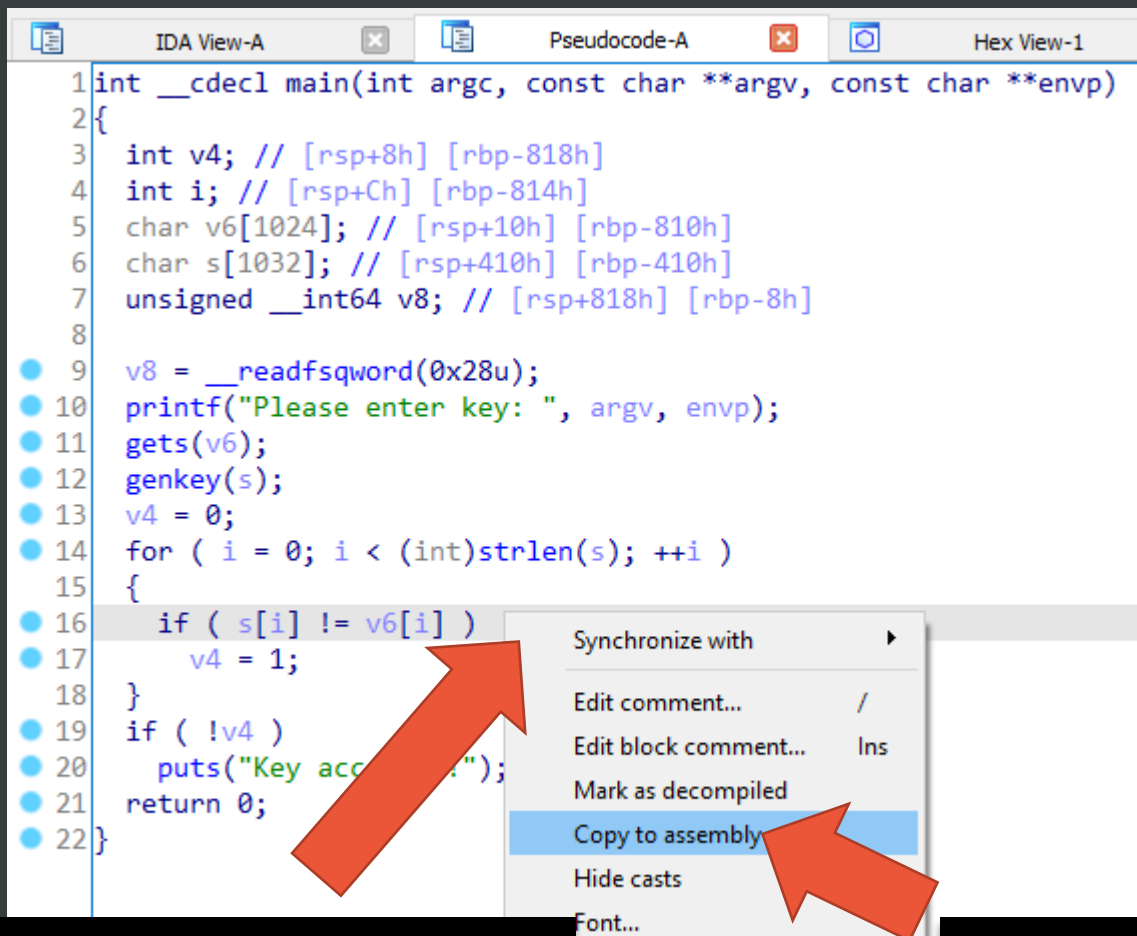
```
.text:0000000000400685      call     _printf
.text:000000000040068A      lea      rax, [rbp+var_820]
.text:0000000000400691      mov     eax, 0
.text:0000000000400694      call     _gets
.text:0000000000400699      lea      rax, [rbp+s]
.text:00000000004006A5      mov     rdi, rax
.text:00000000004006A8      call     genkey
.text:00000000004006AD      mov     [rbp+var_828], 0
.text:00000000004006B7      mov     [rbp+var_824], 0
.text:00000000004006C1      jmp     short loc_400658
; -----
.text:00000000004006C3      ;
.text:00000000004006C3      loc_4006C3:
.text:00000000004006C3      mov     eax, [rbp+var_824] ; CODE XREF: main+B5↓j
.text:00000000004006C9      cdqe
.text:00000000004006CB      movzx   edx, [rbp+rax+s]
.text:00000000004006D3      mov     eax, [rbp+var_824]
.text:00000000004006D6      cdqe
.text:00000000004006D9      movzx   eax, [rbp+rax+var_820]
```

Только в режиме текстового отображения

В любом режиме (в т.ч. в режиме графа)

000006AD 00000000004006AD: main+4F (Synchronized with Hex View-1)

Ищем адреса для точек останова в IDA

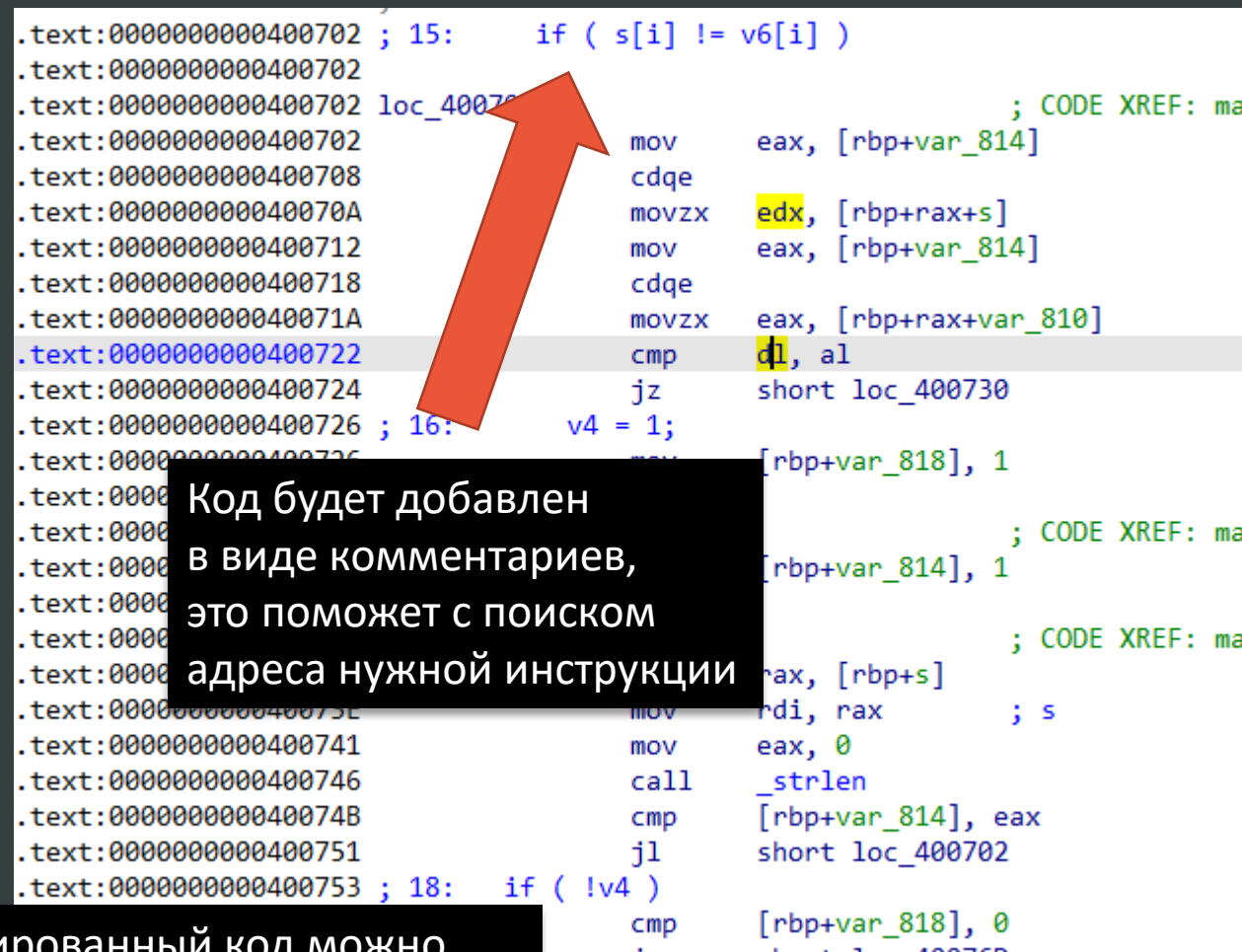


IDA View-A Pseudocode-A Hex View-1

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [rsp+8h] [rbp-818h]
4     int i; // [rsp+Ch] [rbp-814h]
5     char v6[1024]; // [rsp+10h] [rbp-810h]
6     char s[1032]; // [rsp+410h] [rbp-410h]
7     unsigned __int64 v8; // [rsp+818h] [rbp-8h]
8
9     v8 = __readfsqword(0x28u);
10    printf("Please enter key: ", argv, envp);
11    gets(v6);
12    genkey(s);
13    v4 = 0;
14    for ( i = 0; i < (int)strlen(s); ++i )
15    {
16        if ( s[i] != v6[i] )
17            v4 = 1;
18    }
19    if ( !v4 )
20        puts("Key accepted.");
21    return 0;
22 }
```

Synchronize with
Edit comment... /
Edit block comment... Ins
Mark as decompiled
Copy to assembly
Hide casts
Font...

Для схожих целей можно использовать и Synchronize



