Дополнительная лекция 2

ПРО РЕВЕРС-ИНЖИНИРИНГ ПРОГРАММ ДЛЯ WINDOWS
И ДОПИСЫВАНИЕ КОДА В ИСПОЛНЯЕМЫЕ ФАЙЛЫ

rev. 3

Отличия реверса для Windows

Особенности Windows-реверса

- В целом процесс реверс-инжиниринга программ в Windows похож на линуксовый, особенно если не выходить за пределы статического анализа
  - Процессорные инструкции не зависят от операционной системы, так что декомпиляторы все так же работают
- Тем не менее, есть некоторые различия, которые мы рассмотрим в этой лекции:
  - Форматы исполняемых файлов
  - Соглашения о вызове
  - Системные вызовы
  - Инструменты для динамического анализа

Portable Executable

- Формат PE – коллега формата ELF из мира Linux
- Исполняемые файлы такого формата обычно имеют расширение .exe (вместо никакого), а разделяемые библиотеки – .dll (вместо .so)
  - Кстати, .NET-программы тоже имеют расширения .exe и .dll и также имеют корректные PE-заголовки
- Исторически является модификацией формата COFF
  - Сам COFF использовался в Unix, но в большинстве его вариантов был заменен форматом ELF

Отличия PE от ELF

- В PE есть только понятие секций, сегментов нет
  - Кстати, секции немного отличаются от тех, что в ELF
- В PE есть возможность поместить дополнительные метаданные, такие как ресурсы и сертификаты
  - Хотя в ELF, конечно, тоже можно добавить секцию с криптографической подписью (что позволяют некоторые инструменты), это не стандартизовано и почти не используется
- Все импортируемые функции в PE привязаны к конкретным библиотекам
  - В формате ELF для функций неизвестно, откуда они импортируются (если не включено какое-нибудь хитрое версионирование), поэтому если функция нашлась хоть в какой-нибудь библиотеке, она используется
  - Поэтому и использование фокусов вроде LD_PRELOAD крайне затруднено (вы все еще можете загрузить в процесс дополнительную библиотеку, но автоматически подменить функцию, импортируемую из системной библиотеки, уже не выйдет)

Отличия PE от ELF

- PE по умолчанию позиционно-зависим, в каждом исполняемом файле есть поле ImageBase, указывающее, куда исполняемый файл должен быть загружен
- Если же системе хочется загрузить файл по случайному адресу, чтобы реализовать рандомизацию адресного пространства (ASLR), то файл загружается как обычно, а потом все используемые абсолютные адреса исправляются в соответствии с relocation table
  - Причем это касается даже DLL, у них тоже есть предпочтительный адрес загрузки!
  - Впрочем, в 64-битных исполняемых файлах там практически пусто, потому что туда завезли относительную адресацию и позиционно-независимый код стало писать легко
- Чтобы превратить позиционно-независимый исполняемый файл в позиционно-зависимый, достаточно снять флаг "Dll can move" (IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE) в Optional Header – DLL Characteristics

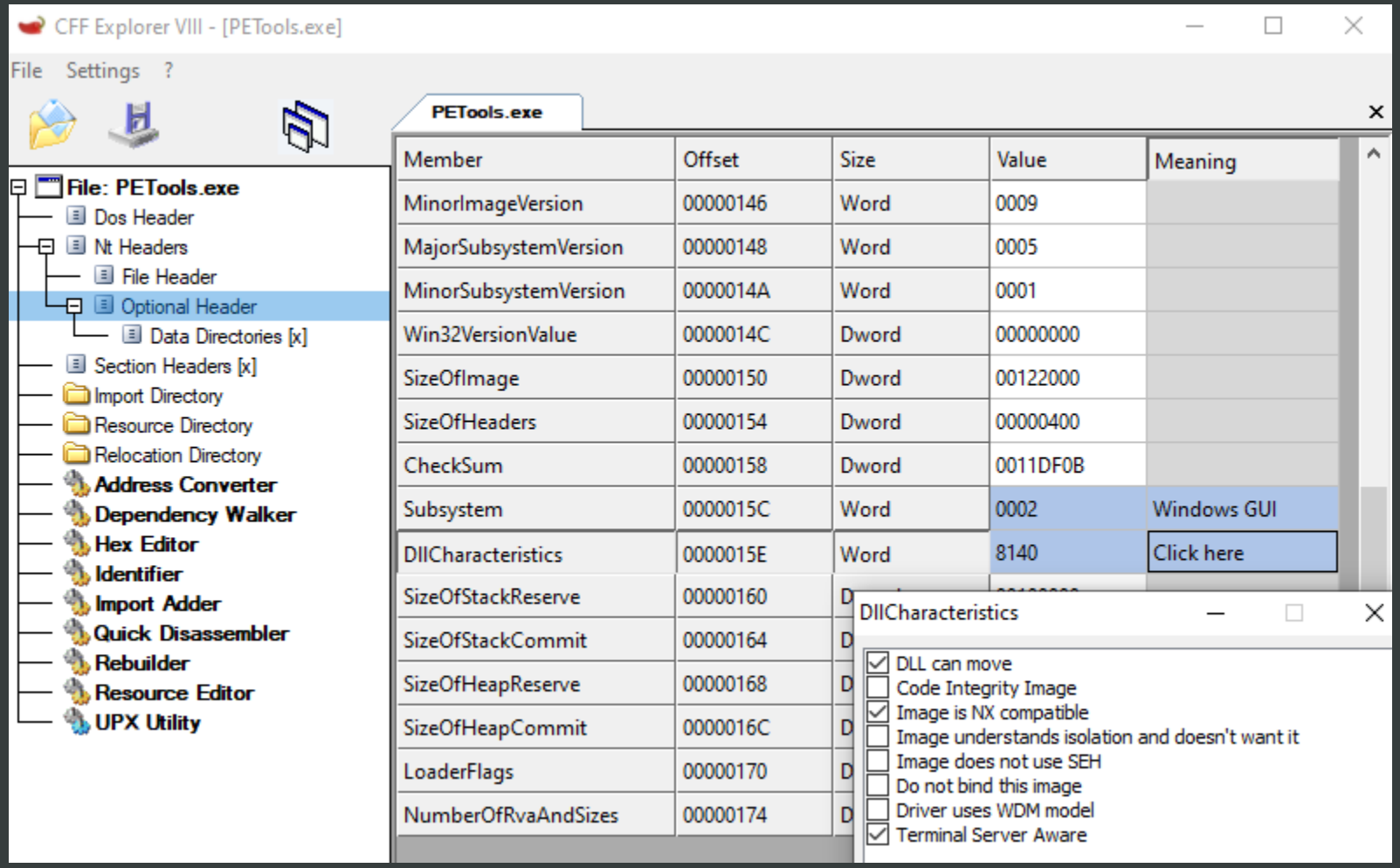
Типичные секции PE-файлов

Название в PE	Название в ELF	Назначение
.text	.text	Секция с исполняемым кодом
.data	.data	Секция с данными, доступными для чтения и записи
.rdata	.rodata	Секция с данными, доступными только для чтения
.bss	.bss	Как .data, но для нулей, не занимает места в исполняемом файле
.reloc	.rela.*	Таблица релокаций (relocation table), вряд ли вам пригодится
.idata / .edata	.dynsym	Секции с описаниями импортируемых и экспортируемых функций. Могут объединяться с .rdata
.rsrc	**нет**	Секция с ресурсами. К ресурсам относятся иконки приложения, курсоры, картинки, строки (в том числе с переводами), диалоги, меню и даже сочетания клавиш. Доступ к ресурсам осуществляется через WinAPI, в кроссплатформенных приложениях ресурсы могут не использоваться

