# Обнаружение руткитов режима ядра с помощью отладчика

Дмитрий Олексюк, eSage Lab
dmitry@esagelab.ru

## Вступление

В настоящее время вредоносные программы всё чаще снабжаются функциями активного противодействия различным защитным системам. Времена, когда трояны и черви банально завершали процессы антивирусов по списку имён, давно прошли; сейчас никого не удивишь даже снятием перехватов и деактивацией фильтров, установленных HIPS-ами. А использование rootkit-технологий для сокрытия вредоносного кода в системе стало практически стандартом.

В процессе анализа пойманных in-the-wild ("в живой природе") экземпляров руткитов трудно переоценить помощь утилит-антируткитов. Бесплатные программы этого класса, такие как GMER, Rootkit Unhooker, IceSword, Safe n'Sec Rootkit Detector, на данный момент остаются одним из самых удобных и эффективных инструментов для поиска и анализа руткитов. Специфика технологий, используемых в перечисленных антируткитах, делает их обход нетривиальной задачей, и как следствие - практически нерешаемой для авторов подавляющего большинства зловредов. Этот факт заставляет плохих парней идти по несколько иному пути: а именно, препятствовать нормальной работе защитных программ вместо их обхода. Что делать, если ни один из этих антируткитов не заработал на зараженной машине? В таком случае можно проанализировать и деактивировать руткит практически "голыми руками", при помощи только отладчика и понимания архитектуры операционной системы. Об этом и пойдёт речь в данной статье.

Как известно, существует два типа руткитов: работающие в режиме пользователя и в режиме ядра. В статье будут рассматриваться именно второй тип, как наиболее опасный и сложнонейтрализуемый. С первой категорией руткитов прекрасно справляются не только специализированные утилиты, но и антивирусные программы многих известных производителей.

В качестве отладчика мной будет использоваться WinDbg из пакета Microsoft Debugging Tools for Windows. Удобство в использовании, гибкость, расширяемость, возможность удалённой отладки по COM, 1394 или USB порту, а также поддержка 64-х разрядных систем делают WinDbg одним из самых привлекательных отладчиков ядра на данный момент. Однако и тем, кто использует Syser или SoftIce, не составит труда произвести описываемые действия.

Данная статья не претендует на достойную справочника по NT internals полноту раскрытия темы руткитов; она лишь даёт общее понимание логики их поиска и нейтрализации.

## Приступая к работе

Для того чтобы иметь возможность полноценно отлаживать ядро операционной системы с помощью WinDbg, требуется два компьютера. При этом в качестве второго (целевого) компьютера можно использовать виртуальную машину. Для этого нужно настроить в виртуальной машине COM-порт и сконфигурировать гостевую ОС для поддержки удаленной отладки, после чего можно подключить выбранному COM-порту WinDbg. [1]

После того, как получена работающая связка "виртуальная машина - отладчик", нужно загрузить отладочные символы для системных модулей. В отличие от SoftIce, при этом нет необходимости что-либо скачивать, конвертировать и добавлять вручную: WinDbg сам загрузит с сервера Microsoft отладочные символы для актуальной версии ядра и драйверов. Для этого достаточно в пункте меню File→Symbol File Path прописать строку вида:

C:\symbols;SRV*C:\symbols*http://msdl.microsoft.com/download/symbols

Здесь 'C:\symbols' - произвольный локальный каталог для сохранения загруженных отладочных символов.

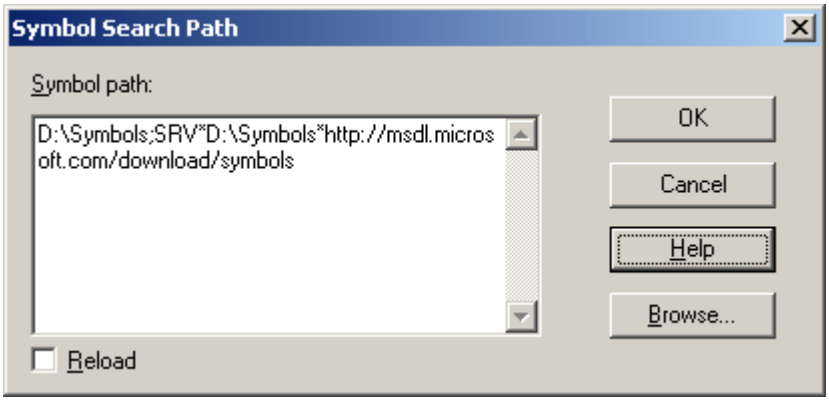


*Рис. 1. Настройка путей для загрузки отладочных символов.*

Для того чтобы загрузить или перезагрузить символы, используется команда отладчика .reload.

Обратимся к сути проблемы. Каким образом работающий в системе руткит может препятствовать работоспособности антируткита? Очевидно, что для этого ему необходимо перехватывать управление в определённых точках операционной системы, препятствуя запуску или инициализации определённой программы. Способов и точек для осуществления такого контроля достаточно много. К ним относятся как стандартные механизмы, предоставляемые операционной системой, так и недокументированные, такие как сплайсинг ключевых функций, подмена указателей в различных внутренних структурах ядра и драйверов, техники вроде Direct Kernel Object Manipulation и многое другое.

По способу реализации и целевому предназначению все механизмы контроля запуска программ можно разделить на четыре условные категории:

- Контроль системных событий с использованием нотификаторов.
- Перехват системных вызовов.
- Перехват файловых операций.
- Перехват методов диспетчера объектов.

Рассмотрим их по порядку.

## Контроль системных событий с использованием нотификаторов

Очень часто легитимным драйверам требуется каким-либо образом узнавать о запуске в системе новых потоков или процессов, загрузке в память исполняемых образов, создании и модификации ключей и параметров системного реестра и т.п. Для того чтобы избавить программистов от необходимости изобретения и применения ненадежных недокументированных решений, разработчиками ядра Windows были предусмотрены удобные механизмы нотификации. Эти механизмы достаточно часто используются и во вредоносных программах.

## Процессы

Для установки нотификатора на создание или завершение процесса используется функция ядра PsSetCreateProcessNotifyRoutine, имеющая следующий прототип:

```
NTSTATUS
PsSetCreateProcessNotifyRoutine(
    IN  PCREATE_PROCESS_NOTIFY_ROUTINE     Notify-
Routine,
    IN BOOLEAN  Remove
);
```

На вход функция получает два аргумента:

- NotifyRoutine - указатель на регистрируемую функцию, которая будет вызвана при создании или завершении какого либо процесса.
- Remove - TRUE если данный нотификатор удаляется, FALSE если устанавливается.