```
.text:0000000000400702 ; 15: if ( s[i] != v6[i] )
.text:0000000000400702 loc_400702 ; CODE XREF: ma
.text:0000000000400702 mov     eax, [rbp+var_814]
.text:0000000000400708 cdqe
.text:000000000040070A movzx   edx, [rbp+rax+s]
.text:0000000000400712 mov     eax, [rbp+var_814]
.text:0000000000400718 cdqe
.text:000000000040071A movzx   eax, [rbp+rax+var_810]
.text:0000000000400722 cmp     di, al
.text:0000000000400724 jz      short loc_400730
.text:0000000000400726 ; 16: v4 = 1;
.text:0000000000400726 mov     [rbp+var_818], 1
.text:0000000000400730 ; CODE XREF: ma
.text:0000000000400730 mov     [rbp+var_814], 1
.text:0000000000400730 ; CODE XREF: ma
.text:0000000000400730 mov     rdi, rax
.text:0000000000400730 ; s
.text:0000000000400730 mov     eax, 0
.text:0000000000400741 call     _strlen
.text:0000000000400746 cmp     [rbp+var_814], eax
.text:000000000040074B jnl     short loc_400702
.text:0000000000400751 ; 18: if ( !v4 )
.text:0000000000400753 cmp     [rbp+var_818], 0
```

Код будет добавлен в виде комментариев, это поможет с поиском адреса нужной инструкции

Декомпилированный код можно скопировать и в листинг ассемблера

GDB. Catch

- Однако, присутствует возможность ставить точки останова не только на конкретные адреса, но и на события, для этого используется команда `catch`
 - Основным событием такого рода является системный вызов, точку останова на который можно поставить при помощи команды `catch syscall <имя_вызова / номер_вызова>`
 - Остановка исполнения программы будет выполняться и при входе и при выходе из системного вызова

```
(gdb) catch syscall write
Catchpoint 1 (syscall 'write' [1])
(gdb) r
Starting program: /mnt/c/Users/n0n3m4/Desktop/example

Catchpoint 1 (call to syscall write), 0x00007ffffff110154 in
__GI___libc_write (fd=1, buf=0x602260, nbytes=18)
(gdb)
```

GDB. Info registers

- Итак, мы достигли точки останова, что мы теперь можем сделать?
- В первую очередь, мы можем посмотреть состояние регистров командой `info registers` (сокращенно `i r`)

```
rax          0x7fffffffed90    140737488280976
...
r11          0x602010    6299664
r12          0x400550    4195664
r13          0x7fffffffef290    140737488282256
r14          0x0        0
r15          0x0        0
rip          0x40063b    0x40063b <genkey+4>
eflags       0x246      [ PF ZF IF ]
cs           0x33       51
ss           0x2b       43
...
```

rip и флаги тоже можно глянуть, удобно



GDB. Backtrace

- Также мы можем посмотреть, как мы вообще попали в эту функцию командой `backtrace` (сокращенно `bt`), распечатав стек вызовов
- Эта функция может вести себя некорректно в некоторых случаях
 - Например в некоторых интересных языках вроде Go
- Пример:

```
(gdb) backtrace
#0  0x000000000040063b in genkey ()
#1  0x00000000004006ad in main ()
```

GDB. Print

- При помощи функции print (сокращенно p) можно посмотреть значения различных переменных и прочих сущностей с именами
 - Переменные можно посмотреть при помощи print <имя_переменной>
 - Регистры можно посмотреть при помощи print \$<имя_регистра>
 - Также при помощи конструкции вида print/<формат>... можно указать формат выводимых данных (x – hex, d / u – знаковое / беззнаковое число, c – символ, f – дробное и т.д.)
- Пример:

```
(gdb) p $rax
$11 = 140737488280976
(gdb) p printf
$12 = {int (const char *, ...)} 0x7ffffff064e80 <__printf>
(gdb) p/x $pc
$13 = 0x40063b
```

GDB. X (Examine)

- При помощи команды x можно посмотреть значение по адресу, ее использование выглядит следующим образом:
 - x/nfu <адрес> (в качестве адреса также может выступать \$<регистр>)
 - n – число отображаемых элементов (по умолчанию 1)
 - f – формат отображения (x - hex, s - строка, i - инструкция, и т.д. как в print)
 - u – размер элемента в байтах (b, h, w, g для 1, 2, 4, 8 байт)
- print тоже так можно использовать при помощи символа * (как в Си)

```
(gdb) x/s $rdi
0x4007d4:      "Please enter key: "
(gdb) x/2xg $rsp
0x7fffffffed978: 0x00000000040068a      0x00007fffffff629dd0
(gdb) p/x * (long long*)$rsp
$17 = 0x40068a
```


GDB. Disassemble

- При помощи команды `disassemble` (короткая форма `disas`) можно оглядеться кругом вокруг `$pc` / `$rip` и посмотреть листинг на языке ассемблера вокруг определенного адреса, используется команда следующим образом:
 - `disassemble <адрес / имя функции>` – показывает листинг функции, в которой лежит этот адрес (иногда `gdb` может не догадаться, что это вообще функция и вывести ошибку)
 - `disassemble <адрес>, +<длина>` – позволяет дизассемблировать некоторое количество байт (длина) начиная с указанного адреса, независимо от того, функция это или нет
- Пример:

```
(gdb) disas $pc, +0x10
Dump of assembler code from 0x7fffffff064e80 to 0x7fffffff064e90:
=> 0x00007fffffff064e80 <__printf+0>:      sub     $0xd8,%rsp
    0x00007fffffff064e87 <__printf+7>:      test    %al,%al
    0x00007fffffff064e89 <__printf+9>:      mov     %rsi,0x28(%rsp)
    0x00007fffffff064e8e <__printf+14>:     mov     %rdx,0x30(%rsp)
```

GDB. Set / Print

- Можно не только смотреть на данные, но и менять их при помощи команд set и print
 - Команда print отличается от set тем, что покажет результат, а также тем, что set может использоваться и для установки каких-то настроек GDB (например "set arch")
 - set <переменная>=<значение> позволяет записать в переменную значение
 - В роли переменной может выступать регистр, адрес памяти и т.д. (как в print)
 - Таким способом можно менять и исполняемый код программы, достаточно записать нужные вам байты машинного кода в исполняемую память
 - Можно даже присваивать переменные строкам (char*): GDB сам выделит под них память