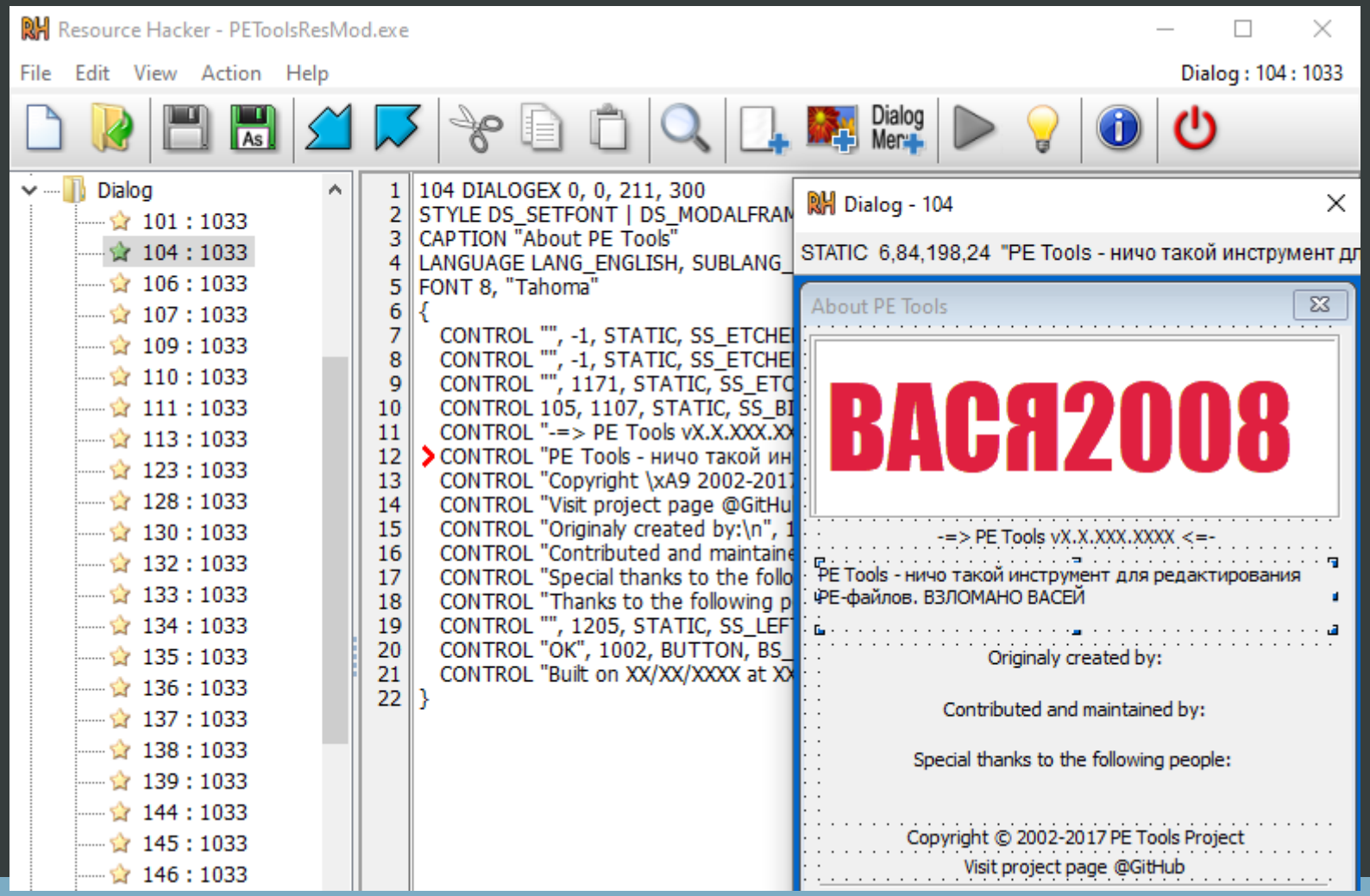
Чем работать с форматом PE

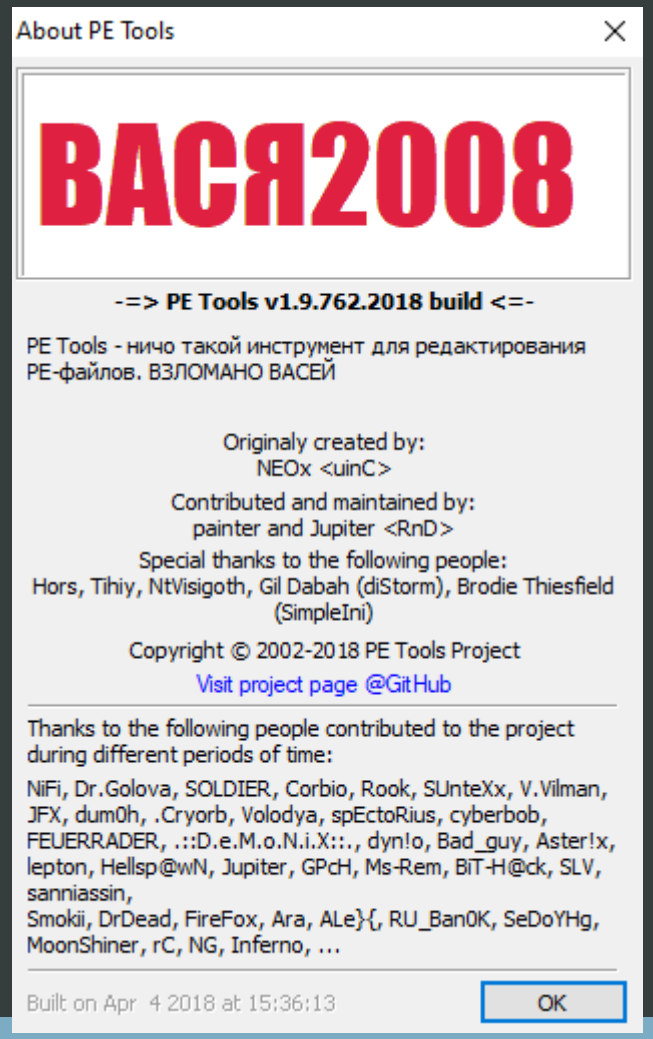
- PE Tools (https://github.com/petoolse/petools) – инструмент, позволяющий смотреть и редактировать все поля заголовка PE, а также секции и таблицы импорта и экспорта
- CFF Explorer (https://ntcore.com/files/CFF_Explorer.zip) – набор функций аналогичен PE Tools, но также есть редактор ресурсов
- Resource Hacker (http://www.angusj.com/resourcehacker/) – чистый редактор ресурсов, лучше всего подходит для этой цели: позволяет работать не только со строковыми ресурсами и изображениями, но и рендерить и редактировать диалоги
- Detect It Easy (https://github.com/horsicq/Detect-It-Easy/) – инструмент, позволяющий определить, использовался ли какой-нибудь пакер или проектор при создании исполняемого файла, впрочем посмотреть заголовок PE он также позволяет

Чем работать с форматом PE



Чем работать с форматом PE





Соглашения о вызове

- Соглашения о вызове в Windows отличаются от используемых в Linux, более того, в 32-битном режиме их использовалось сразу несколько
- Одним из основных соглашений о вызове наряду с cdecl в 32-битном Windows является stdcall, который во всем схож с cdecl кроме одного: стек чистит вызываемая функция, а не вызывающая
  - Кстати, стек чистится как раз при помощи инструкции RET N, позволяющей последним указом (фактически уже после возврата) снять N байт со стека
  - Это соглашение используется повсеместно в WinAPI
  - Для реверса, однако, удобно считать, что эти соглашения практически одинаковы
  - Как нетрудно догадаться, функции с переменным количеством аргументов такое соглашение использовать не могут (обычно оно заменяется на cdecl)
- Также использовалось соглашение fastcall, где аргументы складываются сначала в регистры ECX и EDX, а уже потом на стек. В остальном оно схоже с stdcall

Соглашения о вызове

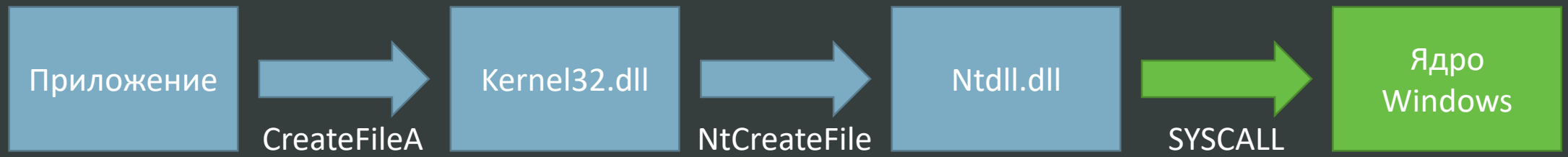
- В 64-битном Windows оно в основном одно, т.н. "Microsoft x64 calling convention"
- Аргументы передаются в регистрах RCX, RDX, R8, R9 (или XMM0-XMM3 для дробных, если функция не с переменным числом аргументов, причем максимум все равно 4 аргумента в регистрах), кто не влез – далее на стеке справа налево
  - В Linux – регистры RDI, RSI, RDX, RCX, R8, R9. Также в Linux дробные аргументы передаются полностью отдельно от целочисленных, здесь же используется или целый регистр, или дробный (т.е. RCX / XMM0, RDX / XMM1, и т.д.)
  - Как и в Linux, this в C++ просто является первым аргументом
  - Как и в Linux, стек чистит вызывающая функция
- Возвращаемое значение, если оно 64 бита и меньше, помещается в регистр RAX (причем если оно меньше, старшие биты могут не обнуляться). Если больше или дробное – помещается в регистр XMM0
  - В Linux для 128-битных чисел используется пара RDX:RAX

Соглашения о вызове

- Также существует соглашение vectorcall, которое мало отличается от предыдущего и имеет смысл для передачи больших аргументов (более 64 бит) или дробных
- В обычном соглашении о вызове большие аргументы, например, 128-битные числа, передаются в виде указателей на память, которую выделяет вызывающая функция
- В vectorcall, в свою очередь, для этого могут использоваться регистры XMM0-XMM5 или даже YMM0-YMM5
- Возвращаемые значения могут быть помещены не только в RAX и XMM0, но и несколько регистров, вплоть до XMM0:XMM3 или YMM0:YMM3

Системные вызовы

- В отличие от Linux, системные вызовы Windows не принято использовать напрямую, причем настолько не принято, что их номера постоянно меняются
  - Взглянуть на то, как это выглядит, можно здесь https://j00ru.vexillium.org/syscalls/nt/64/ или здесь https://hfiref0x.github.io/NT10_syscalls.html
  - Сами же вызовы осуществляются при помощи INT 0x2e (вместо int 0x80 в 32-битном Linux) или SYSCALL (как в 64-битном Linux)
- Вместо этого, системные вызовы принято использовать через WinAPI (официальный способ) или хотя бы NtAPI (менее официальный способ)
  - Эти способы представлены библиотеками kernel32.dll и ntdll.dll соответственно, которые автоматически загружаются во все запускаемые процессы