Функция-нотификатор должна иметь тип PCREATE_PROCESS_NOTIFY_ROUTINE, который объявлен так:

```
VOID
(*PCREATE_PROCESS_NOTIFY_ROUTINE)(
    IN HANDLE  ParentId,
    IN HANDLE  ProcessId,
    IN BOOLEAN  Create
);
```

Параметры ParentId и ProcessId являются идентификаторами родительского и создаваемого или завершаемого процесса, соответственно. В качестве Create передаётся TRUE, если процесс создаётся, и FALSE если завершается.

Принцип использования предоставляемых данным механизмом возможностей во вредоносных программах очевиден: узнав о запуске процесса известного антируткита, можно помешать его нормальной инициализации. Для этого достаточно модифицировать код исполняемого модуля в памяти, или вовсе завершить создаваемый процесс ещё до того, как он успеет выполнить какие-либо действия.

Рассмотрим, что происходит, когда драйвер регистрирует новый нотификатор, вызывая PsSetCreateProcessNotifyRoutine.

Список нотификаторов находится в секции данных ядра. Его размер определяется внутренней константой PSP_MAX_CREATE_PROCESS_NOTIFY, которая в 32-х разрядных операционных системах Windows XP, 2003 Server и Vista равняется 8. Сам список имеет следующий тип:

```
EX_CALLBACK PspCreateProcessNotifyRoutine
[PSP_MAX_CREATE_PROCESS_NOTIFY];
```

А структура EX_CALLBACK выглядит так:

```
typedef struct _EX_CALLB
{
    EX_FAST_REF  RoutineBlock;
}
EX_CALLBACK, *PEX_CALLBACK;
```

Поле RoutineBlock данной структуры представляет собой указатель на ещё одну структуру EX_CALLBACK_ROUTINE_BLOCK. При этом младшие 3 бита поля хранят количество ссылок на данный указатель и используются для синхронизации доступа к нему.

Рассмотрим структуру EX_CALLBACK_ROUTINE_BLOCK:

```
typedef struct _EX_CALLBACK_ROUTINE_BLOCK
{
    EX_RUNDOWN_REF  RundownProtect;
    PEX_CALLBACK_FUNCTION  Function;
    PVOID  Context;
}
EX_CALLBACK_ROUTINE_BLOCK,
*PEX_CALLBACK_ROUTINE_BLOCK;
```

Поле Function является указателем на функцию-нотификатор, которая была передана в PsSetCreateNotifyRoutione при его установке.

Сама функция PsSetCreateNotifyRoutione работает довольно просто:

1. Циклическим перебором массива PspCreateProcessNotifyRoutine ищется в нем нулевой элемент.
2. При нахождении такового - выделяется память в подкачиваемом пуле ядра под нужные служебные структуры.
3. Инициализация служебных структур.
4. Добавление нового указателя в список PspCreateProcessNotifyRoutine.
5. Инкремент глобальной переменной PspCreateProcessNotifyRoutineCount (она содержит количество активных на данный момент нотификаторов).
6. Возврат STATUS_SUCCESS.

Таким образом, всё, что нужно знать для перечисления активных нотификаторов на создание процессов - это адреса внутренних переменных ядра PspCreateProcessNotifyRoutine и PspCreateProcessNotifyRoutineCount. Найти их очень просто. Для этого откроем в WinDbg машинный код функции PsSetCreateProcessNotifyRoutine и введём nt!PsSetCreateProcessNotifyRoutine в поле Offset окна дизассемблера (View→Disassembly, или Alt+7).

Рассмотрим наиболее значимые участки кода.

Определение PSP_MAX_CREATE_PROCESS_NOTIFY и перебор списка:

```
nt!PsSetCreateProcessNotifyRoutine+0x43:
; цикл, перебирающий
; массив PspCreateProcessNotifyRoutine
; счётчик итераций сравнивается с
; PSP_MAX_CREATE_PROCESS_NOTIFY
805c4715 83fb08          cmp     ebx,8
; переход к следующему элементу цикла
805c4718 72cc            jb      805c46e6
; …или выход из функции со значением
STATUS_PROCEDURE_NOT_FOUND
805c471a b87a0000c0      mov     eax,0C000007Ah
805c471f eb65            jmp     805c4786
```

Добавление нотификатора в список:

```
nt!PsSetCreateProcessNotifyRoutine+0x74:
; NULL
805c4746 53              push    ebx
; Адрес нотификатора
805c4747 ff7508          push    dword ptr [ebp+8]
805c474a e8e1d30300      call
nt!ExAllocateCallBack
805c474f 8bf0            mov     esi,eax
805c4751 3bf3            cmp     esi,ebx
805c4753 7507            jne     805c475c
805c4755 b89a0000c0      mov     eax,0C000009Ah
805c475a eb2a            jmp     805c4786
; а вот и указатель на нужный нам список
; нотификаторов
805c475c bfe0935580      mov     edi,offset
     nt!PspCreateProcessNotifyRoutine (805593e0)
805c4761 6a00            push    0
805c4763 56              push    esi
805c4764 57              push    edi
; добавление новой функции в список
805c4765 e8f6d30300      call
nt!ExCompareExchangeCallBack
```

Выделение памяти под EX_CALLBACK_ROUTINE_BLOCK и его заполнение:

```
nt!ExAllocateCallBack:
80601b30 8bff     mov     edi,edi
80601b32 55       push    ebp
80601b33 8bec     mov     ebp,esp
80601b35 6843627262 push    62726243h
; размер выделяемой памяти в пуле
; sizeof(EX_CALLBACK_ROUTINE_BLOCK)
80601b3a 6a0c     push    0Ch
; тип пула (PagedPool)
80601b3c 6a01     push    1
80601b3e e83d25f4ff call
nt!ExAllocatePoolWithTag
80601b43 85c0     test    eax,eax
80601b45 740f     je      80601b56
80601b47 8b4d08   mov     ecx,dword ptr [ebp+8]
80601b4a 832000   and     dword ptr [eax],0
```

```
; помещаем указатель на нотификатор в
; EX_CALLBACK_ROUTINE_BLOCK
80601b4d 894804 mov     dword ptr [eax+4],ecx
80601b50 8b4d0c mov     ecx,dword ptr [ebp+0Ch]
80601b53 894808 mov     dword ptr [eax+8],ecx
80601b56 5d     pop     ebp
80601b57 c20800 ret     8
```

Функция PspCreateProcessNotifyRoutine не экспортируется ядром, однако, если отладочные символы загружены, её имя будет видно в окне отладчика.