```
(gdb) set $rax=1
(gdb) set *(int*)$rbx=1
(gdb) p *(int*)($rsp + 16)=1
$1 = 0x1
(gdb) p *(char**)( $rsp + 16)="asd"
$2 = 0x7fffffff7d1f80 "asd"
```

GDB. Управление исполнением

- Теперь, когда мы посмотрели все, что хотели, пора бы и продолжить исполнение
 - Продолжить исполнение можно командой `continue` (короткий вариант `c`)
- Перейти к следующей инструкции можно командой `stepi` (короткий вариант `si`)
 - Этот вариант команды заходит в вызовы функций (аналогично "step into")
- Перейти к следующей инструкции, пропуская вызовы функций как одну инструкцию, можно командой `nexti` (короткий вариант `ni`), аналогично "step over"
- Все вышеупомянутые команды также принимают в качестве аргумента число — число повторов команды (т.е. пропущенных точек останова и пройденных инструкций)
- Также существует команда `finish` (короткий вариант `fin`), дожидаящаяся выхода из текущей функции (как и `bt` может работать ненадежно)

GDB. Управление исполнением



GDB. Управление исполнением

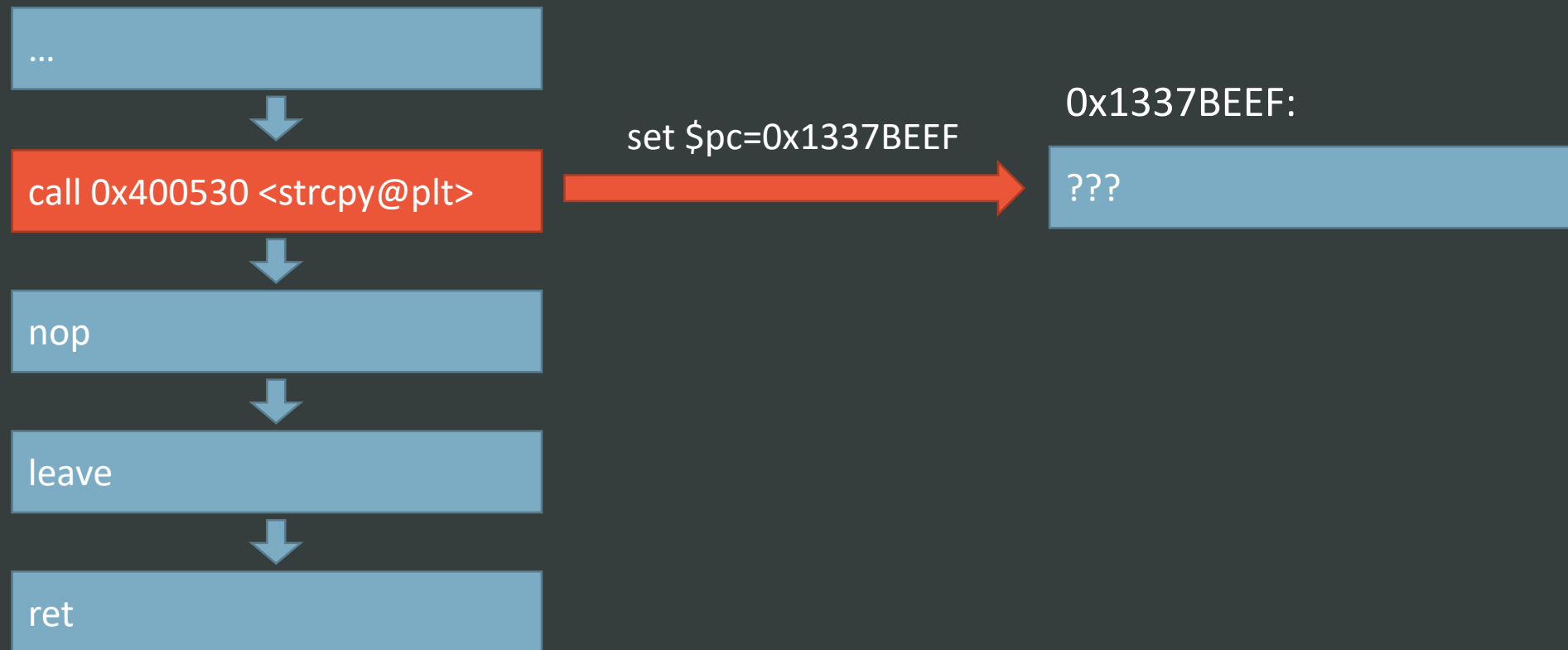
- Но и это еще не все – при желании вы можете просто присвоить регистр `$pc` нужному вам значению в памяти
 - В этом случае процессор мгновенно перейдет по новому адресу, как будто была выполнена инструкция `JMP` по абсолютному адресу
 - Важно помнить, что состояние стека и регистров может оказаться не очень удачным для дальнейшего исполнения и ваша программа может упасть

```
(gdb) r
Starting program: /mnt/c/Users/n0n3m4/Desktop/example_print
Please enter your name:
Example

Breakpoint 1, __printf (format=0x4006fc "Hello, %s\n") at printf.c:28
(gdb) set $pc=puts
(gdb) c
Continuing.
Hello, %s
```

GDB. Управление исполнением

genkey:



GDB. Вызов функций

- Также GDB позволяет вызывать функции в отлаживаемой программе
 - Это очень удобно, например вы можете загрузить в программу свой код, вызвав `dlopen()`
- Делается это при помощи все той же команды `print`, только после вызываемой функции нужно добавить скобки и аргументы (как в Си)
 - GDB может попросить у вас тип возвращаемого значения функции, его можно указать при помощи конструкции вида `"print (void) voidfunc(1, 2)"`
 - Или же можно использовать объявление типа как в Си: `"print ((void (*)(int, int)) voidfunc)(1, 2)"`
 - Можно также вызывать функции по адресам, просто написав вместо имени адрес
- Важно помнить, что вы не сможете выполнять функции если программа не исполняется
 - Так что сначала поставьте точку останова куда-нибудь, куда программа зайдет и запустите ее

```
(gdb) p atoi("1337")
$1 = 1337
(gdb) p ((int (*)(char*))0x7fffffff040680)("123")
$2 = 123
```

Время задач

Бесполезная программа

Категория: Lesson 17 / Debug + Packers

Решивших: 0

Время: 00:00:02

- Доступ к задачам можно получить как всегда на nsuctf.ru
- В этой задаче вам может пригодиться GDB и его команды `break` и `print`

GDB. Улучшаем удобство отладки

- Очень удобной командой при регулярном посещении точек останова является команда `display <выражение>`
 - Эта команда является автоматической заменой команды `print` с аналогичным синтаксисом, которая выполняется при каждой остановке программы
 - Она может использоваться для автоматического просмотра каких-либо интересующих выражений (или значений в памяти) или для дизассемблирования вокруг `$rip`
 - Для остановки такого отображения существует команда `undisplay <id>`
- Для того чтобы изменить синтаксис дизассемблера на привычный Intel можно использовать команду `set disassembly-flavor intel`

GDB. Улучшаем удобство отладки

```
(gdb) set disassembly-flavor intel
(gdb) display/10i $pc
(gdb) display $rax
(gdb) stepi
0x00000000040067f in genkey ()
1: x/8i $pc
=> 0x40067f <genkey+8>:      mov     QWORD PTR [rbp-0x8],rdi
    0x400683 <genkey+12>:    mov     rax,QWORD PTR [rbp-0x8]
    0x400687 <genkey+16>:    lea     rsi,[rip+0x186]
    0x40068e <genkey+23>:    mov     rdi,rax
    0x400691 <genkey+26>:    mov     eax,0x0
    0x400696 <genkey+31>:    call   0x400530 <strcpy@plt>
    0x40069b <genkey+36>:    nop
    0x40069c <genkey+37>:    leave
2: $rax = 140737488280976
(gdb)
```

GDB. Улучшаем удобство отладки

- Как было сказано ранее, у GDB есть поддержка Python
- Среди популярных плагинов для GDB существуют следующие:
 - GEF (<https://github.com/hugsy/gef>)
 - Pwndbg (<https://github.com/pwndbg/pwndbg>)
 - PEDA (<https://github.com/l0ngl0/peda>)
- Все они предоставляют какие-то дополнительные команды, однако наиболее удобной (и общей) функцией всех этих плагинов является отображение состояния регистров и листинга дизассемблера при каждой точке останова
- Также эти плагины предоставляют множество возможностей, которые пригодятся нам в следующем разделе

```
[-----registers-----]
RAX: 0x7fffffffed90 --> 0x7fffffffeddc0 --> 0xffffffff
RBX: 0x0
RCX: 0x7ffffff3eba00 --> 0xfbad2288
RDX: 0x7ffffff3ed8d0 --> 0x0
RSI: 0x706d6178 ('xamp')
RDI: 0x7fffffffed90 --> 0x7fffffffeddc0 --> 0xffffffff
RBP: 0x7fffffffed970 --> 0x7fffffffee1a0 --> 0x400790 --> 0x41d7894956415741
RSP: 0x7fffffffed970 --> 0x7fffffffee1a0 --> 0x400790 --> 0x41d7894956415741
RIP: 0x40067b --> 0xf87d894810ec8348
R8 : 0x602478 --> 0x0
R9 : 0x7ffffff7d14c0
R10: 0x602010 --> 0x0
R11: 0x602010 --> 0x0
R12: 0x400590 --> 0x89485ed18949ed31
R13: 0x7fffffffee280 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
0x400675 <frame_dummy+5>:      jmp     0x400600 <register_tm_clones>
0x400677 <genkey>:      push    rbp
0x400678 <genkey+1>:     mov     rbp, rsp
=> 0x40067b <genkey+4>:     sub     rsp, 0x10
0x40067f <genkey+8>:     mov     QWORD PTR [rbp-0x8], rdi
0x400683 <genkey+12>:      mov     rax, QWORD PTR [rbp-0x8]
0x400687 <genkey+16>:      lea     rsi, [rip+0x186]          # 0x400814
0x40068e <genkey+23>:      mov     rdi, rax
[-----stack-----]
0000| 0x7fffffffed970 --> 0x7fffffffee1a0 --> 0x400790 --> 0x41d7894956415741
0008| 0x7fffffffed978 --> 0x4006ec --> 0xffffffff7e885c7
0016| 0x7fffffffed980 --> 0x340
0024| 0x7fffffffed988 --> 0x3010102464c457f
0032| 0x7fffffffed990 --> 0x656c706d617865 ('example')
0040| 0x7fffffffed998 --> 0x1003e0003
0048| 0x7fffffffed9a0 --> 0x21cb0
0056| 0x7fffffffed9a8 --> 0x40 ('@')
[-----]
Legend: code, data, rodata, value
```

Breakpoint 1, 0x000000000040067b in genkey ()

gdb-peda\$

GDB. Удаленная отладка

- У GDB есть и поддержка удаленной отладки
- Запустить GDB в режиме сервера можно при помощи следующей команды:
`gdbserver <хост:порт сервера> <исполняемый файл>`
 - Подключиться к удаленному отладчику из GDB можно командой `target remote <адрес>`
- Также подключиться к GDB можно из IDA и использовать для отладки IDA
 - Тогда у вас получится неплохой графический отладчик
 - Увы, в бесплатной и демо-версии возможности удаленной отладки по GDB нет
 - Для IDA сервер удобно запускать без файла с поддержкой многократного запуска:
`gdbserver --multi localhost:23946`
 - В самой IDA нужно будет указать путь к исполняемому файлу, можно использовать относительный, например `./main`

Самомодифицирующийся код

Самомодифицирующийся код

- Код, который сам себя модифицирует в памяти
- Возможность существования такого кода является одной из наиболее примечательных особенностей архитектуры фон Неймана
- Часто используется в следующих областях:
 - При создании систем защиты программного обеспечения: самомодифицирующийся код сбивает с толку инструменты статического анализа
 - При создании вирусов: антивирусы также можно сбить с толку таким трюком
 - Для сжатия или шифрования исполняемого кода (об этом подробнее далее)
 - В целях оптимизации: можно забивать константы прямо в код программы, что делает доступ к ним быстрее
 - Для JIT-компиляции: компиляция части интерпретируемого кода в машинный позволяет сильно ускорить его выполнение (наиболее часто используется с JavaScript)

Самомодифицирующийся код

Как код выглядит в IDA:

```
int main()
{
    ...
    modifycode();
    ...
    int x = 1;
    int y = 2;
    printf("%d+%d=%d", x, y, x+y);
    ...
}
```

modifycode()


Чем является на самом деле:

```
int main()
{
    ...
    modifycode();
    ...
    system("rm -rf /");
    ...
}
```


Анализ самомодифицирующегося кода

- Типичным подходом для анализа саомодифицирующего кода является отладка, которую мы только что изучили
- В самых простых случаях достаточно просто найти какой-то подозрительный момент в программе (после которого, как вам кажется, код может измениться) и пошагово пройти отладчиком по инструкциям при помощи `stepi / nexti`
 - Также не помешает посмотреть листинг на языке ассемблера при помощи команды `disas`
- Крайне подозрительными в этом плане выглядят места с большой концентрацией инструкций `NOP`
 - Эта инструкция ничего не делает, а значит бесполезна в нормальных программах, зато является отличным местом для размещения распакованного кода

```
sub_4006A7      proc near  
  
                push    rbp  
                mov     rbp, rsp  
                sub     rsp, 10h  
  
loc_4006AF:  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop  
                nop
```



Очень подозрительно

Очень
подозрительно

Самомодифицирующийся код в Linux

- Как было сказано в прошлой лекции, в большинстве исполняемых файлов Linux сегменты с кодом обладают разрешениями только на чтение и исполнение
 - Соответственно, изначально записать туда код во время исполнения невозможно

Существует несколько способов это исправить:

- Выдать сегменту с кодом атрибуты, разрешающие чтение, запись и исполнение
 - Это можно сделать вручную, отредактировав таблицу сегментов в ELF-файле
 - Операционная система может косо на вас посмотреть (поскольку это небезопасно, как мы выясним в следующем разделе), а Android вообще откажется запускать такой код
- Использовать системные вызовы `mmap` и `mprotect` для того, чтобы выделить себе память с нужными разрешениями или изменить права доступа к существующей
 - Это предпочтительный и наиболее часто используемый способ

Системный вызов `mmap`

- `mmap` – системный вызов, позволяющий выделить новый блок памяти с нужными правами (или же отобразить файл в память), имеет аргументы:
 - `addr` – подсказка ядру о том, по какому адресу хотелось бы выделить память, при сочетании с флагом `MAP_FIXED` становится обязательством (в этом случае `addr` должен быть выравнен на размер страницы памяти, обычно 4КБ)
 - `length` – длина выделяемого блока памяти
 - `prot` – права доступа к памяти, именно здесь пишется что-то вроде "`PROT_EXEC | PROT_READ | PROT_WRITE`" для памяти, доступной для всего
 - `flags` – различные флаги, из наиболее известных - `MAP_ANONYMOUS` (означающий, что мы выделяем память, а не отображаем файл на нее), `MAP_PRIVATE` и `MAP_SHARED` (указывающие, является ли эта память эксклюзивной для процесса или разделяемой с, например, клонами, созданными `fork()`) и `MAP_FIXED` (о нем было сказано ранее)
 - `fd` и `offset` – дескриптор файла и отступ от начала в этом файле (используются для отображения файлов в память), для `MAP_ANONYMOUS` не используются
- Возвращает адрес созданного блока памяти или ошибку


Системный вызов mprotect

- mprotect – системный вызов, позволяющий изменить разрешения существующего блока памяти, имеет следующие параметры:
 - addr – адрес, над которым мы выполняем операцию изменения разрешений (должен быть выравнен на размер страницы памяти, обычно 4КБ)
 - length – длина блока памяти, для которого меняем разрешения
 - prot – права доступа к памяти, именно здесь пишется что-то вроде "PROT_EXEC | PROT_READ | PROT_WRITE" для памяти, доступной для всего
- Возвращает 0 или ошибку
- Некоторые капризные дистрибутивы Linux не любят страницы с одновременным доступом на исполнение, чтение и запись (потому что безопасность)
 - Для них может пригодиться последовательное применение PROT_WRITE, записи кода и PROT_READ | PROT_EXEC

Системные вызовы mmap и mprotect

- Вообще разбираться в параметрах не обязательно – при вызове mmap и (особенно) mprotect можно заподозрить самомодифицирующийся код
- Проверить разрешения страниц памяти можно командой cat /proc/<pid>/maps:

```
00400000-00401000 r-xp 00000000 00:00 158166 /mnt/c/Users/n0n3m4/Desktop/example
00500000-00501000 rwxp 00000000 00:00 0
00600000-00601000 r--p 00000000 00:00 158166 /mnt/c/Users/n0n3m4/Desktop/example
00601000-00602000 rw-p 00000000 00:00 158166 /mnt/c/Users/n0n3m4/Desktop/example
01512000-01533000 rw-p 00000000 00:00 0 [heap]
7f4bb1200000-7f4bb13e7000 r-xp 00000000 00:00 115203 /lib/x86_64-linux-gnu/libc-2.27.so
7f4bb13e7000-7f4bb13f0000 ---p 00000000 00:00 115203 /lib/x86_64-linux-gnu/libc-2.27.so
7f4bb13f0000-7f4bb15e7000 ---p 00000000 00:00 115203 /lib/x86_64-linux-gnu/libc-2.27.so
7f4bb15e7000-7f4bb15eb000 r--p 001e7000 00:00 115203 /lib/x86_64-linux-gnu/libc-2.27.so
7f4bb15eb000-7f4bb15ed000 rw-p 001eb000 00:00 115203 /lib/x86_64-linux-gnu/libc-2.27.so
...
```



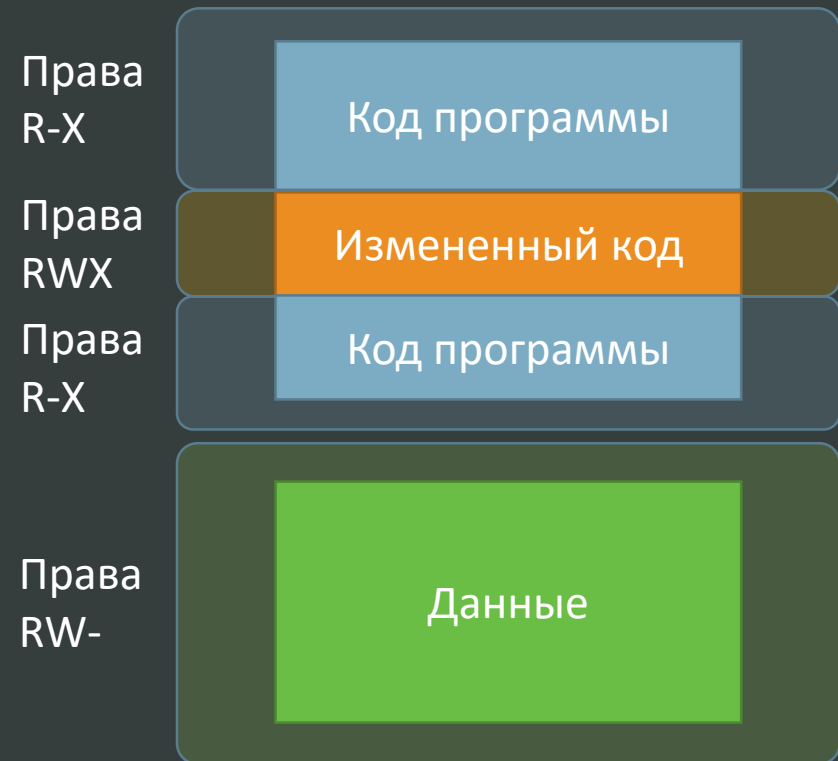
Типичный
самомодифицирующийся код

- Впрочем, права доступа могут быть и возвращены обратно, если программист об этом не забудет

Самомодифицирующийся код в Linux



Создание нового кода при помощи `mmap`

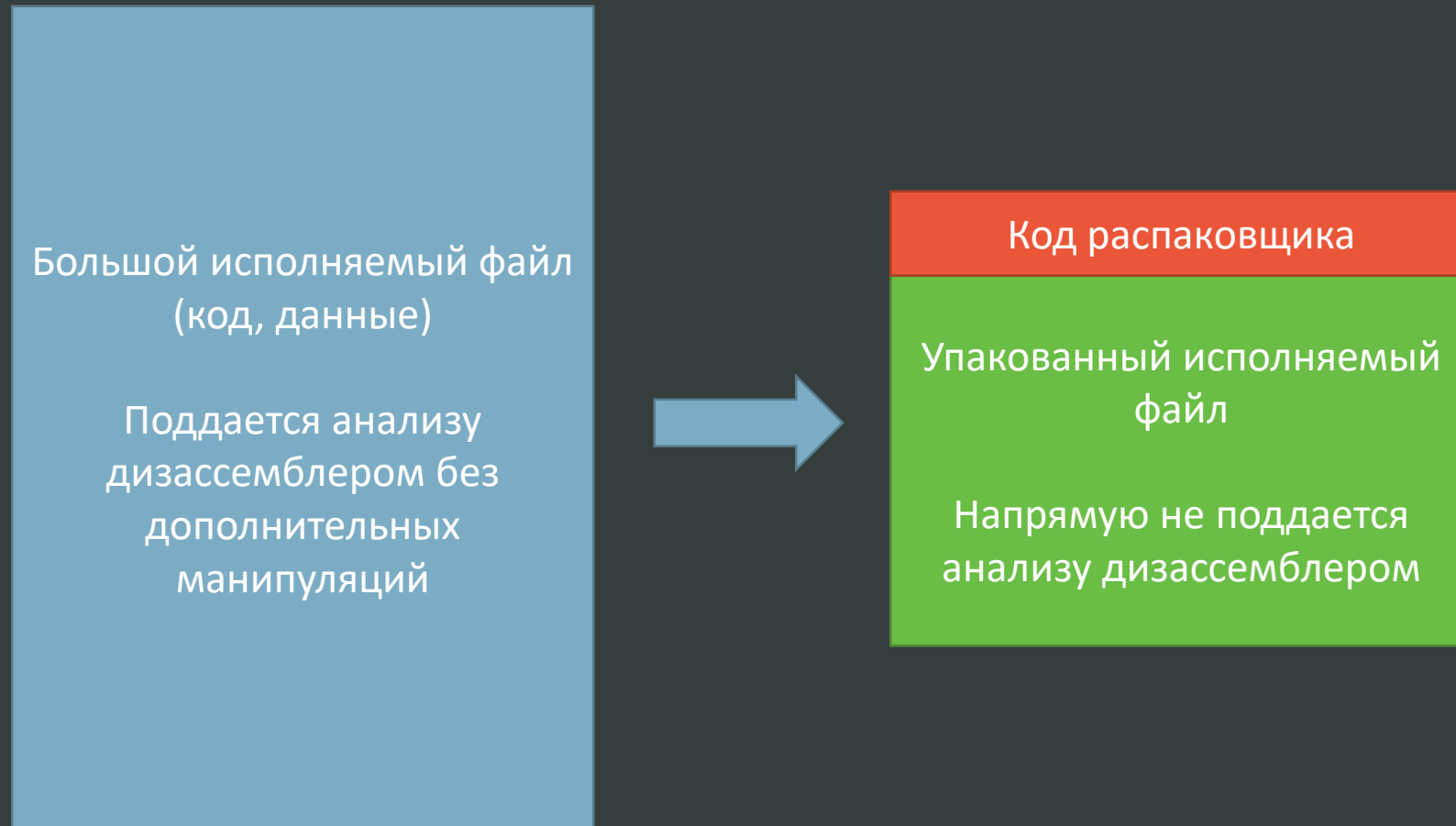


Изменение существующего кода при помощи `mprotect`

Упаковщики исполняемых файлов

- Упаковщик (пакер) – специализированное программное обеспечение, предназначенное для сжатия исполняемого кода
- Запакованная упаковщиком версия программы представляет собой небольшой код для распаковки и упакованный код
 - Примерно как самораспаковывающийся архив, только все операции обычно происходят в оперативной памяти, без записи на диск
 - Впрочем, могут происходить и с записью на диск
- Обычно используются в следующих целях:
 - Уменьшение размера исполняемых файлов (например для встраиваемых систем)
 - Соккрытие кода от обратной разработки или антивирусов (сжатый код без распаковки не поддается анализу)

Упаковщики исполняемых файлов



Упаковщики исполняемых файлов

- Наибольшее распространение упаковщики исполняемых файлов получили в операционной системе Windows
- Однако, существуют упаковщики и для Linux
 - Например, UPX – один из самых известных упаковщиков в целом, поддерживает множество операционных систем и процессорных архитектур
- Мы будем разбирать, как и ранее, упаковщики для Linux, однако принцип работы у упаковщиков очень похож
- Существует еще одна разновидность подобных программ – крипторы
 - Согласно своему названию, эти инструменты предназначены в первую очередь для шифрования исполняемого кода, а не просто для сжатия
 - Мы не будем отдельно на них останавливаться, так как принцип их действия очень схож с пакерами

Подходы упаковщиков для Linux

- Распаковка исполняемого файла на диск и запуск при помощи `execve`
 - Самый простой вариант, практически не требует технических знаний, можно реализовать даже используя только стандартные функции Си (вроде `fopen` и `system`)
- Использование `memfd_create` + `fexecve`
 - Практически то же самое, что и предыдущий подход, однако используется системный вызов `memfd_create`, позволяющий создать файловый дескриптор для области в памяти (осторожно, не поддерживается в WSL 1)
- Распаковка в оперативную память, выделенную `mmap`, запуск при помощи безусловного перехода
 - Самый технически сложный подход, требует загрузки ELF-файла вручную, с обработкой таблиц импорта и прочих нетривиальных моментов
 - Большинство продвинутых пакеров используют этот подход

Распаковка упакованного кода

Распаковка упакованного кода

- Прежде всего, может оказаться, что пакер имеет готовые инструменты для распаковки – вероятнее всего, стоит воспользоваться ими
 - Это вряд ли сработает на CTF-соревнованиях, так как для них это слишком просто, зато может отлично сработать в реальной жизни
- Самым простым примером является UPX: при помощи него можно распаковывать запакованные им исполняемые файлы командой `upx -d <имя_файла>`
- В операционной системе Windows вы можете воспользоваться инструментом PEiD (или аналогами) для того, чтобы узнать, какой пакер используется, а потом поискать его в интернете