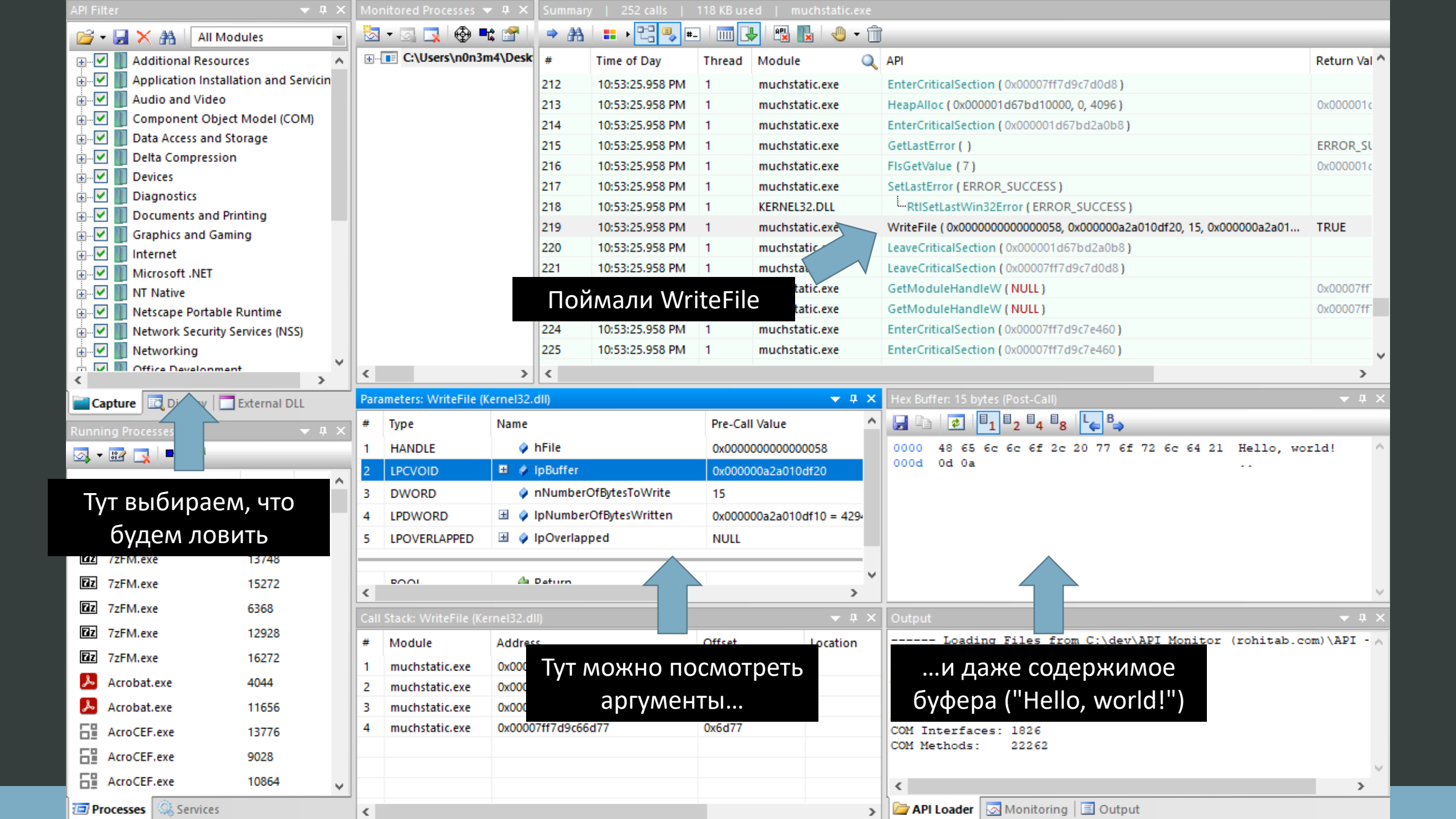
Отслеживание системных вызовов

- В Windows нет простого инструмента вроде strace, позволяющего отследить все системные вызовы
- Вызовы, выполняемые напрямую поймать вообще очень проблематично
  - Порт GDB под MinGW не поддерживает catch syscall, и он такой не один
  - Эта проблема и (такие себе) методы борьбы описываются в статье https://jmpesp.me/malware-analysis-syscalls-example/
- Одно из решений – искать в программе все входы SYSCALL и INT 0x2e и ставить точки останова на них
  - Не очень надежно, потому что динамически созданный код вы не поймаете
- Другое решение – поставить точку останова на обработчик вызова прямо в ядре
- Если у вас возникнет необходимость ловить такие программы, можно попробовать HyperDbg (https://hyperdbg.org/), он вроде бы это умеет

Отслеживание «системных вызовов»

Впрочем, можно попробовать ловить вызовы WinAPI прямо в разделяемых библиотеках – для этого существует множество инструментов:

- API Monitor (http://www.rohitab.com/apimonitor) – старый, но не бесполезный инструмент для перехвата WinAPI, по-видимому застрявший в вечной альфа-версии. Обладает большим списком API для перехвата, позволяет удобно смотреть передаваемые аргументы
- Process Monitor (https://docs.microsoft.com/en-us/sysinternals/downloads/procmon) – утилита из состава SysInternals от Microsoft, обладает не столь гибкими настройками, ловит не все (например не поймал печать в консоль), зато обновляется и широко известна (хотя это может быть и минусом)
- WinAPIOverride (http://jacquelin.potier.free.fr/winapioverride32/) – инструмент, делающий акцент на модификации перехватываемых функций. Тем не менее, может использоваться и для наблюдения аналогично API Monitor (но интерфейс не так хорош и требуется вручную указать, что именно вы хотите перехватывать)

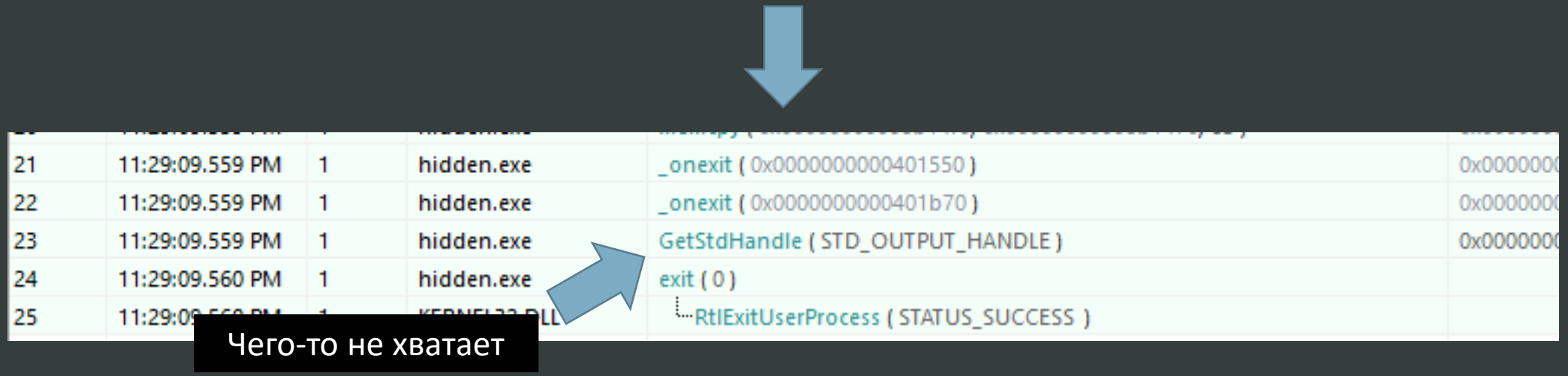

API Filter
All Modules
Additional Resources
Application Installation and Servicin
Audio and Video
Component Object Model (COM)
Data Access and Storage
Delta Compression
Devices
Diagnostics
Documents and Printing
Graphics and Gaming
Internet
Microsoft .NET
NT Native
Netscape Portable Runtime
Network Security Services (NSS)
Networking
Capture
External DLL
Monitored Processes
C:\Users\n0n3m4\Desk
Summary | 252 calls | 118 KB used | muchstatic.exe
Time of Day Thread Module API Return Val
212 10:53:25.958 PM 1 muchstatic.exe EnterCriticalSection (0x00007ff7d9c7d0d8)
213 10:53:25.958 PM 1 muchstatic.exe HeapAlloc (0x000001d67bd10000, 0, 4096)
214 10:53:25.958 PM 1 muchstatic.exe EnterCriticalSection (0x000001d67bd2a0b8)
215 10:53:25.958 PM 1 muchstatic.exe GetLastError ( )
216 10:53:25.958 PM 1 muchstatic.exe FlsGetValue (7)
217 10:53:25.958 PM 1 muchstatic.exe SetLastError (ERROR_SUCCESS)
218 10:53:25.958 PM 1 KERNEL32.DLL RtlSetLastWin32Error (ERROR_SUCCESS)
219 10:53:25.958 PM 1 muchstatic.exe WriteFile (0x0000000000000058, 0x000000a2a010df20, 15, 0x000000a2a01... TRUE
220 10:53:25.958 PM 1 muchstatic.exe LeaveCriticalSection (0x000001d67bd2a0b8)
221 10:53:25.958 PM 1 muchstatic.exe LeaveCriticalSection (0x00007ff7d9c7d0d8)
GetModuleHandleW (NULL)
GetModuleHandleW (NULL)
224 10:53:25.958 PM 1 muchstatic.exe EnterCriticalSection (0x00007ff7d9c7e460)
225 10:53:25.958 PM 1 muchstatic.exe EnterCriticalSection (0x00007ff7d9c7e460)
Поймали WriteFile
Parameters: WriteFile (Kernel32.dll)
Type Name Pre-Call Value
1 HANDLE hFile 0x0000000000000058
2 LPCVOID lpBuffer 0x000000a2a010df20
3 DWORD nNumberOfBytesToWrite 15
4 LPDWORD lpNumberOfBytesWritten 0x000000a2a010df10 = 429
5 LPOVERLAPPED lpOverlapped NULL
Hex Buffer: 15 bytes (Post-Call)
0000 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21 Hello, world!
000d 0d 0a ..
Running Processes
Тут выбираем, что будем ловить
7zFM.exe 13748
7zFM.exe 15272
7zFM.exe 6368
7zFM.exe 12928
7zFM.exe 16272
Acrobat.exe 4044
Acrobat.exe 11656
AcroCEF.exe 13776
AcroCEF.exe 9028
AcroCEF.exe 10864
Processes
Services
Call Stack: WriteFile (Kernel32.dll)
Module Address Offset Location
1 muchstatic.exe 0x000
2 muchstatic.exe 0x000
3 muchstatic.exe 0x000
4 muchstatic.exe 0x00007ff7d9c66d77 0x6d77
Тут можно посмотреть аргументы...
Output
------ Loading Files from C:\dev\API Monitor (rohitab.com)\API -
COM Interfaces: 1826
COM Methods: 22262
...и даже содержимое буфера ("Hello, world!")
API Loader
Monitoring
Output