Информации в приведенных дизассемблерных листингах вполне достаточно для написания функции, ищущей список нотификаторов путём анализа машинного кода. Эта функция, в свою очередь, может использоваться в утилите, выводящей список нотификаторов. Так как написание подобного кода выходит за рамки статьи, ограничимся перечислением нотификаторов в отладчике.

Для вывода списка нотификаторов используются следующие команды:

```
kd> dd nt!PspCreateProcessNotifyRoutineCount L1
80559400 00000001
kd> dd nt!PspCreateProcessNotifyRoutine L 8
805593e0  e184e1d7 00000000 00000000 00000000
805593f0  00000000 00000000 00000000 00000000
```

В данном случае в системе установлен один нотификатор. Адрес указателя на EX_CALLBACK_ROUTINE_BLOCK для него - e184e1d0 (не забываем отбрасывать младшие 3 бита). Из последнего, в свою очередь, легко узнать и адрес функции обработчика:

```
kd> dd e184e1d0 L 3
e184e1d0  00000010 f95398d8 00000000
```

Так как поле Function находится по смещению 4, адрес нотификатора равен f95398d8. (См. Рис. 1)

Удалить этот нотификатор можно, например, затерев нулями соответствующий элемент массива PspCreateProcessNotifyRoutine:

```
kd> ed 805593e0 0
```

…и уменьшив на 1 значение PspCreateProcessNotifyRoutineCount:

```
kd> ed nt!PspCreateProcessNotifyRoutineCount 0
```

Нотификаторы, которые устанавливаются на потоки, исполняемые образы и ключи системного реестра, работают аналогичным образом. Разница лишь в названиях глобальных переменных, хранящих список нотификаторов.

Что касается переменной PspCreateProcessNotifyRoutineCount (и ей аналогичных переменных для других типов объектов) - в памяти она всегда находится сразу же за списком PspCreateProcessNotifyRoutine. Этот факт избавляет разработчика от необходимости писать отдельный код для поиска PspCreateProcessNotifyRoutineCount при разработке, к примеру, монитора нотификаторов как отдельного приложения или части руткит-детектора. (См. Рис. 3)

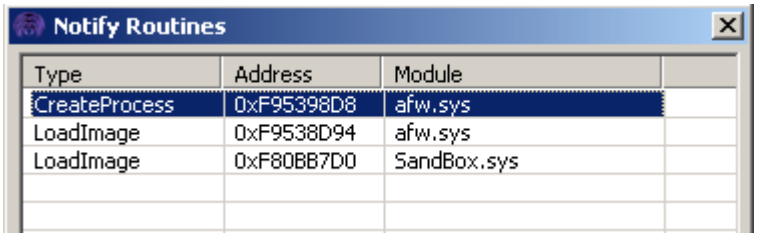


*Рис.2. Rootkit Unhooker обнаружил нотификатора, установленные фаерволом Outpost.*

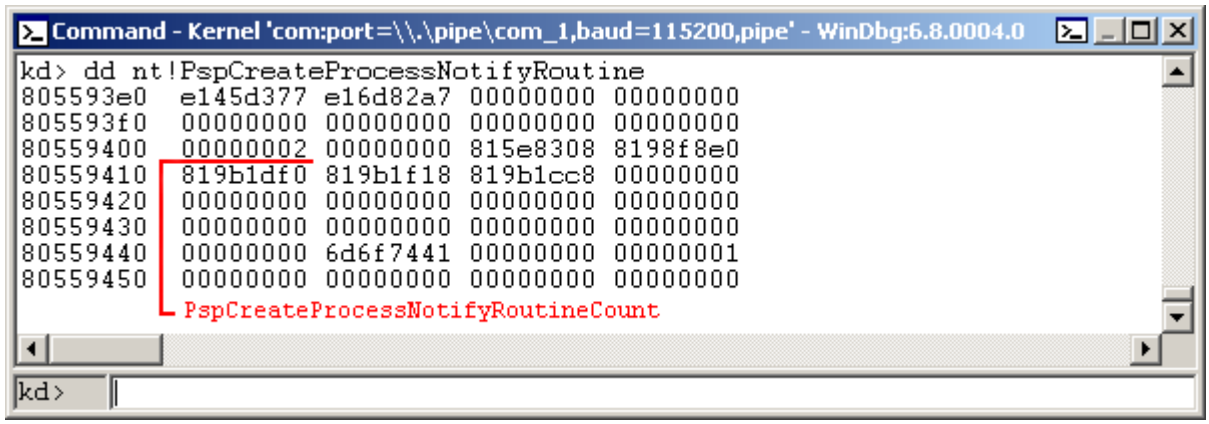


*Рис. 3. Поиск PspCreateProcessNotifyRoutineCount.*

## Потоки

Для установки нотификатора на создание и завершение потоков используется функция PsSetCreateThreadNotifyRoutine:

```
NTSTATUS
PsSetCreateThreadNotifyRoutine(
    IN PCREATE_THREAD_NOTIFY_ROUTINE  Notify-
Routine
);
```

Параметр NotifyRoutine - это указатель на функцию-нотификатор, которая будет вызвана при создании или завершении какого-либо потока.

Нотификатор имеет следующий прототип:

```
VOID (*PCREATE_THREAD_NOTIFY_ROUTINE)(
    IN HANDLE  ProcessId,
    IN HANDLE  ThreadId,
    IN BOOLEAN  Create
);
```

Параметры ProcessId и ThreadId содержат идентификатор процесса и идентификатор создаваемого или завершаемого потока, соответственно. В качестве параметра Create, нотификатору передаётся TRUE если поток создаётся и FALSE, если он завершается. Список установленных нотификаторов хранится в глобальной переменной ядра PspCreateThreadNotifyRoutine. Размер списка нотификаторов (т.е., их максимальное количество) определяется константой PSP_MAX_CREATE_THREAD_NOTIFY, которая, так же как и в случае с нотификаторами на создание и завершение процессов, равняется 8-ми. Количество активных на данный момент нотификаторов определяется переменной PspCreateThreadNotifyRoutineCount.