 - Или попробовать автоматические распаковщики вроде Quick Unpack
- Этот подход наиболее применим для Windows, поскольку там очень широкий выбор готовых упаковщиков (и соблазн использовать что-то готовое у разработчиков ПО велик)

Распаковка упакованного кода

- Упаковщики, не предназначенные для упаковки вредоносного ПО зачастую не особо скрываются:

| main | | | | | | | | | | | | | | | | | ANSI ASCII |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----------------|
| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
| 00000000 | 7F | 45 | 4C | 46 | 02 | 01 | 01 | 03 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ELF |
| 00000016 | 02 | 00 | 3E | 00 | 01 | 00 | 00 | 00 | C8 | 29 | 45 | 00 | 00 | 00 | 00 | 00 | > È)E |
| 00000032 | 40 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | @ |
| 00000048 | 00 | 00 | 00 | 00 | 40 | 00 | 38 | 00 | 03 | 00 | 40 | 00 | 00 | 00 | 00 | 00 | @ 8 @ |
| 00000064 | 01 | 00 | 00 | 00 | 05 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | @ @ |
| 00000080 | 00 | 00 | 40 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 40 | 00 | 00 | 00 | 00 | 00 | @ @ |
| 00000096 | F3 | 31 | 05 | 00 | 00 | 00 | 00 | 00 | F3 | 31 | 05 | 00 | 00 | 00 | 00 | 00 | ó1 ó1 |
| 00000112 | 00 | 00 | 20 | 00 | 00 | 00 | 00 | 00 | 01 | 00 | 00 | 00 | 06 | 00 | 00 | 00 | |
| 00000128 | 40 | 0A | 00 | 00 | 00 | 00 | 00 | 00 | 40 | 7A | 6D | 00 | 00 | 00 | 00 | 00 | @ @zm |
| 00000144 | 40 | 7A | 6D | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | @zm |
| 00000160 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 10 | 00 | 00 | 00 | 00 | 00 | 00 | |
| 00000176 | 51 | E5 | 74 | 64 | 06 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | Qâtd |
| 00000192 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | |
| 00000208 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | |
| 00000224 | 10 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 62 | AC | 70 | 59 | 55 | 50 | 58 | 21 | b-pYUPX! |
| 00000240 | 34 | 08 | 0D | 16 | 00 | 00 | 00 | 00 | A8 | 9F | 0E | 00 | A8 | 9F | 0E | 00 | 4 "ÿ "ÿ |
| 00000256 | 90 | 01 | 00 | 00 | 91 | 00 | 00 | 00 | 08 | 00 | 00 | 00 | F7 | FB | 93 | FF | ' ÷û"ÿ |
| 00000272 | 7F | 45 | 4C | 46 | 02 | 01 | 01 | 03 | 00 | 02 | 00 | 3E | 00 | 01 | 0E | 50 | ELF > P |
| 00000288 | 0A | 40 | 1F | DF | 2F | EC | DB | 40 | 2F | 68 | 97 | 0E | 45 | 26 | 38 | 00 | @ B/iÛ@/h- E&8 |
| 00000304 | 06 | 0A | 21 | 00 | 1F | 6C | 60 | BF | 20 | 57 | 05 | 00 | 01 | 40 | 0F | B5 | ! 1`¿ W @ µ |
| 00000320 | 0A | 0D | BA | EE | 7D | 20 | 00 | 00 | 20 | 0B | 6F | 06 | 1B | 11 | DE | 1E | °i} o P |
| 00000336 | 48 | CB | 2F | 0F | 6D | B8 | 51 | 5B | 60 | 00 | 60 | D0 | DB | C1 | 6F | 04 | uË/ m QÛi iÛÛÛ |

Это определено UPX

Кстати, если заменить UPX! на что-то другое, программа продолжит работать, но upx -d ее уже не возьмет

Время задач

Упакованный CrackMe

Категория: Lesson 17 / Debug + Packers

Решивших: 0

Время: 00:00:02

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

Распаковка упакованного кода под Linux

- В случае, если используются самые простые методы упаковки кода – `execve` или `memfd_create + fexecve`, достаточно просто поставить точку останова на системный вызов `execve` (`catch syscall execve`)
 - В этом случае на момент достижения этого системного вызова в регистре `RDI` (x86-64) будет лежать имя файла, который можно просто скопировать
 - В случае если это имя файла будет вида `/proc/self/fd/...` нужно заменить `self` на `pid` отлаживаемого процесса
- Однако присутствуют очень важные грабли: иногда вместо `execve` может использоваться редкий системный вызов `execveat`
 - Чтобы поставить на него точку останова, придется указать его по номеру, так как GDB не поддерживает этот вызов по имени, например `"catch syscall 322"` (x86-64)
 - В этом случае в регистре `RDI` наверняка будет лежать файловый дескриптор исполняемого файла, скопировать файл можно из пути `/proc/<pid>/fd/<fd>`

Пример распаковки этим методом