Прячем свои системные вызовы

- Соответственно, если вам очень нужно спрятать свои системные вызовы от мониторинга, можно вызвать их напрямую
  - Это может сбить с толку даже антивирусное ПО, а не только исследователей
- Для этого есть отличный инструмент SysWhispers2 (https://github.com/jthuraisamy/SysWhispers2), позволяющий сгенерировать заголовочные файлы для более-менее кроссплатформенного вызова системных вызовов напрямую
- Кстати, не обязательно осуществлять так все вызовы – иначе исследователь может расстроиться, не найдя ничего в выводе инструментов наподобие API Monitor

```
// python3 syswhispers.py -f NtWriteFile -o syscalls
// nasm -f win64 -o syscallsstubs.x64.o syscallsstubs.x64.nasm
// x86_64-w64-mingw32-gcc -s -m64 hidden.c syscalls.c syscallsstubs.x64.o -o hidden.exe

#include <windows.h>
#include "SysWhispers2/syscalls.h"
int main() {
        IO_STATUS_BLOCK isb;
        HANDLE h = GetStdHandle(STD_OUTPUT_HANDLE);
        // This call will be invisible
        NtWriteFile(h, NULL, NULL, NULL, &isb, "Hello, world!\n", 14, NULL, NULL);
}
```



Отладка в Windows

Отладка в Windows

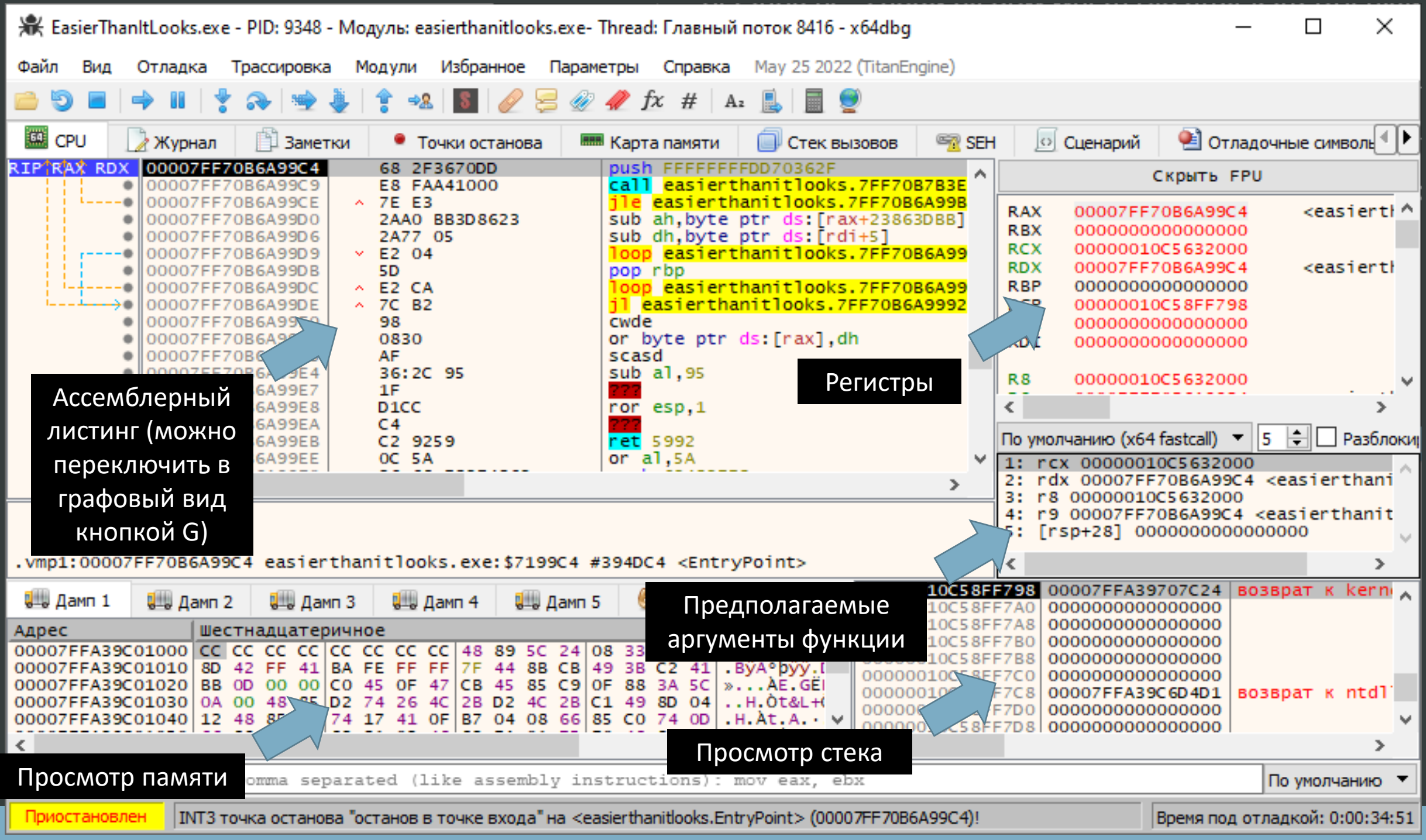
- GDB в Windows не слишком популярен, хотя и доступен для установки в составе MinGW / MSYS2 (https://www.msys2.org/)

Для реверс-инжиниринга в основном используются следующие отладчики:

- Встроенный в IDA отладчик, доступен через меню Debugger – Local Windows debugger. Благодаря интеграции с декомпилятором очень хорош для быстрой отладки процессов, не использующих антиотладку или пакеры
- OllyDbg (https://www.ollydbg.de/) – широко известный в прошлом отладчик, имеет множество возможностей и плагинов. К сожалению, является только 32-битным, а его разработка давно остановлена
- x64dbg (https://x64dbg.com/) – идейный наследник OllyDbg, обладает большинством его плюсов, и вдобавок активно поддерживается
- WinDbg – официальный отладчик от Microsoft, для самых тяжелых случаев

x64dbg

- Запустить отладчик можно зайдя в соответствующую папку (x64 или x32) и запустив x64dbg.exe (или x32dbg.exe), он является графическим приложением
- Для начала отладки достаточно перетащить исполняемый файл на окно отладчика и нажать кнопку запуска (F9)
- После этого программа будет запущена и остановится на «системной точке останова» – специальном месте где ntdll дает отладчику понять (при помощи INT 3), что собирается запустить процесс
  - Это отключается в настройках (Параметры - Параметры - События)
  - Кстати, там же можно включить точку останова на завершение программы
- После нажатия кнопки F9 выполнение программы продолжится и вы окажетесь на настоящей точке входа
  - И теперь, если нажать F9 еще раз, программа начнет свое настоящее выполнение