Код функции PspSetCreateThreadNotifyRoutine приведен в Листинге 1. Принципы работы всех функций установки нотификаторов, как уже было сказано выше, одинаковы. Перечисление и удаление нотификаторов на потоки происходит точно таким же образом, как и нотификаторов на процессы.

## Исполняемые образы

Ещё один способ блокировать работу какого-либо приложения заключается в использовании нотификаторов на загрузку исполняемых образов. Устанавливаются они с помощью функции PsSetLoadImageNotifyRoutine:

```
NTSTATUS
PsSetLoadImageNotifyRoutine(
    IN PLOAD_IMAGE_NOTIFY_ROUTINE  NotifyRoutine
);
```

Параметр NotifyRoutine - это указатель на функцию-нотификатор, которая вызывается при загрузке в адресное пространство процесса какого-либо модуля, от исполняемого файла самого процесса до связанных с ним динамических библиотек, а также при загрузке в память ядра драйверов. Нотификатор имеет следующий прототип:

```
VOID
(*PLOAD_IMAGE_NOTIFY_ROUTINE)(
    IN PUNICODE_STRING  FullImageName,
    IN HANDLE  ProcessId,
    IN PIMAGE_INFO  ImageInfo
);
```

В FullImageName передаётся полный дисковый путь к исполняемому образу, в ProcessId - идентификатор процесса, в который загружается образ (или 0, если нотификатор был вызван для драйверов режима ядра).

Структура IMAGE_INFO определена так:

```
typedef struct  _IMAGE_INFO
{
    union {
        ULONG  Properties;
        struct {
            ULONG ImageAddressingMode  : 8;
            ULONG SystemModeImage      : 1;
            ULONG ImageMappedToAllPids : 1;
            ULONG Reserved             : 22;
        };
    };
    PVOID  ImageBase;
    ULONG  ImageSelector;
    ULONG  ImageSize;
    ULONG  ImageSectionNumber;
}
IMAGE_INFO, *PIMAGE_INFO;
```

Подробное описание этой структуры может быть найдено в документации, входящей в состав Windows Driver Development Kit. Как и в случае с другими функциями по установке нотификаторов, нас больше всего интересуют переменная PspLoadImageNotifyRoutine, в которой хранится список нотификаторов, и переменная PspLoadImageNotifyRoutineCount, определяющая текущее количество нотификаторов. Размер нужного нам списка задаётся константой PSP_MAX_LOAD_IMAGE_NOTIFY, которая так же равна 8-ми.

Код функции очень простой и предельно похож на код PsSetCreateThreadNotify. (См. Листинг 2).

## Системный реестр

Достаточно часто попадаются экземпляры вредоносных программ, устанавливающие нотификаторы на операции с системным реестром. Это может потребоваться для блокировки доступа к своим ключам или для того, чтобы помешать нормальной работе антируткита или антивируса, предотвратив создание нужных ему ключей реестра.

Нотификаторы на события системного реестра устанавливаются функцией CmRegisterCallback:

```
NTSTATUS
CmRegisterCallback(
    IN PEX_CALLBACK_FUNCTION  Function,
    IN PVOID  Context,
```

```
    OUT PLARGE_INTEGER  Cookie
);
```

Параметр Function - указатель на функцию-нотификатор. В переменной Context можно указать данные, которые будут передаваться нотификатору в одноименном параметре при его вызове. Cookie - уникальный числовой идентификатор устанавливаемого нотификатора, используемый для его удаления функцией CmUnRegisterCallback.

Функция-нотификатор имеет следующий вид:

```
NTSTATUS
RegistryCallback(
    IN PVOID  CallbackContext,
    IN REG_NOTIFY_CLASS  Argument1,
    IN PVOID  Argument2
);
```

В качестве параметров Argument1 и Argument2 передаются код типа события, для которого был вызван идентификатор, и ассоциированные с этим типом данные. Событий, которые могут обрабатываться нотификатором, существует огромное множество - от открытия ключа и до перечисления его подключей или параметров. Кроме того, события делятся на пре-нотификационные (те, для которых нотификатор вызывается перед осуществлением какого-либо действия) и пост-нотификационные (нотификатор вызывается уже по факту выполнения какого-либо действия). Подробное описание этих параметров можно найти в документации к Windows Driver Development Kit.

Список нотификаторов, установленных этой функцией, хранится в переменной CmpCallBackVector, а их количество - в CmpCallBackCount. Размер списка определяется константой CM_MAX_CALLBACKS, которая, в отличие от ранее рассмотренных нотификаторов, равняется 100. Код установки нотификатора приведен в Листинге 3.

## Автоматизация разбора списков нотификаторов

Отладчик WinDbg имеет свой скриптовый язык. Скрипты представляют собой текстовые файлы, содержащие обычные команды отладчика и управляющие директивы: .if, .else, .elsif, .foreach, .for, .while, .do, .break, .continue, .catch, .leave, .printf, .block. В качестве переменных могут использоваться псевдорегистры ($tn, где n - целое число), назначаемые командой r, а также алиасы. [2] Для логических выражений есть как MASM-подобный, так и C-подобный синтаксис (в последнем случае выражение обрамляется в конструкцию @@c++(expression)).

Для того, что бы продемонстрировать возможности данного языка, я приведу пример очень простого скрипта, выводящего список всех нотификаторов. (См. Листинг 4).

Для запуска скрипта необходимо сохранить его в произвольный текстовый файл и выполнить в отладчике команду:

```
$$><C:\dbg_show_callbacks.txt
```

Здесь 'C:\dbg_show_callbacks.txt' - путь к нужному скрипту.

Пример вывода скрипта отображен на Рис. 4.