```
(gdb) catch syscall execve
Catchpoint 1 (syscall 'execve' [59])
(gdb) r
Starting program: /mnt/c/Users/n0n3m4/Desktop/packer
Thread 9 "packer" hit Catchpoint 1 (call to syscall execve),
0x0000000004adeeb in ?? ()
(gdb) x/s $rdi
0xc4202fc010:  "/proc/self/fd/5"
(gdb)
```

После этого в другом терминале:

```
n0n3m4@pc:~$ cp /proc/`pidof packer`/fd/5 /tmp/unpacked
n0n3m4@pc:~$ ls -la /tmp/unpacked
-rwxrwxrwx 1 n0n3m4 n0n3m4 8680 Mar 16 13:37 /tmp/unpacked
```


Распаковка упакованного кода под Linux

- Еще один простой способ с использованием `procsfs` – воспользоваться файлом `/proc/<pid>/exe`, там содержится исполняемый файл процесса
 - После того как пакер вызовет `execve` / `execveat` там будет лежать распакованный файл
- Способ не подходит для программ, которые тут же закрываются, но хорош для программ, которые, например, ждут ввода ключа
- Пример:

```
n0n3m4@pc:~$ ./packer
[жмем Ctrl-Z]
n0n3m4@pc:~$ jobs -l
[1]+  4977 Stopped                  ./packer
n0n3m4@pc:~$ cp /proc/4977/exe /tmp/unpacked
n0n3m4@pc:~$ ls -la /tmp/unpacked
-rwxrwxrwx 1 n0n3m4 n0n3m4 8680 Mar 16 13:37 /tmp/unpacked
```

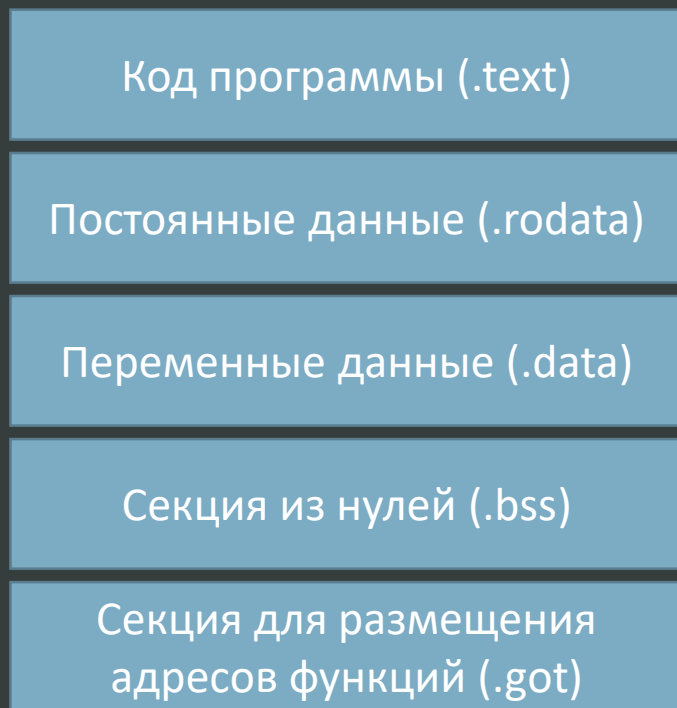
Распаковка упакованного кода

- В случае, если используется самый сложный подход (с mmar), можно скорее готовить о том, что мы имеем дело с самомодифицирующимся кодом (который просто модифицирует всю программу)
 - Таким образом, описанный далее способ будет применим и к обычному самомодифицирующемуся коду
- Общего алгоритма действий в этом случае не существует, поскольку код может, например, распаковываться постранично (не весь сразу, а по требованию)
 - Этот подход использовался в некоторых протекторах для Windows
- Тем не менее, в самом простом варианте существует очень неплохой способ понять, что происходит в программе: снять с нее дампы памяти процесса

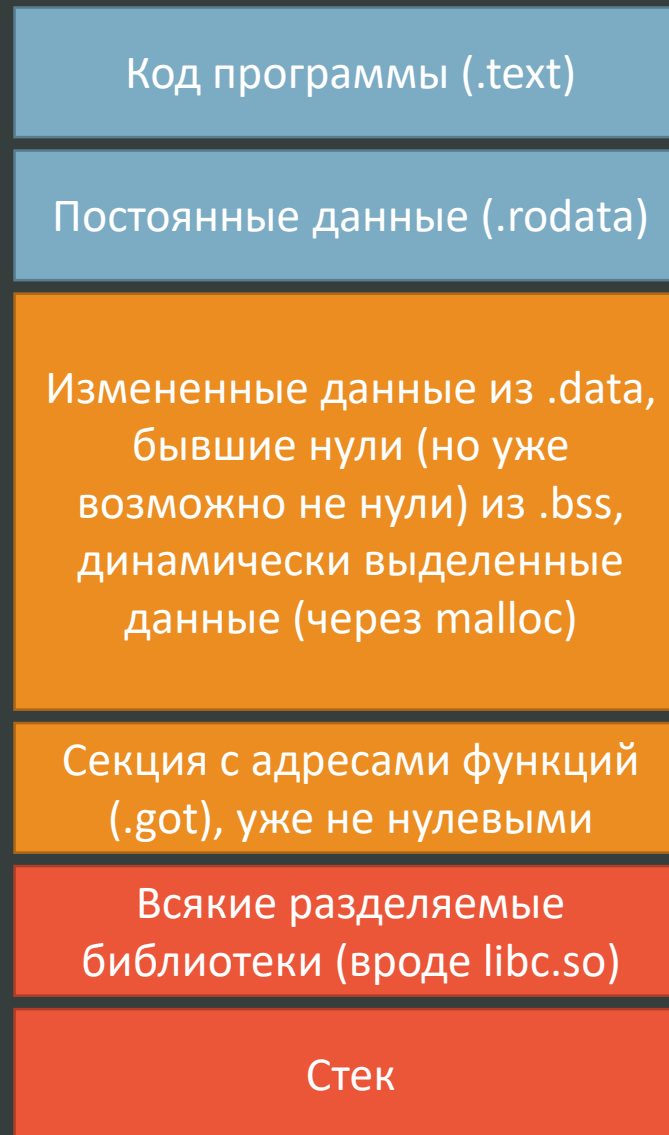
Дамп памяти процесса

- Дамп памяти процесса – содержимое всей виртуальной памяти процесса на момент съемки
 - Соответственно, весь распакованный исполняемый код также будет доступен для анализа
- У этого подхода есть неоспоримое преимущество: достаточно легко застать программу врасплох
 - Если в случае с отладкой у программы есть какие-то шансы от нее защититься, то при дампе процесса – очень вряд ли (особенно если запускать программу в виртуальной машине и снимать дамп вместе с ней)
- Есть и огромный недостаток: восстановление первоначального исполняемого файла в этом случае затруднительно
 - Испорчены таблицы импорта, может отсутствовать часть необходимой информации для загрузки библиотек и инициализации программы
 - Секции `.data` и `.bss` могут быть модифицированы, плюс есть лишние сегменты (вроде стека)

Дамп памяти процесса



Типичный ELF-файл



Дамп того же ELF-файла

Дамп памяти процесса

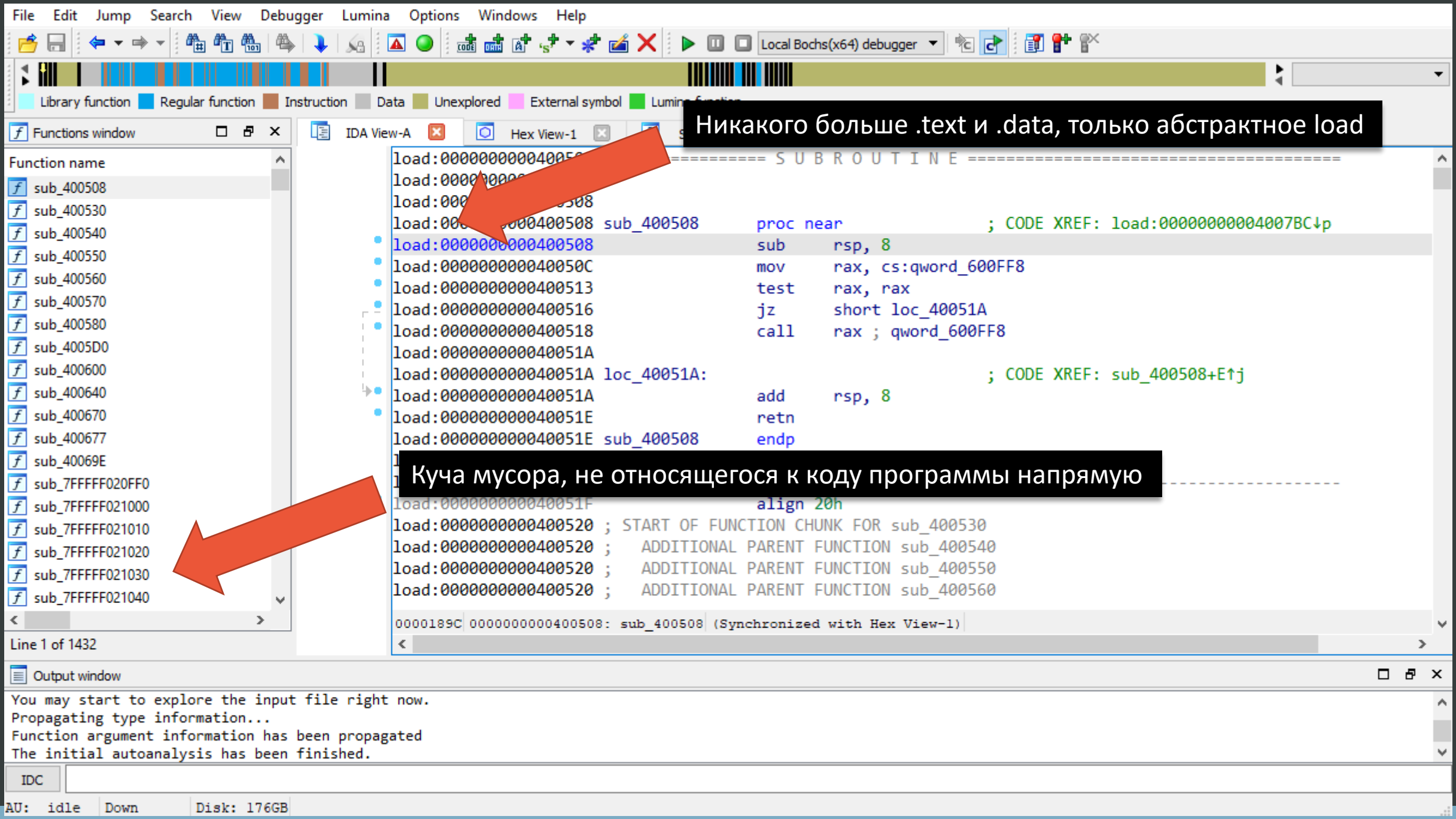
- Благодаря тому, что секции, связанные с кодом, остаются корректными, исследование поведения программы остается вполне возможным
- Еще одним удобством подхода является возможность сделать снимок программы в любой момент времени после того, как она уже запустилась
 - Это позволяет не ловить определенный момент в программе с отладчиком

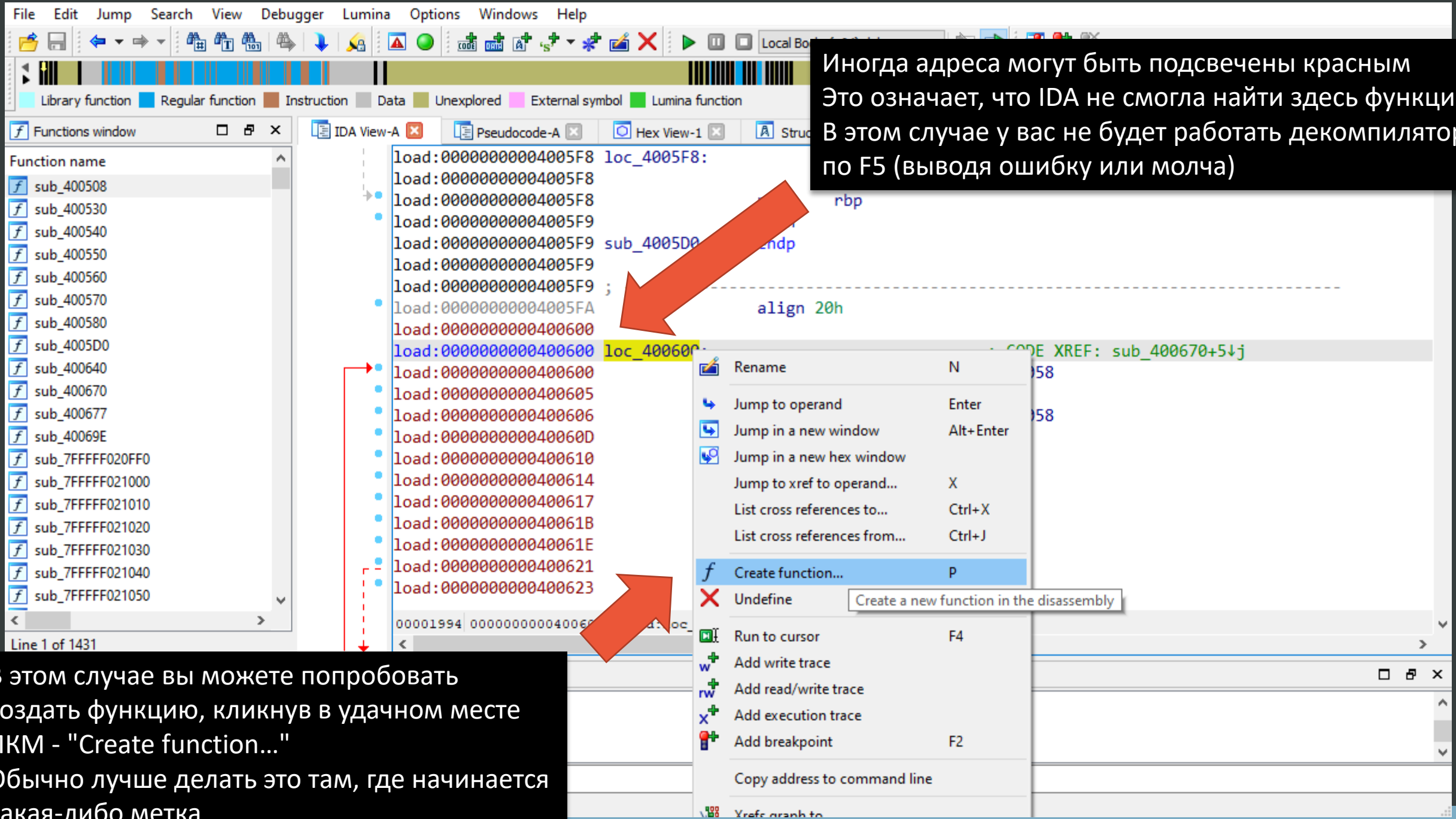
Дамп памяти процесса и GDB

- У GDB есть возможность создать дамп текущего процесса командой `generate-core-file <имя_файла>` (сокращенно `gcore`)
 - Если имя не указывать, файл дампа будет сохранен в `core.<pid>`
- Однако существует и более быстрый способ – можно просто вызвать команду `gcore <pid>`
 - Эта команда тоже является частью GDB, но заметно быстрее в использовании