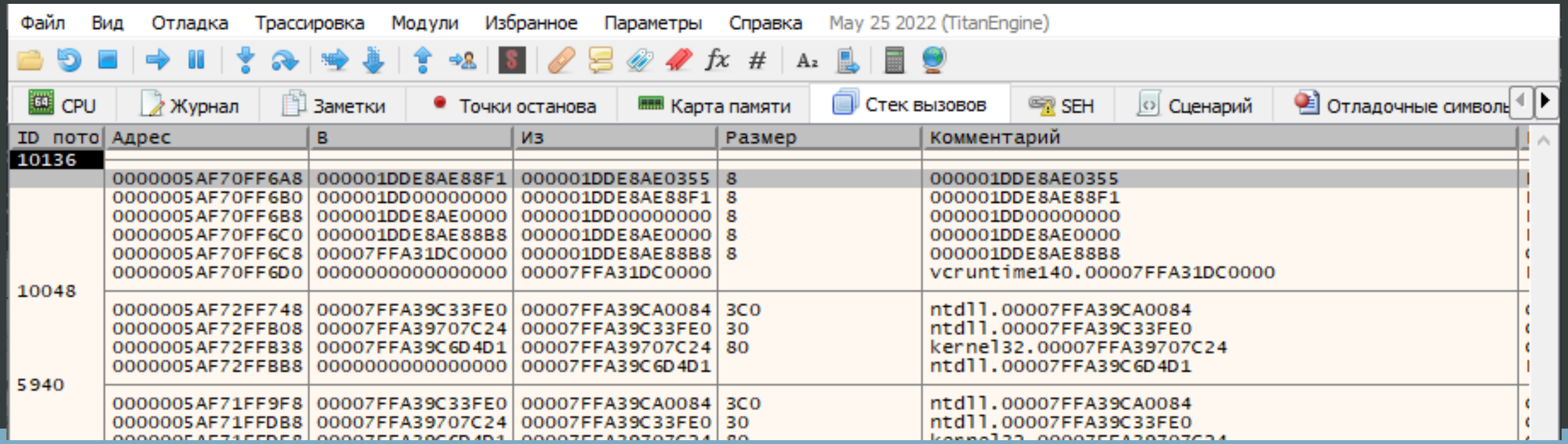

EasierThanItLooks.exe - PID: 9348 - Модуль: easierthanitlooks.exe- Thread: Главный поток 8416 - x64dbg
Файл Вид Отладка Трассировка Модули Избранное Параметры Справка May 25 2022 (TitanEngine)
CPU Журнал Заметки Точки останова Карта памяти Стек вызовов SEH Сценарий Отладочные символы
Скрыть FPU
Ассемблерный листинг (можно переключить в графовый вид кнопкой G)
Регистры
Предполагаемые аргументы функции
Просмотр стека
Просмотр памяти
Дамп 1 Дамп 2 Дамп 3 Дамп 4 Дамп 5
Адрес Шестнадцатеричное
По умолчанию (x64 fastcall)
Приостановлен
INT3 точка останова "останов в точке входа" на <easierthanitlooks.EntryPoint> (00007FF70B6A99C4)!
Время под отладкой: 0:00:34:51

x64dbg. Break

- Чтобы поставить точку останова, достаточно переместить курсор в нужное место и нажать F2
- Чтобы удобнее перемещаться в нужное место, можно воспользоваться сочетанием клавиш Ctrl-G
  - Кнопка *, в свою очередь, позволит вам вернуться к RIP
- Как и в GDB, на точку останова можно навесить условие, это можно сделать, перейдя на вкладку «точки останова» и нажав ПКМ - редактировать, или просто Shift-F2 (это можно сделать даже на вкладке листинга)
  - Синтаксис условий описан здесь: https://help.x64dbg.com/en/latest/introduction/ConditionalBreakpoint.html
- Поставить аппаратную точку останова можно при помощи ПКМ - Точка останова - «Поставить аппаратную точку останова на исключение» (да, перевод явно мог бы быть лучше)

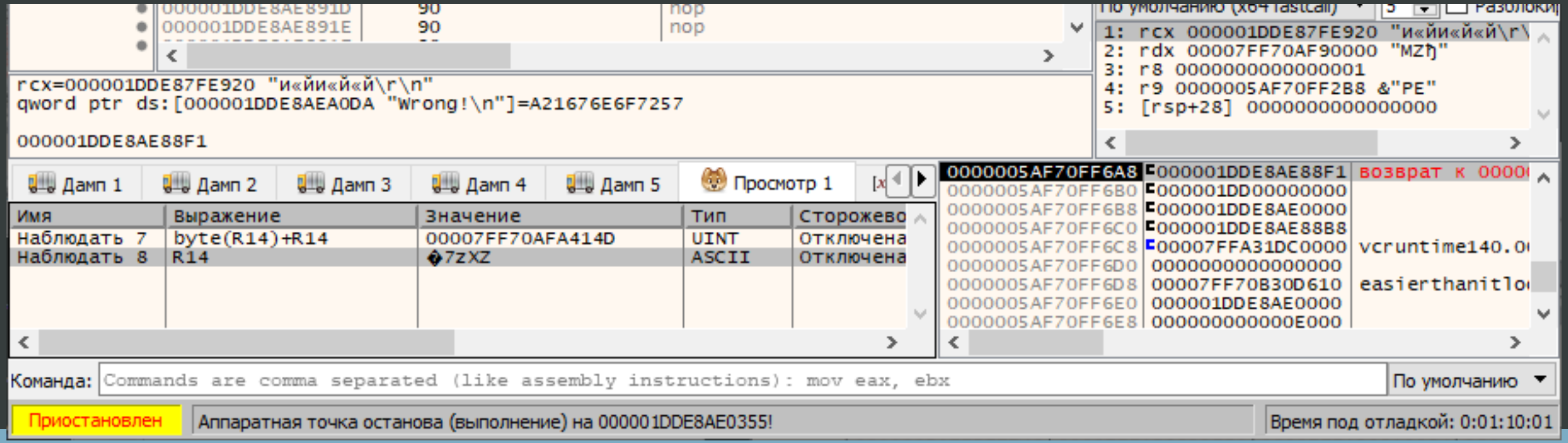
x64dbg. Backtrace

- Чтобы посмотреть стек вызовов и узнать, как вы попали в эту точку останова, можно заглянуть на вкладку Стек вызовов
  - Выводятся данные по всем потокам программы
  - Двойным щелчком можно перейти на место вызова



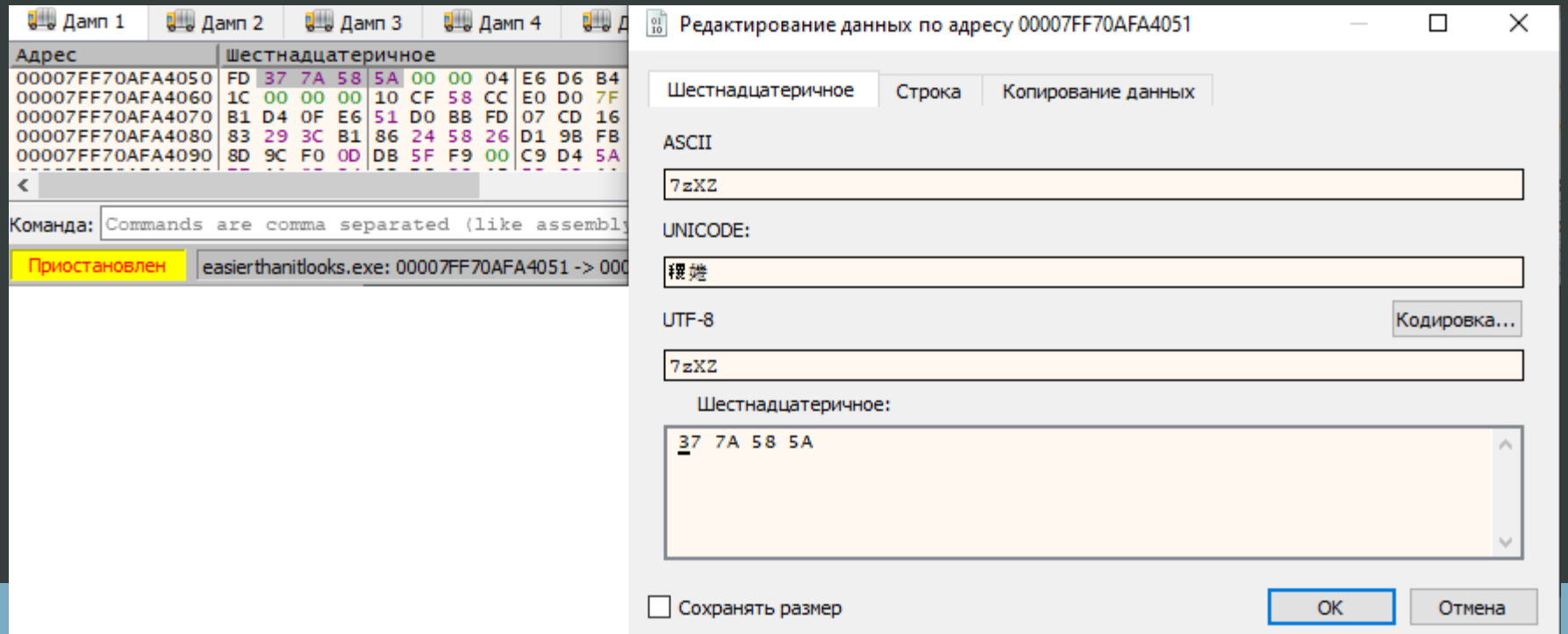
x64dbg. Print / Examine

- Выражения для наблюдений можно добавить на вкладке Просмотр под листингом
  - При помощи правой кнопки можно менять тип выражения, например, на строку
  - Синтаксис выражений и доступные функции можно изучить здесь: https://help.x64dbg.com/en/latest/introduction/Expression-functions.html



x64dbg. Set

- Редактировать регистры можно двойным нажатием на их значения
- Редактировать память на вкладке Дамп можно выделив блок памяти и нажав ПКМ - Двоичные операции - Редактировать (или Ctrl-E)
  - Там же их можно удобно скопировать на вкладке Копирование данных