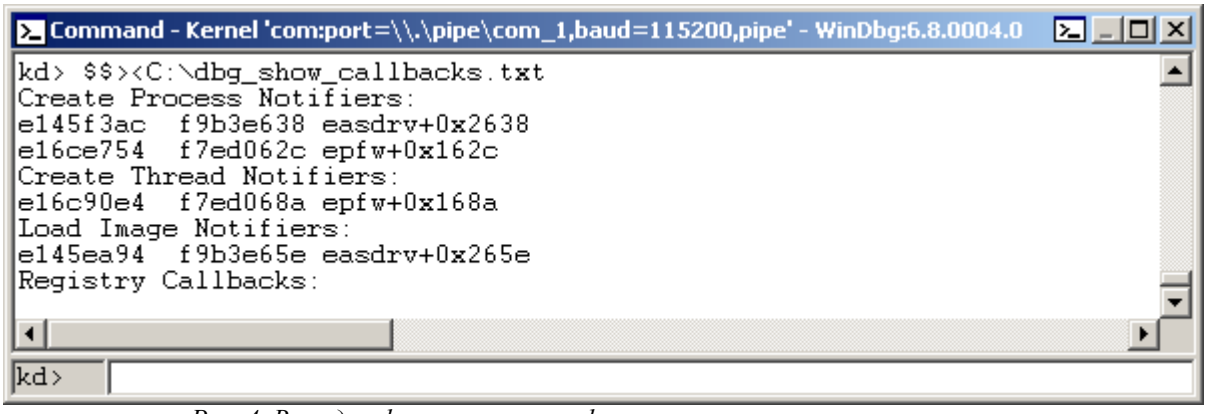


*Рис. 4. Вывод информации о нотификаторах с помощью скрипта.*

### *Отключение всех нотификаторов*

Альтернатива ручному исправлению списков нотификаторов с целью нейтрализации вредоносного кода - полное отключение механизма нотификации.

Как я уже упоминал, существуют механизмы для синхронизации доступа к элементам EX_CALLBACK массива нотификаторов (т.н. механизмы rundown protection). При использовании какого-либо нотификатора вызывающий его код выполняет инкрементирование количества ссылок на EX_CALLBACK_ROUTINE_BLOCK. Для этого при переборе массива нотификаторов он вызывает функцию ExReferenceCallBackBlock, проверяя её возвращаемое значение.

Как видно из Листинга 5, функция-нотификатор никогда не будет вызвана, если ExReferenceCallBackBlock вернёт нулевое значение. Поэтому всё, что необходимо сделать для отключения нотификаторов - это модифицировать функцию ExReferenceCallBackBlock так, чтобы она всегда возвращала 0.

Так как данная функция имеет тип вызова stdcall и принимает на вход всего один параметр (указатель на элемент типа EX_CALLBACK), для этого достаточно заменить её первые байты последовательностью инструкций xor eax,eax; ret 4.

Пролог функции до патчинга:

```
kd> u nt!ExreferenceCallbackBlock
nt!ExReferenceCallBackBlock:
80601c34 8bff        mov     edi,edi
80601c36 55          push    ebp
80601c37 8bec        mov     ebp,esp
80601c39 51          push    ecx
80601c3a 51          push    ecx
80601c3b 53          push    ebx
80601c3c 8b4508      mov     eax,dword ptr [ebp+8]
80601c3f 8b18        mov     ebx,dword ptr [eax]
```

Правим первые 8 байт функции:

```
kd> ed nt!ExreferenceCallbackBlock 04c2c033
kd> ed nt!ExreferenceCallbackBlock+4 90909000
```

А так выглядит функция после патчинга:

```
kd> u nt!ExreferenceCallbackBlock
nt!ExReferenceCallBackBlock:
80601c34 33c0        xor     eax,eax
80601c36 c20400      ret     4
80601c39 90          nop
80601c3a 90          nop
80601c3b 90          nop
80601c3c 8b4508      mov     eax,dword ptr [ebp+8]
80601c3f 8b18        mov     ebx,dword ptr [eax]
80601c41 f6c307      test    bl,7
```

Подобная модификация структур ядра при помощи отладчика настолько же безопасна, насколько обычный сплайсинг: проблемы возможны только в том случае, если в момент остановки системы по Ctrl+Break какой-либо поток исполнял код, который мы собираемся модифицировать. Так как ExReferenceCallbackBlock вызывается сравнительно редко - вероятность краха системы после патчинга пренебрежительно мала. Однако стоит иметь в виду, что подобные манипуляции могут привести к частичной или полной неработоспособности абсолютно всех драйверов, использующих нотификаторы. Таких драйверов достаточно мало, а в "чистой" операционной системе с конфигурацией по умолчанию - нет вовсе.

## Диспетчер системных вызовов

Одним из ключевых и самых главных механизмов взаимодействия между компонентами пользовательского режима и режима ядра ОС является механизм системных вызовов. Поскольку обращение практически к любой API-функции в конечном итоге приведёт к генерации соответствующих системных вызовов, руткит может контролировать всю операционную систему посредством контроля диспетчера системных вызовов.

### *Обработка системного вызова*

Рассмотрим, как происходит процедура обработки вызова какой-нибудь API-функции на примере CreateFileA из kernel32.dll. (См. Рис. 5)

Архитектура системного API в операционных системах семейства NT имеет иерархическую организацию. Базовыми кирпичиками для большинства dll-библиотек являются функции Native API. Эти функции импортируются из ntdll.dll, и их имена начинаются с Zw* или Nt* префиксов. В режиме пользователя Zw* и Nt* функции (например, ZwCreateFile и NtCreateFile) имеют одну и ту же точку входа. Код всех этих функций представляет собой небольшие заглушки, задачей которых является осуществить переход текущего потока в режим ядра, где диспетчер системных вызовов, в свою очередь, после некоторых манипуляций передаёт управление аналогичной функции из ntoskrnl (собственно, ядро NT) или win32k (сердце графической подсистемы). Так выглядит упомянутая заглушка в коде ntdll.dll:

```
ntdll!NtCreateFile:
7c90d090 b825000000      mov     eax,25h
7c90d095 ba0003fe7f      mov     edx,
         offset     SharedUserData!SystemCallStub
(7ffe0300)
7c90d09a ff12            call    dword ptr [edx]
7c90d09c c22c00          ret     2Ch
```

Вышеприведенный код помещает в регистр eax номер системного вызова (о том, что это за номер, будет рассказано ниже) и вызывает код, указатель на который находится в SharedUserData!SystemCallStub. По этому указателю, в зависимости от типа процессора, находится код вызова инструкции sysenter (ntdll!KiFastSystemCall) или генерации программного прерывания 2Eh (ntdll!KiIntSystemCall). Стоит упомя-

