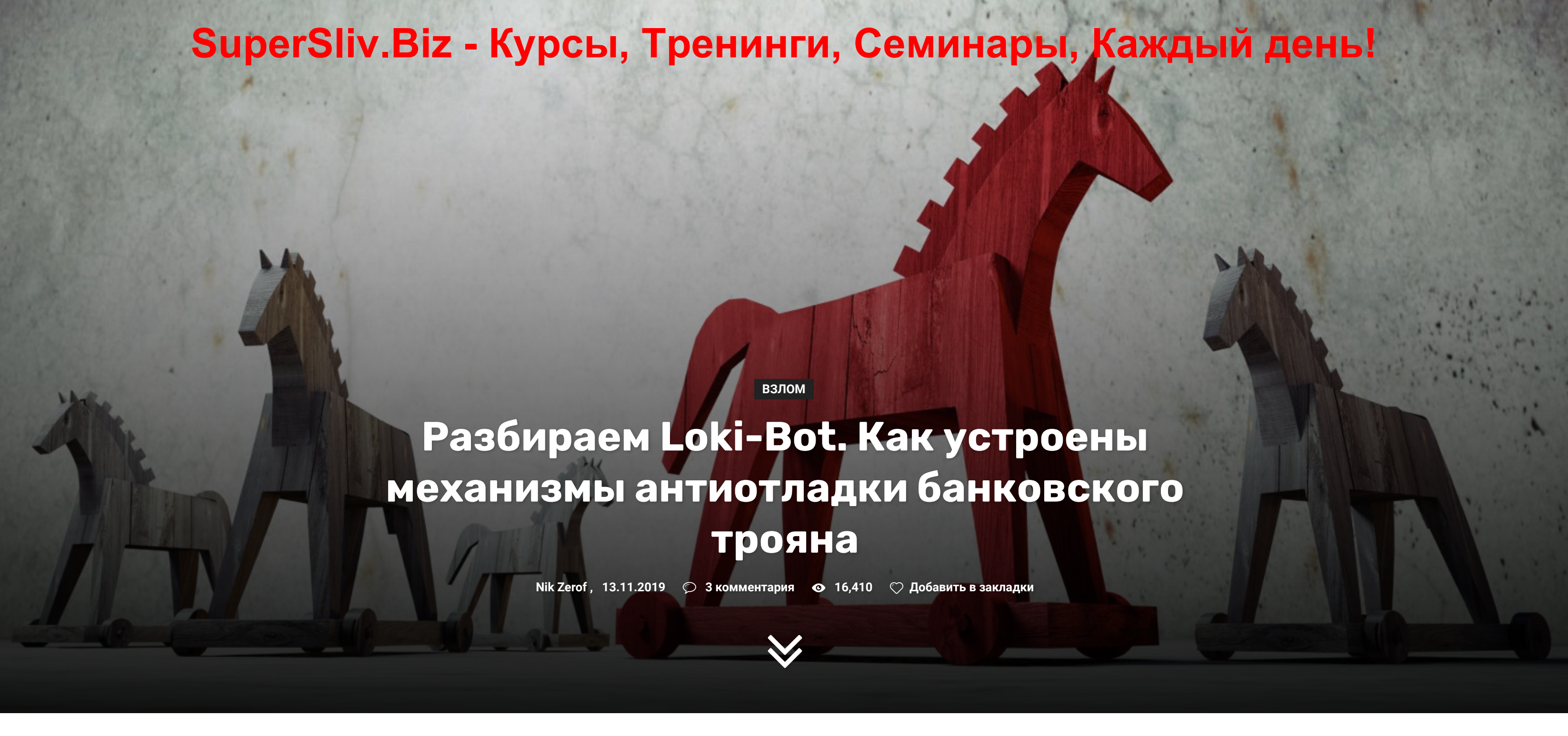
ВЗЛОМ

Разбираем Loki-Bot. Как устроены механизмы антиотладки банковского трояна

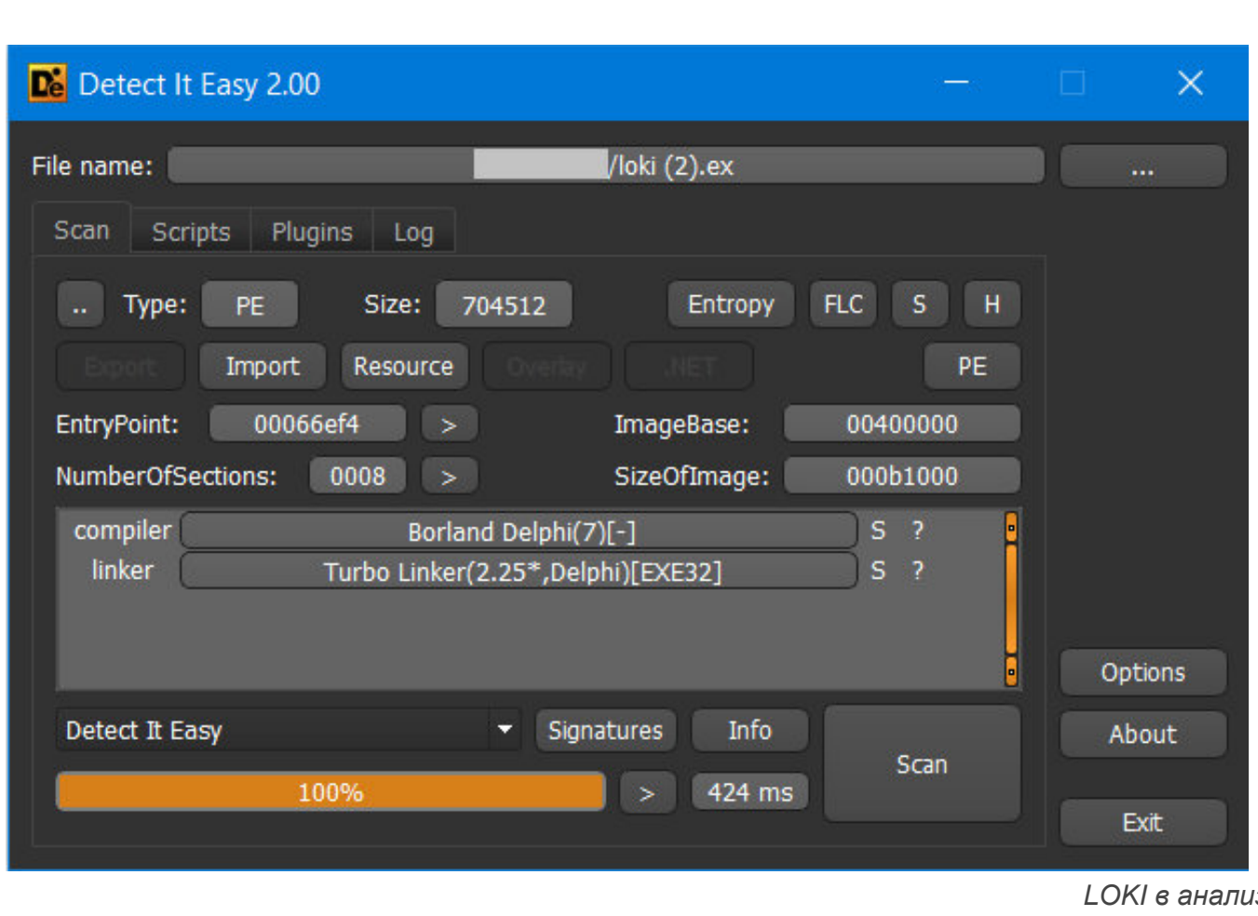
Nik Zerof , 13.11.2019 3 комментария 16,410 Добавить в закладки