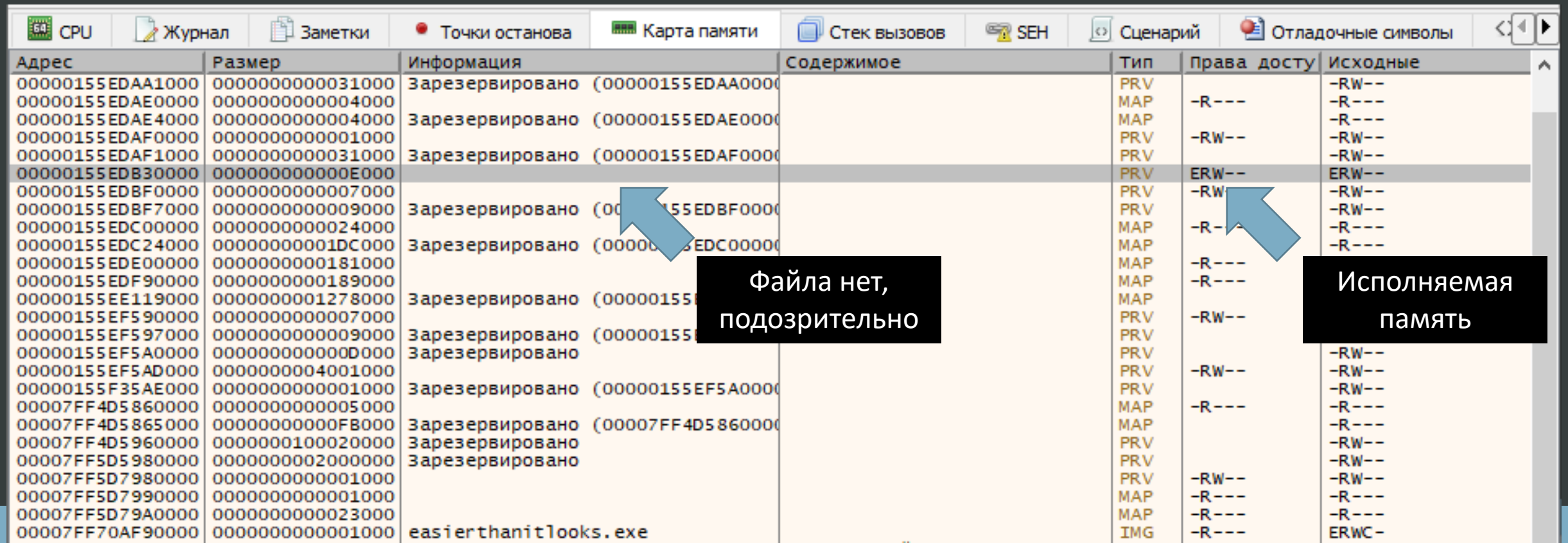
x64dbg. Управление исполнением

Управлять исполнением можно традиционными для отладчиков действиями:

- Выполнить / продолжить: в отличие от GDB для запуска и продолжения используется одна и та же кнопка F9, если вам нужно запустить программу заново (как при помощи Run в GDB), воспользуйтесь кнопкой Перезапустить
- Шаг с заходом / шаг с обходом (F7 / F8): аналогично stepi и nexti в GDB
- Выполнить до возврата (Ctrl-F9): аналогично finish в GDB
- Также, как и в GDB, можно отправить процессор по любому адресу исполнения, нажав ПКМ - Set RIP Here

x64dbg. Смотрим виртуальную память

- В Windows нет procfs, а значит и в файл /proc/…/maps заглянуть нельзя
- К счастью, на вкладке Карта памяти можно найти ту же самую информацию:

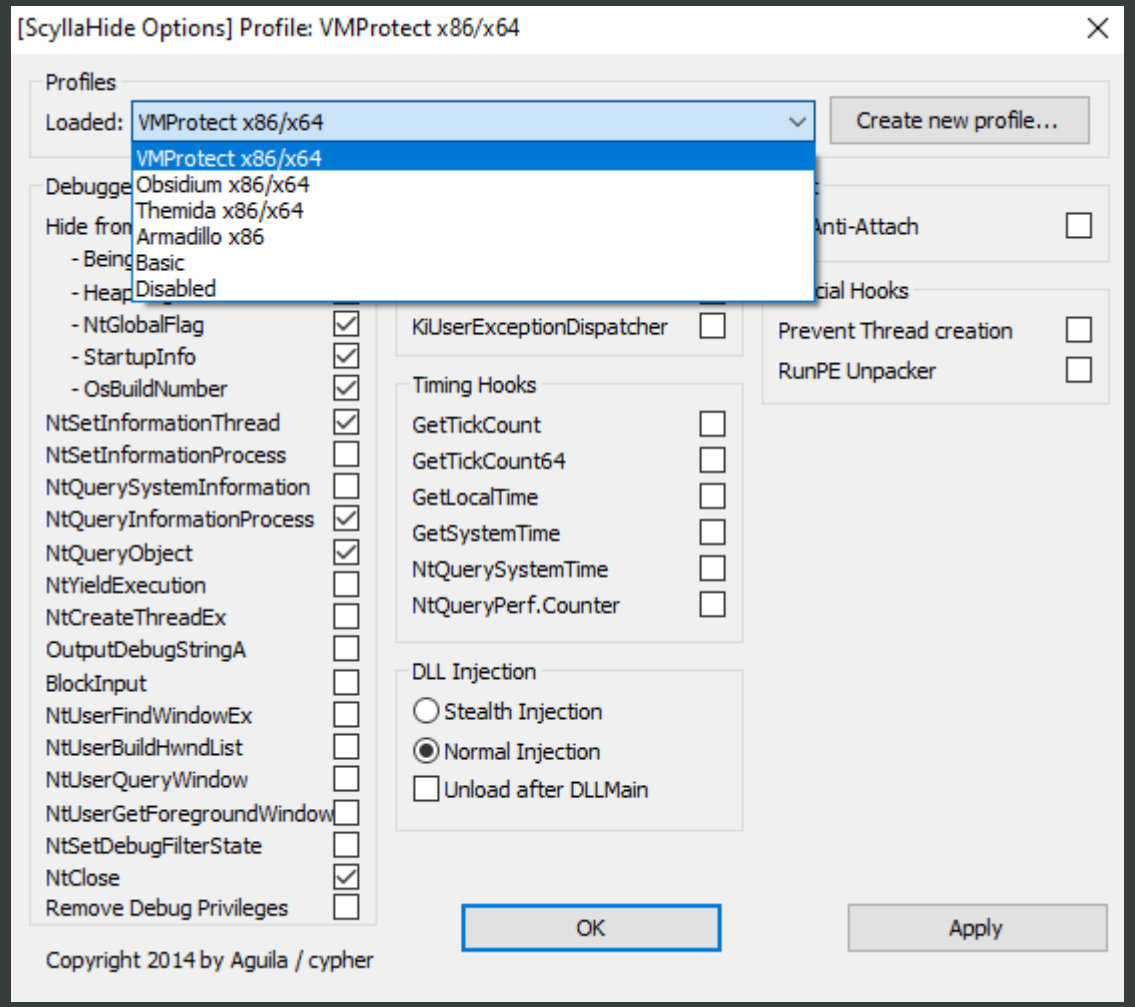


Специфические особенности отладки в Windows

- В Windows есть множество способов антиотладки, бороться с которыми вручную, как можно было делать в Linux, не очень рационально
- Поэтому существует множество плагинов антиантиотладки, вот некоторые из них:
  - ScyllaHide (https://github.com/x64dbg/ScyllaHide) – антиантиотладочный плагин, работающий в режиме пользователя, имеет плагины для OllyDbg, x64dbg (а также его можно использовать с любым отладчиком вручную), относительно прост в настройке и использовании
  - TitanHide (https://github.com/mrexodia/TitanHide) – антиантиотладочный плагин режима ядра, достаточно сложен в настройке (например, требует выключения проверки подлинности Windows), поэтому стоит использовать только если ScyllaHide не хватило, также имеет плагин для x64dbg
- Еще один оригинальный вариант заставить антиотладку отстать – отлаживать программу под Wine (если она, конечно, там запустится)
  - Эмулируемый WinAPI очень вряд ли расскажет о линуксовом отладчике

x64dbg. ScyllaHide

- Чтобы ScyllaHide заработал, необходимо зайти в Модули - ScyllaHide - Options и выбрать подходящий вам профиль и настройки
- Перед этим можно попробовать при помощи статического анализа и инструментов вроде Detect It Easy выяснить, какой пакер используется