нуть, что Windows 2000 и более ранние версии NT использовали исключительно вызов 2Eh, код которого, сохранен и в более поздних версиях Windows с целью обеспечения обратной совместимости с программами, использующими системные вызовы напрямую.

```
ntdll!KiFastSystemCall:
7c90e4f0 8bd4            mov     edx,esp
7c90e4f2 0f34            sysenter
ntdll!KiFastSystemCallRet:
7c90e4f4 c3              ret
ntdll!KiIntSystemCall:
7c90e500 8d542408        lea     edx,[esp+8]
7c90e504 cd2e            int     2Eh
7c90e506 c3              ret
```

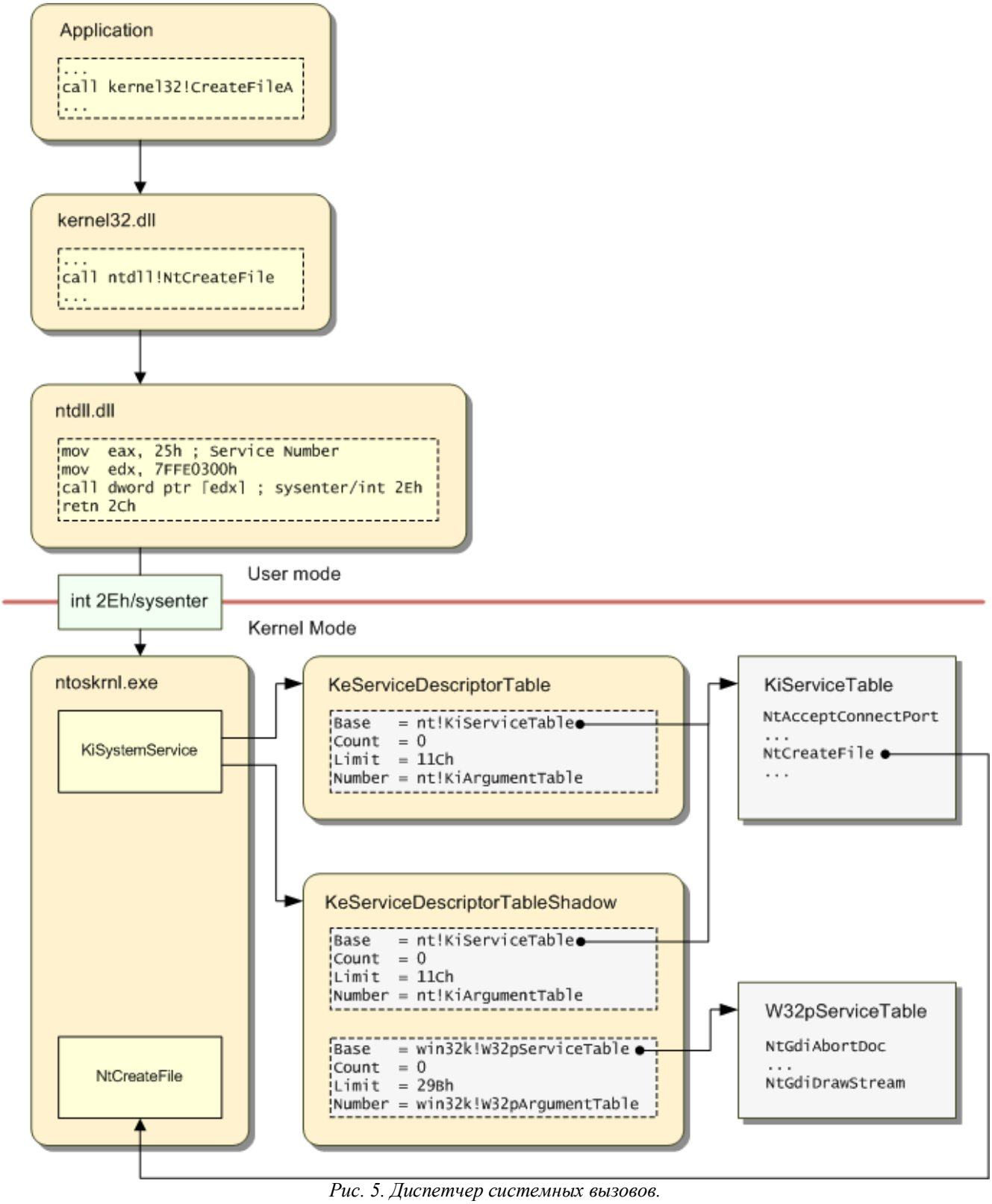


*Рис. 5. Диспетчер системных вызовов.*

В регистр edx, в обоих случаях, помещается указатель на область стека, в которой находятся переданные Native API функции аргументы. Вектор прерывания 2Eh обычно указывает на nt!KiSystemService, и его значение легко посмотреть в отладчике:

```
kd> !idt 2e
Dumping IDT:
2e:       8053c651 nt!KiSystemService
```

Вектор этого прерывания сохраняет контекст пользовательского режима, переключает стек приложения на стек режима ядра, а селектор fs со структуры TEB (Thread Environment Block) на структуру ядра KPCR (Kernel Processor Control Region). По завершению обработки прерывания вызывается инструкция iret, которая возвращает управление обратно в режим пользователя.

С инструкцией sysenter дело обстоит несколько сложнее. Обработчик её описывают следующие msr-регистры (model-specific registers):

| Название регистра | Номер | Назначение |
|---|---|---|
| IA32_SYSENTER_CS | 174h | Номер селектора кода (cs) |
| IA32_SYSENTER_ESP | 175h | Адрес стека, который будет установлен в esp |
| IA32_SYSENTER_EIP | 176h | Адрес обработчика (eip) |

*Таблица 1. Model Specific Registers*

Таким образом, при исполнении инструкции sysenter управление получает код, адрес которого (cs:eip) определяется описанными выше регистрами. Для возврата обратно, в режим пользователя, вызывается инструкция sysexit. Указатели на код ntdll!KiFastCallRet и стек пользовательского режима помещаются в регистры edx и ecx, соответственно.

Для чтения msr-регистров используется машинная инструкция rdmsr, а для записи - wrmsr. Команды отладчика для работы с данными регистрами называются аналогичным образом.

```
kd> rdmsr 174
msr[174] = 00000000`00000008
kd> rdmsr 175
msr[175] = 00000000`f9dcc000
kd> rdmsr 176
msr[176] = 00000000`8053c710
```

Код, находящийся по адресу 0x8053c710 (nt!KiFastCallEntry), также как и nt!KiSystemService, является точкой входа диспетчера системных вызовов.