Содержание статьи

Исследовать малварь не только весело, но и крайне познавательно. Недавно в мои цепкие руки попал банковский троян LOKI-bot. Подобные вредоносы обычно пишут настоящие профессионалы, и потому банкеры зачастую содержат достаточно интересные хаки. Так случилось и в этот раз — трой сопротивляется отладке и пытается «спалить» нас довольно нетривиальными способами.

Для начала загрузим семпл в DiE и посмотрим, что он покажет.



```
File name: [redacted]/loki (2).ex
Type: PE    Size: 704512
EntryPoint: 00066ef4    ImageBase: 00400000
NumberOfSections: 0008    SizeOfImage: 000b1000
compiler  Borland Delphi(7)[-]
linker    Turbo Linker(2.25*,Delphi)[EXE32]
100%    424 ms
```

LOKI в анализаторе DiE

Как видно на скриншоте, DiE отчего-то уверен, что наш семпл написан на Delphi, но это, разумеется, не так. Ты наверняка знаешь, что существуют специальные тулзы, которые умеют склеивать один файл с другим. Создатели LOKI-bot именно так и поступили (обрати внимание на размер секции ресурсов `rsrc` в файле относительно его общего размера). Оригинальный LOKI запустится после того, как отработает его «обертка».

INFO

Все эксперименты с боевой малварью я настоятельно рекомендую проводить на изолированной от сети виртуальной машине, иначе есть риск получить заражение банкером LOKI-bot и лишиться своих данных!

Для того чтобы разобраться в механизме самозащиты этого бота, мы должны представлять себе, каким образом вообще малварь может сопротивляться отладке. Обычно так или иначе трой старается получить список процессов в системе, а после этого уже начинаются дальнейшие действия. Давай попробуем двигаться в эту сторону, чтобы изучить механизм самозащиты бота. Как известно, получить список процессов в Windows можно несколькими способами.

1. Перечислить процессы при помощи функции `PSAPI EnumProcesses`.
2. Создать снимок процессов при помощи функции `CreateToolhelp32Snapshot`.
3. Перечислить процессы при помощи `ZwQuerySystemInformation`.
4. Применить трюки со счетчиками производительности.
5. Использовать Windows Management Instrumentation (WMI).

Безусловно, самый распространенный метод получения процессов — при помощи функции `CreateToolhelp32Snapshot`, поэтому мы начнем с установки брейк-пойнта на эту функцию (используется отладчик x64dbg, версия для архитектуры x86).

INFO

Исследование малвари или решение крякмисов может растянуться надолго: не всегда с первого раза ты ставишь брейк-пойнты на нужные функции, не всегда все идет по плану. В любом случае это вереница проб и ошибок. Когда ты читаешь статью, у тебя на это уходит максимум 10–15 минут, но это не значит, что любой взлом или снятие упаковки с файла будет занимать столько же времени. Просто есть «рабочие» моменты, рутина, которой не стоит утомлять читателя и которую автор пропускает. Поэтому создается впечатление, будто любой взлом быстр и элементарен, а автор с первого раза «угадывает», на какие функции WinAPI установить бряки, чтобы победить защиту. Но в жизни это не всегда так! 🙂