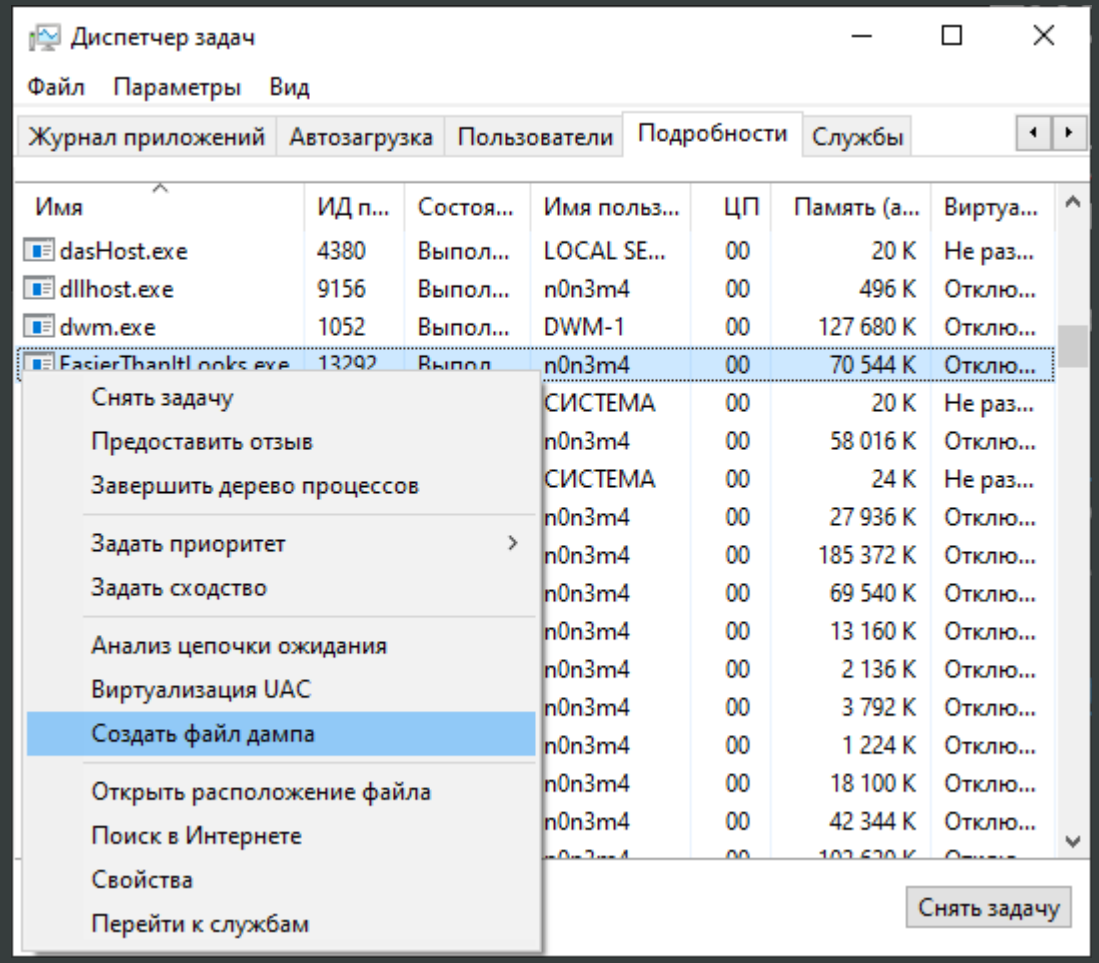


Снимаем дамп процесса

- Снятие дампа запакованной программы – достаточно непростой процесс, особенно если хочется, чтобы полученный PE-файл можно было запускать
  - Для этого необходимо найти начальную точку входа программы, а также таблицу импортов
- Для дампа из x64dbg можно использовать плагин Scylla (из комплекта поставки) или классический плагин OllyDumpEx (https://low-priority.appspot.com/ollydumpex/)
  - Эти плагины позволяют как сохранять память, так и восстанавливать весь PE-файл, причем OllyDumpEx также позволяет сохранить отдельную область памяти под видом PE-файла, используя заголовок-пустышку (запускать его, конечно, будет нельзя, но хоть адреса будут корректные)

Снимаем дамп процесса

- К счастью, есть способ проще – просто воспользоваться диспетчером задач
- Полученный дамп можно открывать в IDA (но возможно вам понадобится установить WinDbg)
  - Впрочем, в бесплатной IDA такой файл не откроется, но можно использовать Ghidra в режиме без заголовка
  - IDA откроется в не очень удобном представлении отладки, чтобы перейти в нормальный режим статического анализа, можно нажать кнопку остановки программы (хотя она вроде и не запущена), а потом выбрать "All segments", чтобы перенести все содержимое памяти в файл базы данных IDA



Дописываем свой код в исполняемые файлы

Дописываем свой код

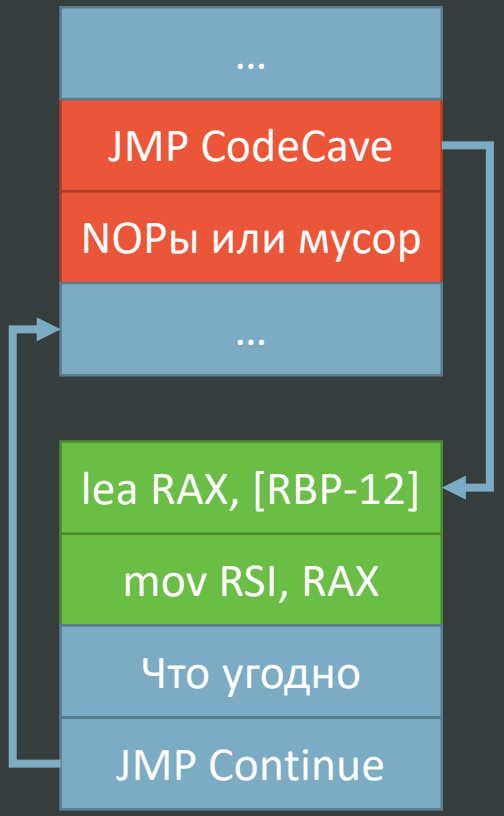
- Это уже не совсем про Windows, похожие подходы применимы почти везде
- Ранее в лекции про модификацию кода мы рассмотрели обычные способы патчинга, когда код меньше или равен по длине тому, который мы писали
- В этой лекции мы рассмотрим способы, применимые в более тяжелых случаях, когда нам нужно дописать больше кода, чем влазит в конкретное место программы:
  - Использование т.н. "Code Cave"
  - Добавление новых сегментов в программу

Code Cave

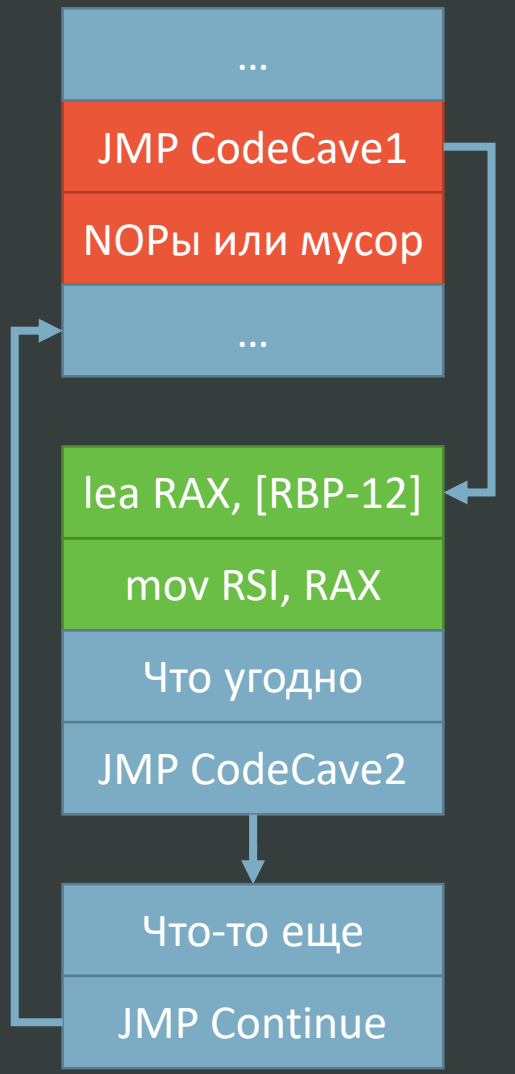
- Code Cave – места в исполняемой памяти процесса, которые забиты нулями и не используются, их можно использовать для размещения своего кода
- Патчинг выглядит следующим образом:
  - Ищем инструкцию, после которой хотим дописать код
  - Заменяем ее на JMP в Code Cave, куда копируем оригинальную инструкцию (или даже несколько, если наш JMP зацепил несколько)
  - Дописываем далее свой код
  - В конце пишем JMP обратно, на следующую инструкцию после скопированных
- При желании Code Cave можно собирать в паровозик, если весь ваш код в один не влез
- Кстати, никто не мешает вам взять в качестве Code Cave какой-нибудь не особо нужный программе код (скажем, какое-нибудь окно About :)