Теперь рассмотрим, каким образом происходит обработка системного вызова в ядре. Очевидно, что диспетчеру системных вызовов необходимо найти адрес функции ядра по её номеру. Для этого, существует два дескриптора системных сервисов, которыми оперирует ядро: KeServiceDescriptorTable и KeServiceDescriptorTableShadow. Каждый их этих дескрипторов содержит 4 таблицы, из которых в KeServiceDescriptorTable используется только первая (для функций ядра) а в KeServiceDescriptorTableShadow дополнительно и вторая (для функций графической подсистемы). Формат структуры, которая описывает каждую таблицу, следующий:

```
typedef struct _KSERVICE_TABLE_DESCRIPTOR
{
    PULONG_PTR Base;
    PULONG Count;
    ULONG Limit;
    PUCHAR Number;

} KSERVICE_TABLE_DESCRIPTOR,
*PKSERVICE_TABLE_DESCRIPTOR;
```

Поле Base указывает на таблицу адресов функций (nt!KiServiceTable для функций ядра и win32k!W32pServiceTable для графической подсистемы), количество записей в которой определяется полем Limit. Переменная Number - указатель на массив, содержащий размер аргументов, принимаемых каждой функцией на стеке (nt!KiArgumentTable для функций ядра и win32k!W32pArgumentTable для графической подсистемы). Значения из этого массива диспетчер системных вызовов использует при копировании аргументов со стека пользовательского режима в стек ядра, перед тем как передать управление нужной функции. Поле Count используется только в отладочных сборках ядра, и обычно оно указывает на массив счётчиков использования для обработчиков системных вызовов.

Оба дескриптора находятся в секции данных ядра, однако, при обработке системного вызова, диспетчер получает указатель на дескриптор из поля ServiceTable структуры KTHREAD, описывающей текущий поток:

```
nt!KiFastCallEntry+0x8d:
; в eax - номер системного вызова
8053c79d 8bf8          mov  edi,eax
8053c79f c1ef08        shr  edi,8
8053c7a2 83e730        and  edi,30h
8053c7a5 8bcf          mov  ecx,edi
; esi - указывает на KTHREAD текущего потока
8053c7a7 03bee0000000 add edi,dword ptr [esi+0E0h]
8053c7ad 8bd8          mov  ebx,eax
8053c7af 25ff0f0000    and  eax,0FFFh
; проверка номера вызова относительно поля Limit
8053c7b4 3b4708        cmp  eax,dword ptr [edi+8]
8053c7b7 0f8345fdffff jae  nt!KiBBTUnexpectedRange
```

Это поле инициализируется при создании потока указателем на KeServiceDescriptorTable:

```
nt!KeInitThread+0x53:
805959f7 c786e000000080215580
     mov dword ptr [esi+0E0h],
     offset nt!KeServiceDescriptorTable (80552180)
```

Что происходит при вызове сервиса графической подсистемы, таблица системных сервисов которой находится только в KeServiceDescriptorTableShadow? Так как номера системных вызовов графической подсистемы начинаются с 1000h, приведенный выше код при проверке номера вызова выполнит переход на

nt!KiBBTUnexpectedRange, где, в случае если системный вызов должен быть обработан графической подсистемой, вызовется функция PsConvertToGuiThread:

```
nt!KiBBTUnexpectedRange:
8053c502 83f910          cmp     ecx,10h
8053c505 7539            jne     8053c540
8053c507 52              push    edx
8053c508 53              push    ebx
8053c509 e8f44a0800      call
nt!PsConvertToGuiThread
8053c50e 0bc0            or      eax,eax
8053c510 58              pop     eax
8053c511 5a              pop     edx
8053c512 8bec            mov     ebp,esp
8053c514 89ae34010000    mov     dword ptr
[esi+134h],ebp
8053c51a 0f847d020000    je      8053c79d
```

Функция PsConvertToGuiThread, в свою очередь, выделит нашему потоку больший стек переменного размера, и занесёт в поле SystemService структуры KTHREAD указатель на дескриптор KeServiceDescriptorTableShadow, во второй таблице которого будет содержаться нужный системный вызов.

## Zw* и Nt* функции

Как уже упоминалось, в режиме пользователя Zw* и Nt* функции (ZwCreateFile и NtCreateFile, например) будут иметь одну и ту же точку входа в ntdll.dll. Однако в режиме ядра это будут две совершенно разные функции. В чём различия между ними? Функции, имена которых начинаются на Zw*, вызываются исключительно из кода драйверов режима ядра. Они не выполняют проверку входных параметров, проверку прав доступа и другие манипуляции, необходимые при обработке системных вызовов от кода пользовательского режима. Код ZwCreateFile выглядит так:

```
nt!ZwCreateFile:
80500010 b825000000      mov     eax,25h
80500015 8d542404        lea     edx,[esp+4]
80500019 9c              pushfd
8050001a 6a08            push    8
8050001c e830140400      call
nt!KiSystemService (80541451)
80500021 c22c00          ret     2Ch
```

Он практически аналогичен коду ZwCreateFile в ntdll.dll за тем исключением, что вместо инструкции sysenter выполняется прямой вызов KiSystemService. Структура KTHREAD, описывающая поток, имеет важный параметр под названием PreviousMode, значение которого устанавливается диспетчером системных вызовов в зависимости от того, был ли он вызван из режима пользователя или из режима ядра. Это значение проверяется в коде Nt* функции, которая является обработчиком системного вызова, и если оно равняется KernelMode - упомянутые выше проверки безопасности пропускаются.

## Методика перехвата системных вызовов

Теперь, зная общие принципы работы диспетчера системных вызовов, рассмотрим способы, с помощью которых руткит может взять его под контроль.

- Подмена указателя на функцию в KiServiceTable.
- Подмена значения msr-регистра IA32_SYSENTER_CS/IA32_SYSENTER_EIP.
- Подмена указателя ServiceTable в структуре KTHREAD.
- Модификация ссылок на KeServiceDescriptorTable/KeServiceDescriptorTableShadow в функциях KeInitThread, PsConvertToGuiThread и других.
- Сплайсинг кода диспетчера системных вызовов.
- Сплайсинг функции-обработчика системного вызова.

Их не так уж и мало. На данный момент ни в одном антирутките или каком-либо другом продукте не реализовано полноценное детектирование всех этих техник перехвата.