Начинаем погружение

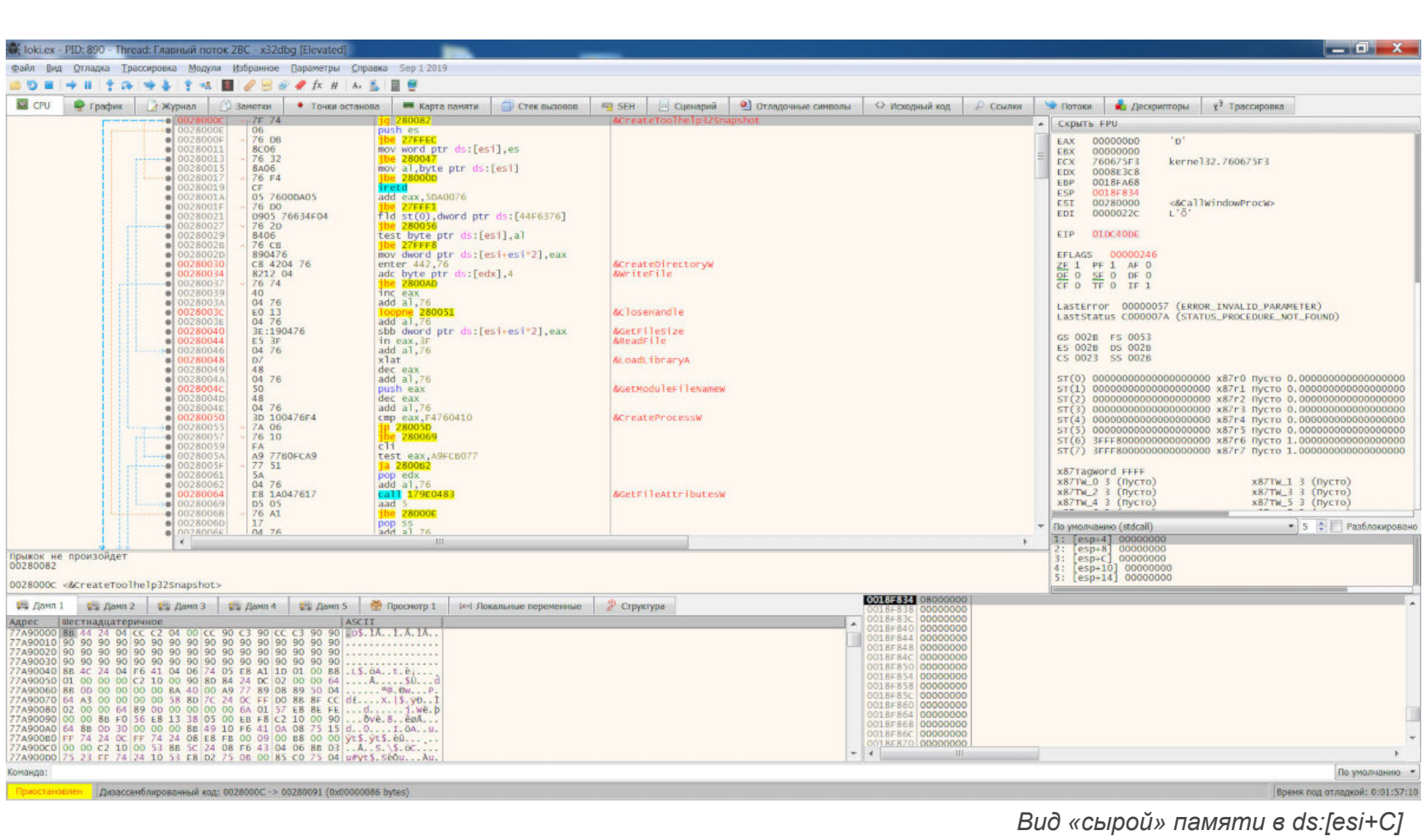
Итак, мы оказались в теле функции `CreateToolhelp32Snapshot`. Давай выполним ее и сделаем еще один шаг, чтобы вернуться по `ret`. В итоге мы попадаем в такой код:

```
01DC40D8 | 53                          | push ebx
01DC40D9 | 6A 02                       | push 2
01DC40DB | FF56 0C                     | call dword ptr ds:[esi+C]
01DC40DE | 8BD8                        | mov ebx,eax   <----- мы здесь
01DC40E0 | 83FB FF                     | cmp ebx,FFFFFFFF
01DC40E3 | 0F85 634C0000               | jne 1DC8D4C
01DC40E9 | 33C0                        | xor eax,eax
01DC40EB | E9 6AE5FFFF                 | jmp 1DC265A
```

Мы стоим на `01DC40DE`, но ты ведь помнишь, что мы только что вернулись из `CreateToolhelp32Snapshot`? Стало быть, call, который выше по коду, и есть вызов `CreateToolhelp32Snapshot`. Вспоминаем прототип `CreateToolhelp32Snapshot`:

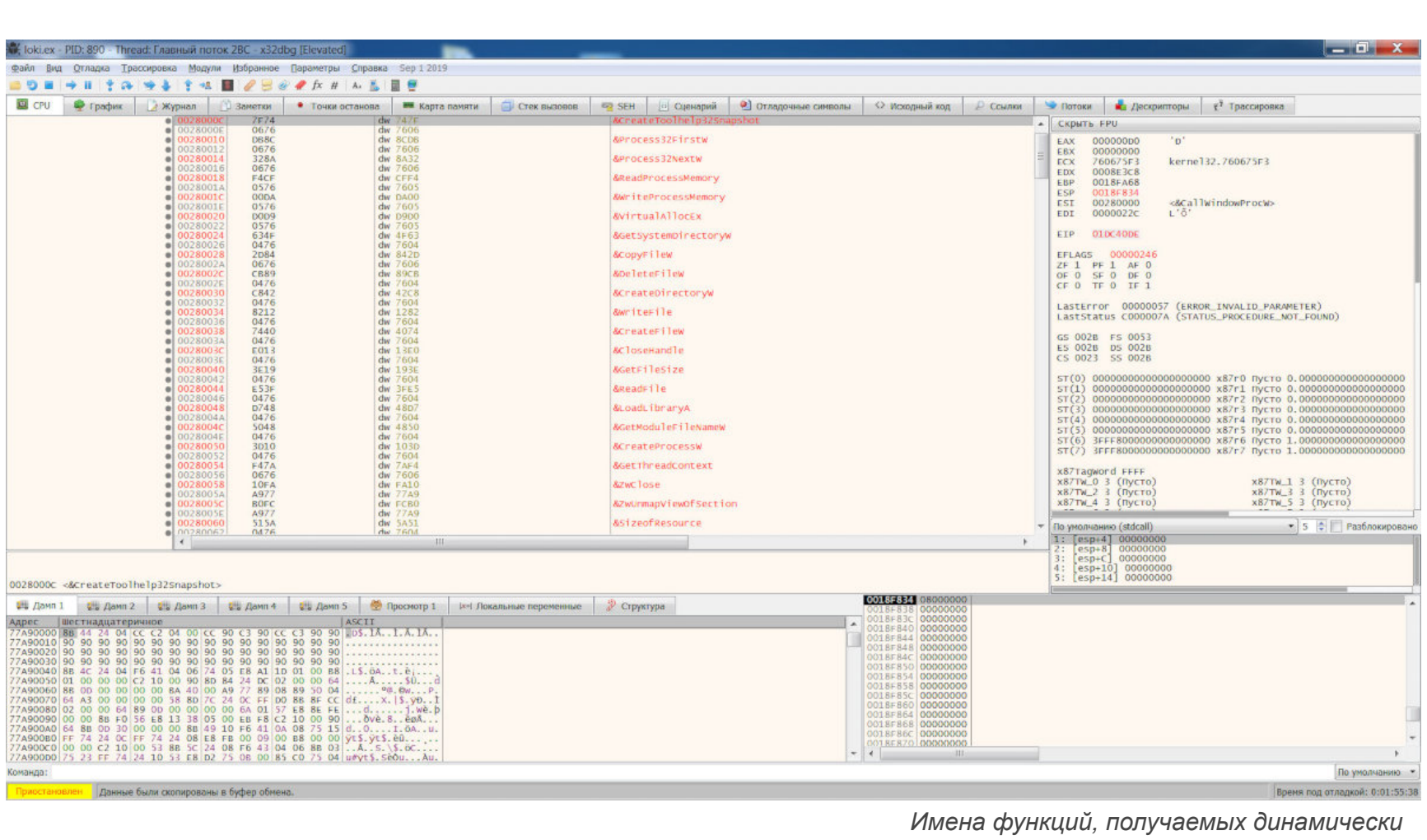
```
HANDLE CreateToolhelp32Snapshot(
    DWORD dwFlags,
    DWORD th32ProcessID
);
```

Как видим, передаются два аргумента, один из которых — `push 2`, что говорит о передаче параметра `TH32CS_SNAPPROCESS`. Он заставляет `CreateToolhelp32Snapshot` сделать снимок всех процессов. Все указывает на вызов `CreateToolhelp32Snapshot`, но этот call не похож на стандартный вызов WinAPI в коде. Идем в `ds:[esi+C]` и смотрим, что там есть.



Вид «сырой» памяти в ds:[esi+C]

Видим какой-то код, похожий на мусор, среди которого прослеживаются имена функций WinAPI. Давай представим весь код в виде DWORD'ов.



Имена функций, получаемых динамически

Перед нами список функций, которые LOKI получает динамически. По этому списку мы можем судить о том, как работает банкер. В дальнейшем вызовы используемых WinAPI будут выполняться подобными call'ами. Итак, со списком WinAPI разобрались, теперь вернемся в код. Чтобы ты понимал его структуру, приведу листинг части функции поиска процессов (обрати внимание на адреса!):

```
; CreateToolhelp32Snapshot
01DC40DB | FF56 0C         | call dword ptr ds:[esi+C]
01DC40DE | 8BD8            | mov ebx,eax
01DC40E0 | 83FB FF         | cmp ebx,FFFFFFFF
01DC40E3 | 0F85 634C0000   | jne 1DC8D4C
01DC40E9 | 33C0            | xor eax,eax
01DC40EB | E9 6AE5FFFF     | jmp 1DC265A

00228D4C | 8D85 D4FDFFFF   | lea eax,dword ptr ss:[ebp-22C]
00228D52 | 50              | push eax
00228D53 | E9 53ABFFFF     | jmp 2238AB

002238AB | 53              | push ebx
002238AC | 89BD D4FDFFFF   | mov dword ptr ss:[ebp-22C],edi
; Process32FirstW
002238B2 | FF56 10         | call dword ptr ds:[esi+10]
002238B5 | E9 97EDFFFF     | jmp 222651
```

Как видишь, код сильно фрагментирован и связан переходами (`jmp`). Это нас не запугает: ставим точку останова на конец функции получения списка процессов и жмем на выполнение. Далее нас ждет такой код (некоторые не особо интересные функции я буду называть, но не стану в них заглядывать — по названию будет все понятно):

```
; Имя процесса
007C5D83 | 8D8D F8FDFFFF   | lea ecx,dword ptr ss:[ebp-208]
; Считаем число символов имен процессов
007C5D89 | E8 4AD8FFFF     | call <str_len>
; Сравниваем размер имени процесса с числом 40h (64)
007C5D8E | 83F8 40         | cmp eax,40
; Все хорошо, продолжаем корректное выполнение трояна
007C5D91 | 0F82 08000000   | jb 7C5D9F
; Все плохо, завершаемся
007C5D97 | 6A 00           | push 0
; ExitProcess
007C5D99 | FF96 AC000000   | call dword ptr ds:[esi+AC]
007C5D9F | E9 E1D1FFFF     | jmp 7C2F85
```

Здесь считается размер имени каждого процесса и сравнивается с числом `40h`, или 64 в десятеричной системе. Если процесс с таким длинным именем найден, то выполняется выход. В чем тут был замысел?

Замысел создателя

Все дело в том, что обычно исследователи в вирусных лабораториях дают имена файлам семплов вредоносных программ в виде их хешей SHA-256, которые как раз имеют длину 64 символа. Если такой файл запустить, LOKI поймет это по числу символов процесса и выполнит выход. Или не поймет? Думаю, что внимательный читатель уже догадался: дело в том, что получаемое при помощи функции `CreateToolhelp32Snapshot` имя процесса также содержит его расширение, которое добавляет еще четыре символа к имени — `.exe`. Очевидно, разработчик забыл об этом, поэтому его замысел не сработает. Хотя идея достаточно оригинальна. Вывести из строя эту защиту банкера можно, проследив, чтобы в системе не было процессов с именами из 64 символов, включая расширение.