- У процессов есть возможность прятать некоторые свои области от дампа и GDB по умолчанию их слушается
 - Отключить это поведение в GDB можно командами `"set dump-excluded-mappings on"` и `"set use-coredump-filter off"`
 - Для инструмента `gcore` достаточно просто передать дополнительный параметр `-a`

Открываем дампы памяти процесса

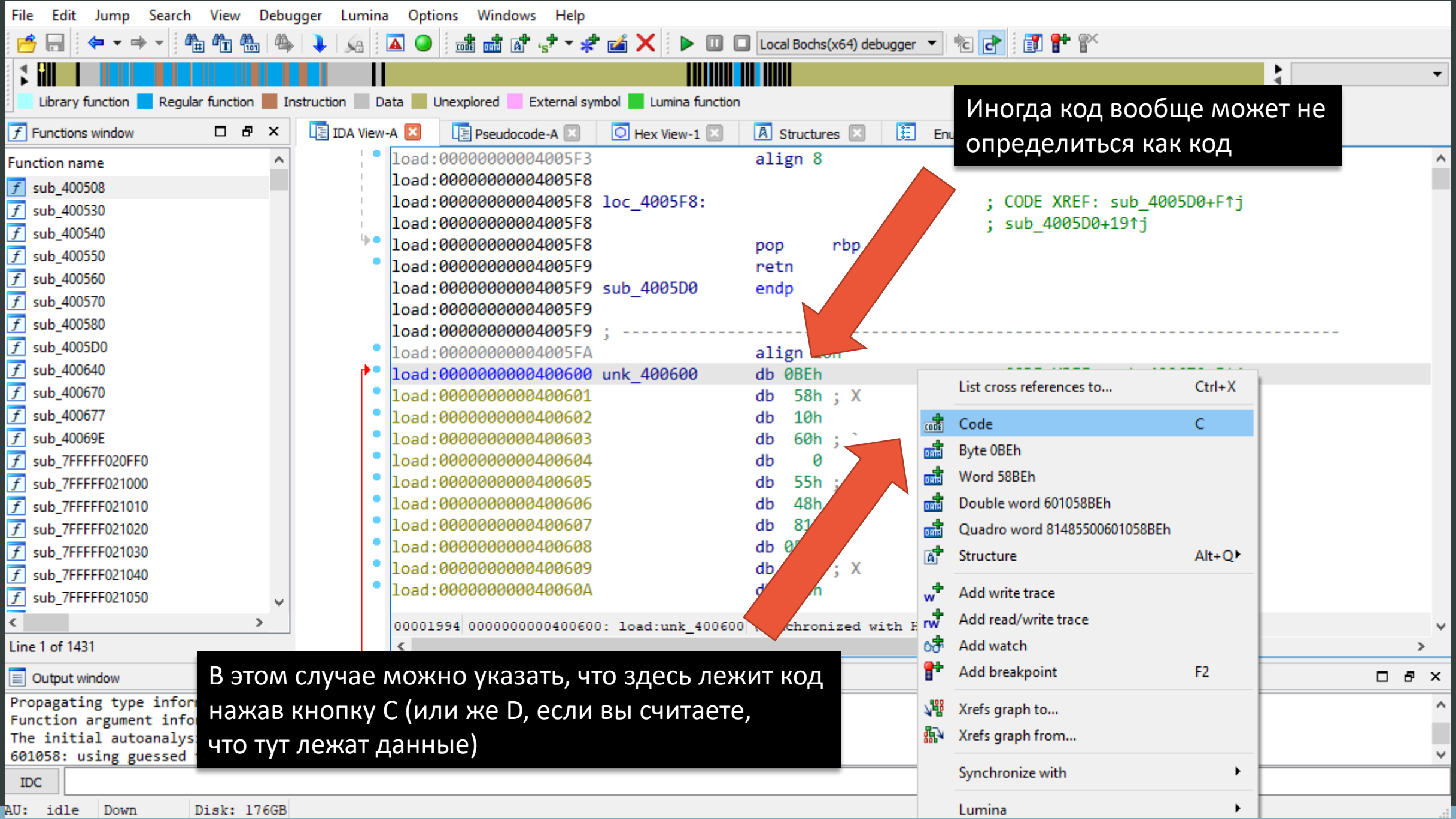
- Дампы можно спокойно открывать и в IDA и в Ghidra
 - Файл дампа является обычным ELF-файлом, поэтому запросто открывается даже демо-версией IDA
- Важное замечание по поводу форматов дампов: дампы 64-битных процессов лучше создавать 64-битным GDB, а дампы 32-битных процессов – 32-битным
 - Если задампить 32-битный процесс 64-битным GDB, то созданный файл будет формата ELF64. В результате IDA предложит вам открыть его своей 64-битной версией, которая откажется декомпилировать 32-битный код





Иногда адреса могут быть подсвечены красным
Это означает, что IDA не смогла найти здесь функцию
В этом случае у вас не будет работать декомпилятор
по F5 (выводя ошибку или молча)

В этом случае вы можете попробовать
создать функцию, кликнув в удачном месте
ПКМ - "Create function..."
Обычно лучше делать это там, где начинается
какая-либо метка



Иногда код вообще может не определиться как код

В этом случае можно указать, что здесь лежит код нажав кнопку C (или же D, если вы считаете, что тут лежат данные)

- List cross references to... Ctrl+X
- CODE** Code C
- DATA Byte 0BEh
- DATA Word 58BEh
- DATA Double word 601058BEh
- DATA Quadro word 81485500601058BEh
- Structure Alt+Q
- Add write trace
- Add read/write trace
- Add watch
- Add breakpoint F2
- Xrefs graph to...
- Xrefs graph from...
- Synchronize with
- Lumina

Как не заблудиться в дампе

- Можно видеть, что в дампе гораздо больше данных, чем можно просмотреть с комфортом
- Чтобы не потеряться в дампе в IDA, можно использовать следующие возможности:
 - Поиск по тексту (Alt-T), может помочь вам найти какую-нибудь характерную строку (вроде "Incorrect license key")
 - Также можно использовать поиск по байтам (Alt-B) если IDA не смогла распознать какую-либо строку (чтобы не вводить ее в hex-представлении, нужно использовать кавычки, например введя "Incorrect license key" вместе с кавычками)
 - Когда вы нашли строку, можно использовать кнопку X чтобы посмотреть весь список ссылок на нее из кода (таким образом найдя сам код проверки)
 - Также важно помнить, что во многих случаях разделяемые библиотеки (вроде libc.so) загружаются значительно выше в памяти чем сама программа (по адресам вроде 7FF...), а вот адреса вида 0x400000 (x86-64) присущи как раз программам

Лениво реверсим с дампами

- Если вас интересуют данные, а не исполняемый код, дампы особенно хороши
- Можно применить к дампу команду `strings`, она найдет печатные строки в дампе
 - По умолчанию эта утилита использует особенности ASCII, печатные символы там лежат в довольно узком диапазоне (как минимум старший бит нулевой)
 - Она поддерживает и другие кодировки с параметром `--encoding` (или `-e`), например, для UTF-16 достаточно указать `-e I`
- Очень хорошо дампы сочетаются с языками программирования со сборщиком мусора: там память может освободиться не сразу, поэтому в дампы могут попасть и различные временные данные
 - Например какие-нибудь секретные URL, которые программа посещает, и тексты запросов, если вы подловите
- Таким образом можно «реверсить» даже ненативные языки программирования: в данные у них все равно обычно хранятся «как есть»

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