Пример использования Code Cave

Оригинальный код



Патчинг с использованием Code Cave



Паровозик из двух Code Cave

Пример использования Code Cave

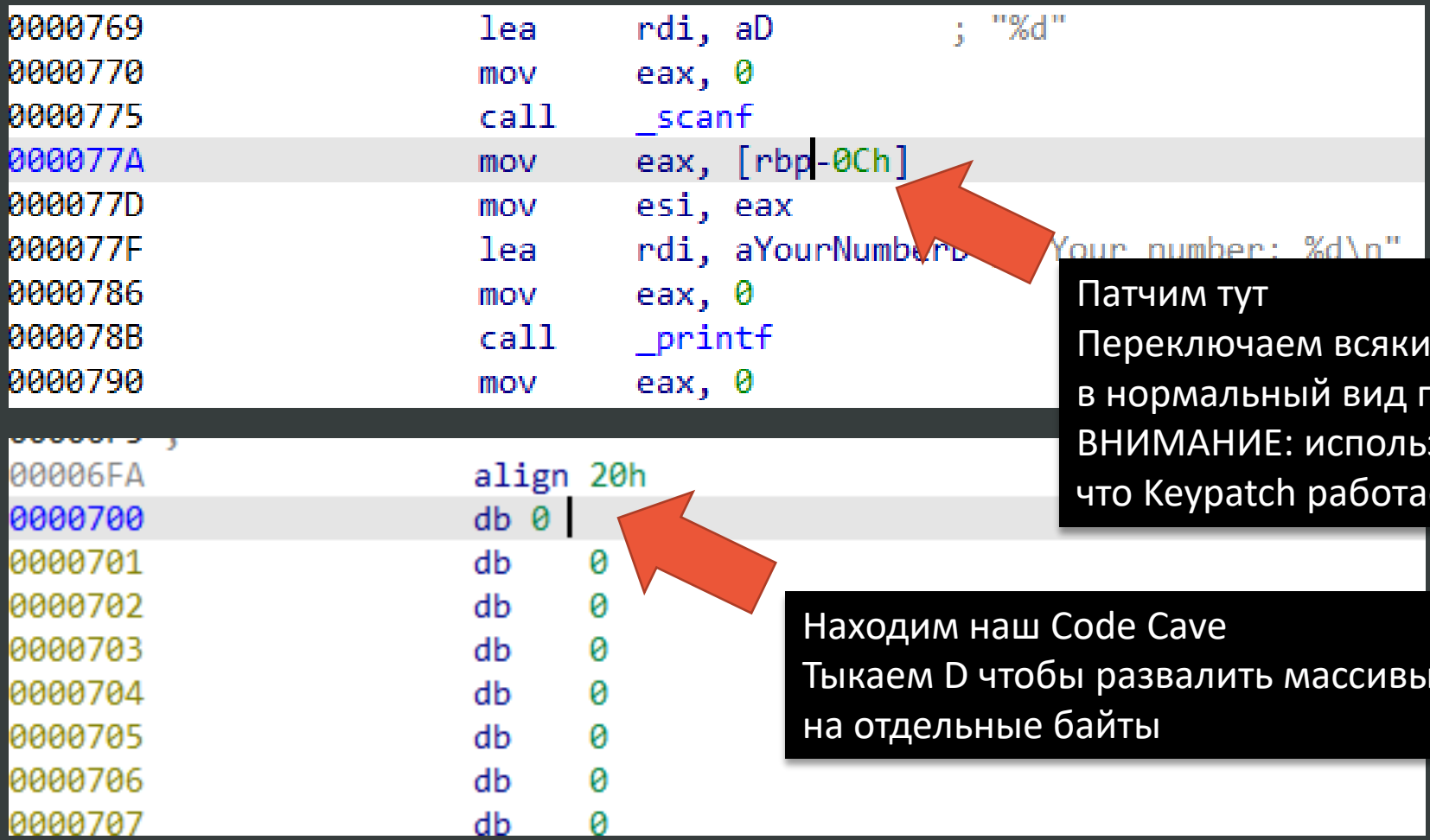
- Рассмотрим конкретный пример:

```
// Some code cave
asm(".quad 0, 0, 0, 0, 0, 0, 0, 0");

int main()
{
        int i;
        printf("Please enter number: ");
        scanf("%d", &i);
        printf("Your number: %d\n", i);
}
```

- Попробуем сделать так, чтобы число возводилось в квадрат

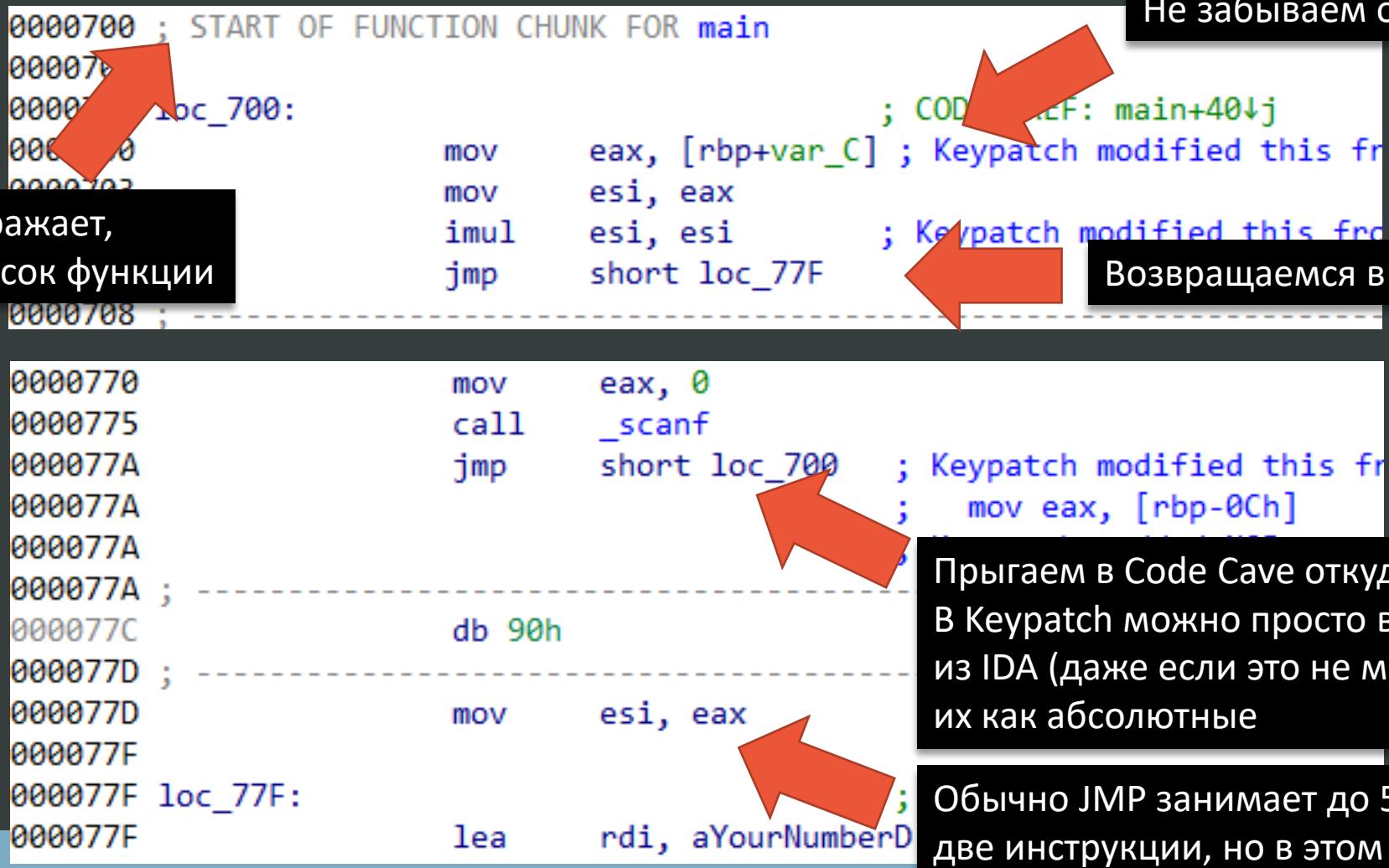
Пример использования Code Cave



Патчим тут
Переключаем всякие "[rbp+var_C]"
в нормальный вид правой кнопкой
ВНИМАНИЕ: используем hex отступы, потому что Keypatch работает с ними

Находим наш Code Cave
Тыкаем D чтобы развалить массивы на отдельные байты

Пример использования Code Cave



Не забываем оригинальные инструкции

IDA соображает,
что это кусок функции

Возвращаемся в функцию

Прыгаем в Code Cave откуда хотели
В Keypatch можно просто вводить адреса
из IDA (даже если это не метки), он воспримет
их как абсолютные

Обычно JMP занимает до 5 байт, поэтому я взял
две инструкции, но в этом примере хватило бы и одной

Пример использования Code Cave

- Запатченный код в декомпиляторе выглядит в полном соответствии с планом:

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [rsp+6h] [rbp-Ch] BYREF
  unsigned __int64 v5; // [rsp+Ah] [rbp-8h]

  v5 = __readfsqword(0x28u);
  printf("Please enter number: ");
  scanf("%d", &v4);
  printf("Your number: %d\n", (unsigned int)(v4 * v4));
  return 0;
}
```

- И выполняется соответственно:

```
Please enter number: 1337
Your number: 1787569
```

Добавление новых сегментов в программу

- Для некоторых форматов, например ELF, добавление сегментов может быть довольно проблематичной задачей из-за нехватки места уже для их описания
- Можно попробовать угнать какой-нибудь «ненужный» сегмент вроде GNU_RELRO или GNU_STACK (WSL 1, правда, такой фокус может не оценить)
  - Чтобы сделать это, просто смените тип сегмента на LOAD, добавьте байтов в хвост исполняемого файла и укажите это место содержимым сегмента
  - Можно воспользоваться редактором ELF, например https://github.com/horsicq/XELFViewer
- Также можно воспользоваться готовой библиотекой для таких манипуляций – LIEF
  - Как всегда, ее можно установить из pip при помощи pip install lief
  - Она, кстати, поддерживает не только ELF
  - И это далеко не единственная ее возможность

Добавление новых сегментов в программу

- Пример кода для добавления сегмента при помощи LIEF:

```
import lief
from lief import ELF
binary = lief.parse('nocave')
segment = ELF.Segment()
segment.type = ELF.SEGMENT_TYPES.LOAD
segment.flags = ELF.SEGMENT_FLAGS(ELF.SEGMENT_FLAGS.R | ELF.SEGMENT_FLAGS.X)
segment.content = [0] * 9000
segment.alignment = 0x1000
binary.add(segment, base=0x9900000)
binary.write('withcave')
```

- После этого добавленным сегментом можно пользоваться как обычно

Спасибо за внимание!