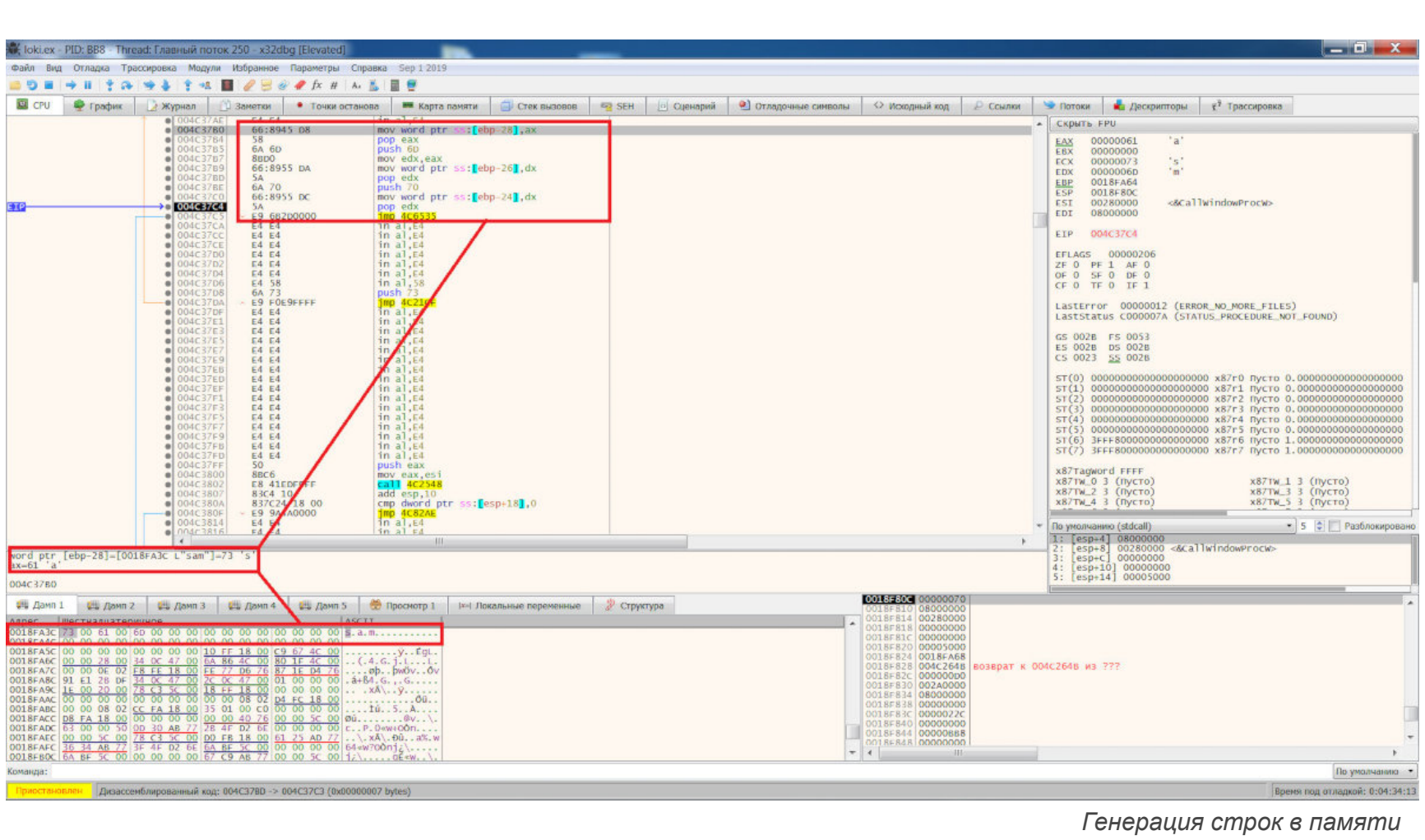
На очереди следующая проверка.

```
002567B4 | 85C0            | test eax,eax
002567B6 | 0F85 07000000   | jne 2567C3
002567BC | 53              | push ebx
; ExitProcess
002567BD | FF96 AC000000   | call dword ptr ds:[esi+AC]
002567C3 | 56              | push esi
002567C4 | E8 D4DBFFFF     | call <check_name>
002567C9 | 59              | pop ecx
002567CA | 85C0            | test eax,eax
```

Я обозначил нужный call меткой `<check_name>`, чтобы было понятнее. Если прыгнуть в него, можно увидеть такой код (я добавил комментарии, хоть код и простой):

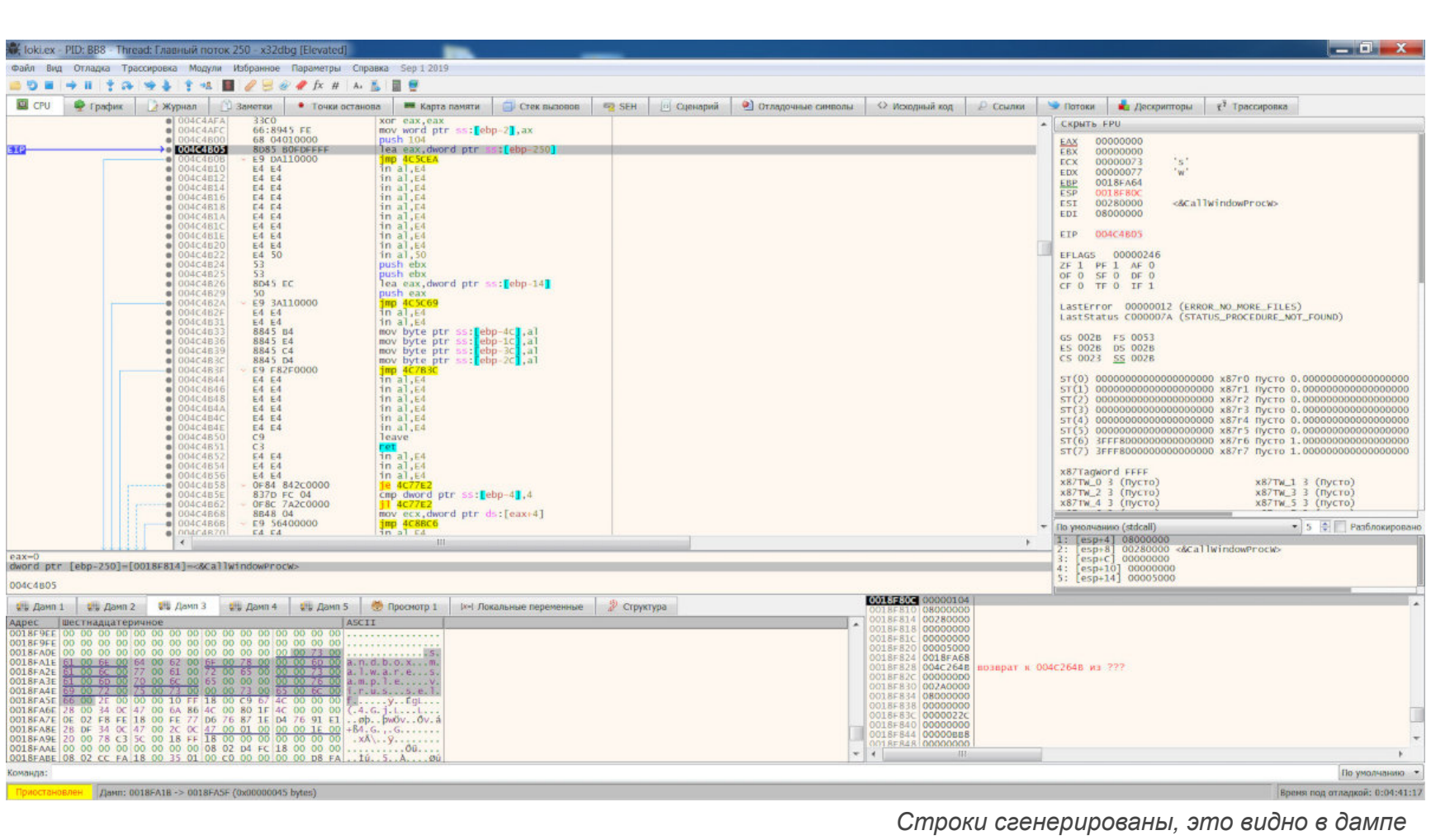
```
; Положим в память символ по адресу ss:[ebp-28]
004C37B0 | 66:8945 D8    | mov word ptr ss:[ebp-28],ax
004C37B4 | 58            | pop eax
; Берем символ
004C37B5 | 6A 6D         | push 6D
004C37B7 | 8BD0          | mov edx,eax
; Положим в память по адресу ss:[ebp-26]
004C37B9 | 66:8955 DA    | mov word ptr ss:[ebp-26],dx
004C37BD | 5A            | pop edx
; Берем символ
004C37BE | 6A 70         | push 70
; Положим в память по адресу ss:[ebp-24]
004C37C0 | 66:8955 DC    | mov word ptr ss:[ebp-24],dx
004C37C4 | 5A            | pop edx
004C37C5 | E9 6B2D0000   | jmp 4C6535
```

Очевидно, в память помещаются значения, являющиеся кодами символов. Давай пройдем этот код (это только фрагмент, код связан jmp'ами) и посмотрим на окно дампа, например в окрестностях `[ebp-26]`.



Генерация строк в памяти

В памяти сгенерированы слова `self`, `sample`, `sandbox`, `virus`, `malware`. Такой подход к генерации нужных строк в памяти используется для того, чтобы строковые константы не бросались в глаза в исполняемом файле LOKI.



Строки сгенерированы, это видно в дампе

Далее идет вызов WinAPI `GetModuleFileName`:

```
004C7B13 | 6A 00         | push 0
; GetModuleFileName
004C7B15 | FF50 4C       | call dword ptr ds:[eax+4C]
004C7B18 | 85C0          | test eax,eax
004C7B1A | E9 73F4FFFF   | jmp 4C6F92
```

Так LOKI проверяет, не содержит ли имя его исполняемого файла этих слов, и если содержит, то он сразу завершает работу. Кроме того, подобным образом генерируются названия процессов популярных исследовательских программ, например ollydbg.exe, procmon.exe, procmon64.exe, windbg.exe, procexp64.exe, и названия процессов антивируса Avast — avgsvc.exe, iavgui.exe, avastsvc.exe и так далее.

После этого сгенерированные названия сверяются с полученным ранее при помощи функции `CreateToolhelp32Snapshot` списком процессов. Что интересно, проверяются процессы только антивируса Avast, другие LOKI почему-то не просматривает. Чтобы успешно пройти и эту проверку, переименуй все приложения из списка или просто не запускай их, иначе это спугнет LOKI и он завершится. А мы подходим к следующему антиотладочному трюку.

После проверки имен процессов мы видим такой код:

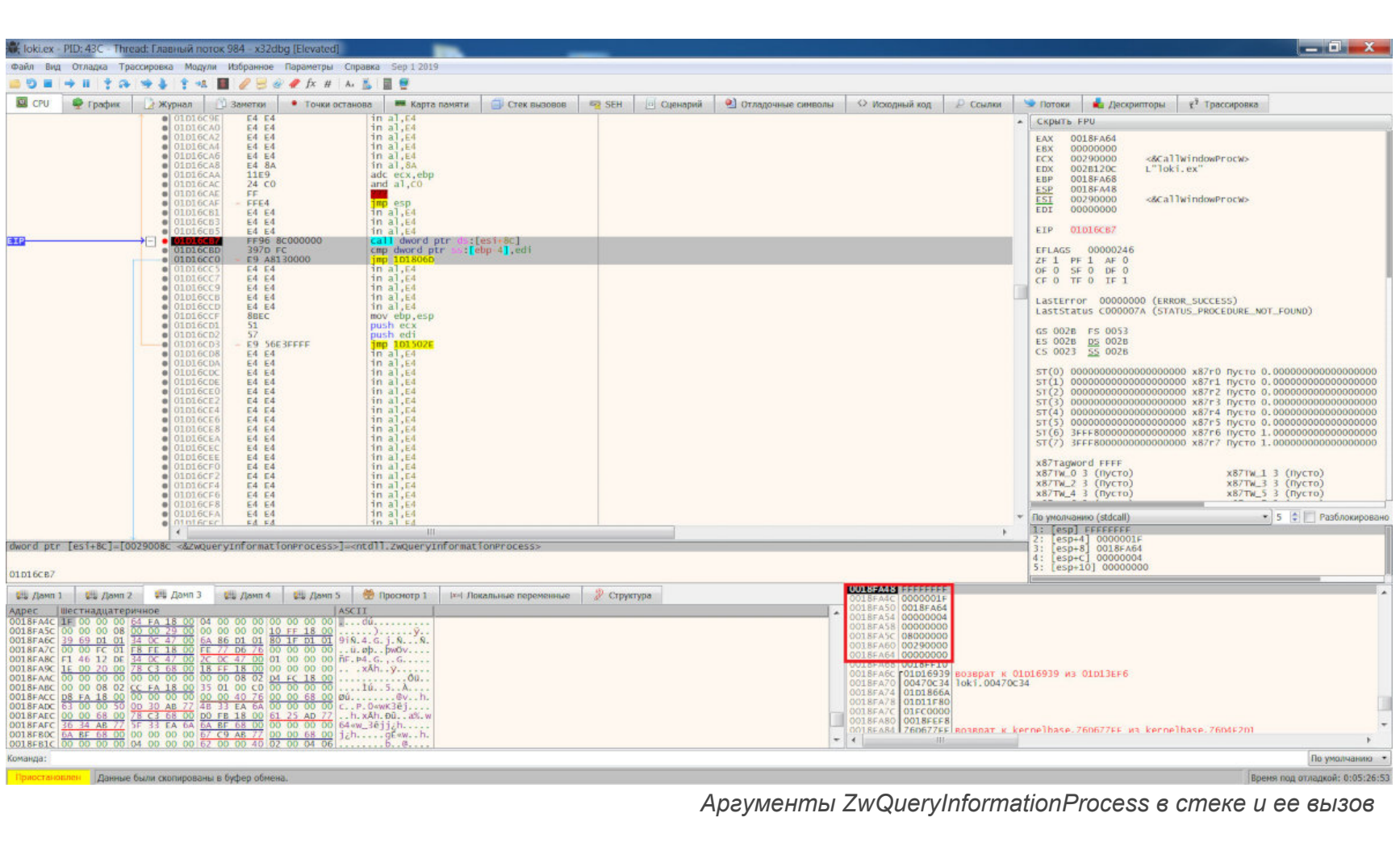
```
; ZwQueryInformationProcess
01D16CB7 | FF96 8C000000   | call dword ptr ds:[esi+8C]
01D16CBD | 397D FC         | cmp dword ptr ss:[ebp-4],edi
01D16CC0 | E9 A8130000     | jmp 1D1806D
```

Очевидно, вызов `ZwQueryInformationProcess` здесь используется не просто так. Вспоминаем, что есть антиотладочный трюк, основанный на вызове `WinAPI ZwQueryInformationProcess`. Смотрим, какие аргументы передаются в эту функцию, попутно вспоминая ее прототип:

```
NTSTATUS WINAPI ZwQueryInformationProcess(
    _In_        HANDLE           ProcessHandle,
    _In_        PROCESSINFOCLASS ProcessInformationClass,
    _Out_       PVOID            ProcessInformation,
    _In_        ULONG            ProcessInformationLength,
    _Out_opt_   PULONG           ReturnLength
);
```

Больше всего нас будут интересовать поля `ProcessHandle` и `ProcessInformationClass`. Первое поле указывает на процесс, о котором мы получаем информацию, второе — интересующий нас флаг. Отладчик показывает, что первое поле имеет значение `FFFFFFFF`, а второе — `0x1f`. Чтобы в этом убедиться, посмотри, какие параметры передаются в функцию через стек, на скриншоте.

Переданное значение `FFFFFFFF` в поле `ProcessHandle` говорит о том, что мы работаем со своим собственным процессом (`GetCurrentProcess();`). Значение поля `ProcessInformationClass 0x1f` указывает на использование `ProcessDebugFlags`, который сигнализирует о том, что процесс находится под отладкой. Также будет проверен флаг `0x1E` (`ProcessDebugObjectHandle`), который тоже указывает на отладку.

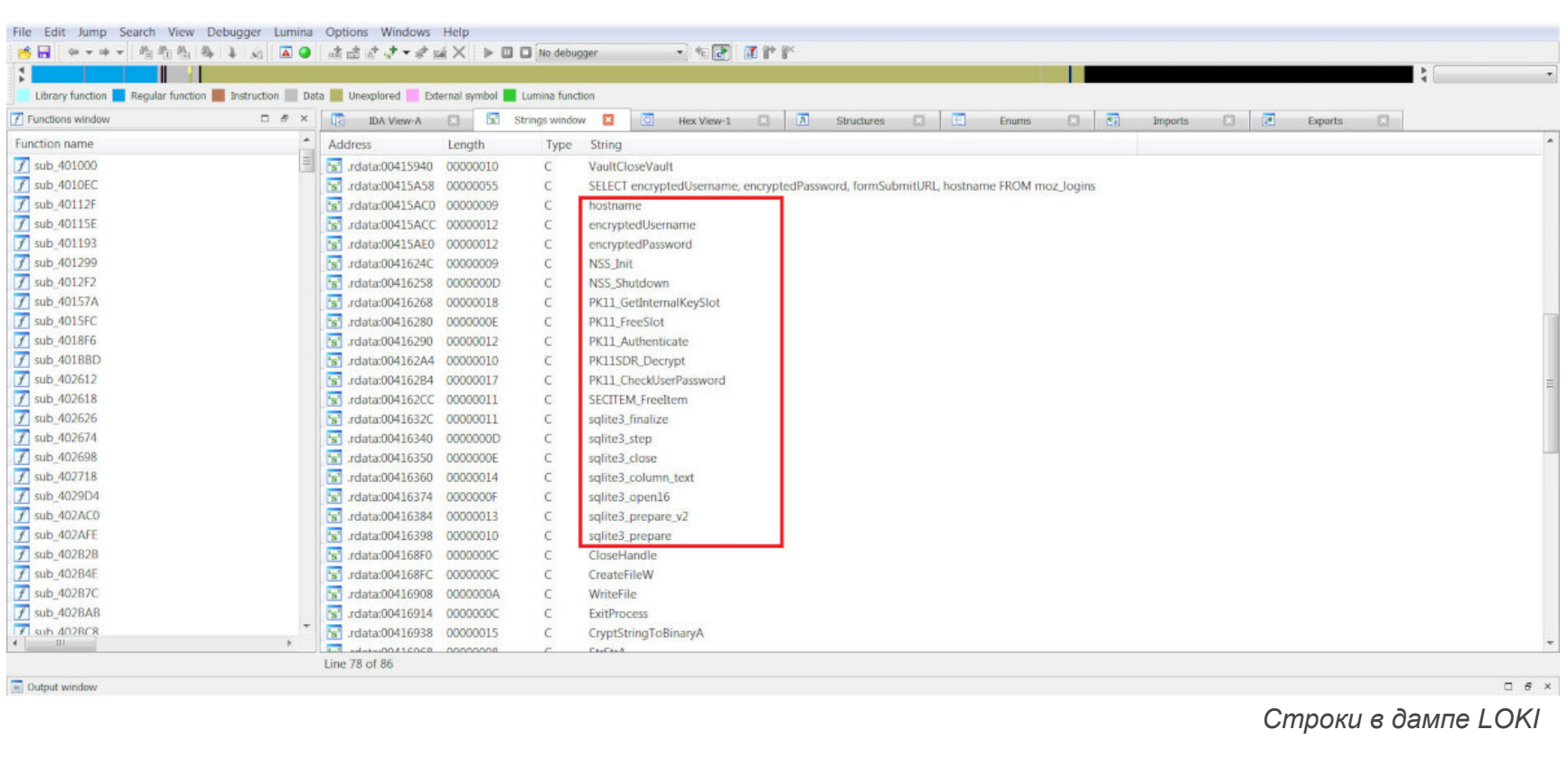


Аргументы ZwQueryInformationProcess в стеке и ее вызов

Кстати, в «Хакере» была отдельная статья, которая описывала этот (и другие) методы определения отладки.

Ну а далее, разумеется, мы обнаруживаем условный переход в зависимости от состояния указанных выше флагов. Как его преодолевать, ты и сам знаешь: либо перескочить, установив регистр `eip` на нужный адрес, либо забить nop'ами, либо поменять значение флага перехода в отладчике. Лично я просто поменял значение регистра `eip` на следующую после условного перехода команду.

Это был последний рубеж антиотладочной защиты LOKI, после этого начинает работать его полностью распакованное тело, о чем говорят достаточно интересные строки в снятом дампе.



```
Address              Length    Type  String
.rdata:00415940      00000010  C     VaultCloseVault
.rdata:004115A58     00000055  C     SELECT encryptedUsername, encryptedPassword, formSubmitURL, hostname FROM moz_logins
.rdata:00415AC0      00000009  C     hostname
.rdata:00415ACC      00000012  C     encryptedUsername
.rdata:00415AE0      00000012  C     encryptedPassword
.rdata:0041624C      00000009  C     NSS_Init
.rdata:00416258      0000000D  C     NSS_Shutdown
.rdata:00416268      00000018  C     PK11_GetInternalKeySlot
.rdata:00416280      0000000E  C     PK11_FreeSlot
.rdata:00416290      00000012  C     PK11_Authenticate
.rdata:004162A4      00000010  C     PK11SDR_Decrypt
.rdata:004162B4      00000017  C     PK11_CheckUserPassword
.rdata:004162CC      00000011  C     SECITEM_FreeItem
.rdata:0041632C      00000011  C     sqlite3_finalize
.rdata:00416340      0000000D  C     sqlite3_step
.rdata:00416350      0000000E  C     sqlite3_close
.rdata:00416360      00000014  C     sqlite3_column_text
.rdata:00416374      0000000F  C     sqlite3_open16
.rdata:00416384      00000013  C     sqlite3_prepare_v2
.rdata:00416398      00000010  C     sqlite3_prepare
.rdata:004168F0      0000000C  C     CloseHandle
.rdata:004168FC      0000000C  C     CreateFileW
.rdata:00416908      0000000A  C     WriteFile
.rdata:00416914      0000000C  C     ExitProcess
.rdata:00416938      00000015  C     CryptStringToBinaryA
Line 78 of 86
```

Строки в дампе LOKI

Очевидно, LOKI собрался стилить мои пароли из браузера Firefox.

Итог

В этой статье мы рассмотрели, какие уловки вирмейкеры применяют против средств анализа бинарных файлов, на примере банкера LOKI-bot. Какие-то способы были уже много раз описаны, какие-то достаточно нетривиальны по своему подходу. Но, как видишь, при помощи отладчика и дизассемблера они преодолеваются быстро и эффективно.