Рассмотрим перечисленные техники по порядку.

## Подмена указателя на функцию в KiServiceTable

Это самый тривиальный способ перехвата системных вызовов, который можно встретить не только в "любительских" руткитах режима ядра, но и в очень многих защитных программах, таких как HIPS-ы, антивирусы, фаерволы и другие. Обнаруживаются подобные перехваты так же легко. Для этого нам потребуется таблица номеров системных вызовов для разных версии Windows. [3] Проведём ряд несложных манипуляций с отладчиком:

```
kd> dd nt!KeServiceDescriptorTable L4
80552180  80501030 00000000 0000011c 805014a4
```

Поля Base (nt!KeServiceDescriptorTable+0x00) и Number (nt!KeServiceDescriptorTable+0x0c) должны содержать корректные указатели на KiServiceTable и KiArgumentTable соответственно. Многие руткиты подменяют указатель на KiServiceTable значением, которое является указателем на копию данной таблицы. Также поле Base могут менять и некоторые программы, устанавливающие дополнительные системные вызовы для взаимодействия со своим драйвером (например, Kaspersky Internet Security версий 7.x). Восстановить эти указатели можно так:

```
kd> ed nt!KeServiceDescriptorTable
nt!KiServiceTable
kd> ed nt!KeServiceDescriptorTable+c
nt!KiArgumentTable
```

То же самое для Shadow-таблицы:

```
kd> ed nt!KeServiceDescriptorTableShadow
nt!KiServiceTable
kd> ed nt!KeServiceDescriptorTableShadow+c
nt!KiArgumentTable
```

Теперь взглянем на KiServiceTable:

```
kd> dps nt!KiServiceTable L 11c
80501030  8059849a nt!NtAcceptConnectPort
<skipped>
805010c0  8060bb94 nt!NtCreateEventPair
805010c4  f7cc5e4c aprtgsbrlm+0xe4c
<skipped>
80501130  805e950a nt!NtDeleteObjectAuditAlarm
80501134  f7cc6a9a aprtgsbrlm+0x1a9a
<skipped>
80501148  8060b84e nt!NtQueryBootOptions
8050114c  f7cc6484 aprtgsbrlm+0x1484
80501150  8060b310
nt!NtEnumerateSystemEnvironmentValuesEx
80501154  f7cc667d aprtgsbrlm+0x167d
<skipped>
80501200  8056e26a nt!NtOpenFile
80501204  f7cc5fde aprtgsbrlm+0xfde
80501208  805ca2ac nt!NtOpenJobObject
8050120c  f7cc6305 aprtgsbrlm+0x1305
80501210  8060c064 nt!NtOpenMutant
80501214  805e8fcc nt!NtOpenObjectAuditAlarm
80501218  f7cc5b30 aprtgsbrlm+0xb30
<skipped>
8050122c  805b95f6 nt!NtOpenSymbolicLinkObject
80501230  f7cc5cb6 aprtgsbrlm+0xcb6
<skipped>
80501270  806062be nt!NtQueryDefaultUILanguage
80501274  f7cc6146 aprtgsbrlm+0x1146
<skipped>
80501408  8060a0ce nt!NtSetUuidSeed
8050140c  f7cc68b7 aprtgsbrlm+0x18b7
<skipped>
8050149c  805c007a
nt!NtQueryPortInformationProcess
```

Задача при поиске руткитов - обращать внимание на системные вызовы, обработчики которых не находятся в ntoskrnl.exe (т.е. без префикса nt!*). В данном примере это ссылки на модуль aprtgsbrlm. Воспользовавшись вышеупомянутой таблицей системных вызовов, выясняем, что руткитом были перехвачены следующие вызовы:

- NtCreateFile (f7cc5e4c).
- NtDeleteValueKey (f7cc6a9a).
- NtEnumerateKey (f7cc6484).
- NtEnumerateValueKey (f7cc667d).
- NtOpenKey (f7cc6305).
- NtOpenProcess (f7cc5b30).
- NtOpenThread (f7cc5cb6).
- NtQueryDirectoryFile (f7cc6146).
- NtSetValueKey (805c007a).

Восстанавливаются оригинальные адреса функций следующим образом:

```
kd>  ed  nt!KiServiceTable  +  0x25  *  4
nt!NtCreateFile
```

0x25, в данном примере, является индексом системного вызова NtCreateFile в KiServiceTable для Windows XP. Перехваты других функций снимаются аналогично.

## Подмена значения msr-регистров

О назначении msr-регистров, относящихся к инструкции sysenter, я писал выше. Восстановить оригинальные значения в них можно так:

```
kd> wrmsr 0x174 8
kd> wrmsr 0x176 nt!KiFastcallEntry
```

Данный перехват будет работать в операционных системах старше Windows XP. В Windows 2000 использовался исключительно перехват вектора прерывания 2Eh, который сейчас некоторые авторы руткитов также устанавливают для сохранения работоспособности своего кода под 2000 и NT 4.0. Так как WinDbg не представляет возможности удобным образом редактировать таблицу векторов прерываний, более простым способом снятия перехвата 2Eh будет патчинг кода обработчика, установленного руткитом, безусловным переходом на nt!KiSystemService.

## Подмена указателя ServiceTable в структуре KTHREAD

Для обнаружения такого перехвата понадобится перечислить все существующие в системе потоки и проверить значение поля ServiceTable в структуре KTHREAD, описывающей каждый из них. Так как совершать подобные манипуляции вручную достаточно долго, напишем скрипт, автоматизирующий этот процесс (см. Листинг 6).

После запуска скрипт выведет значение поля ServiceTable для всех системных потоков, показав дополнительное предупреждающее сообщение для тех из них, где это поле не указывает на KeServiceDescriptorTable или KeServiceDescriptorTableShadow. (См. Рис. 6). После обнаружения модификаций остаётся только проанализировать принадлежащую руткиту копию таблицы системных вызовов и снять установленные в ней перехваты.

Следует иметь в виду, что наличие подменённого указателя в KTHREAD::ServiceTable практически всегда означает и наличие каких-то дополнительных перехватов ключевых функций, в обработчиках которых этот указатель устанавливается. Поэтому, если на исследуемой машине обнаружена подмена указателей на таблицы сервисов, необходимо обратить особое внимание на проверку системных нотификаторов и перехватов, описанных в двух следующих разделах статьи.

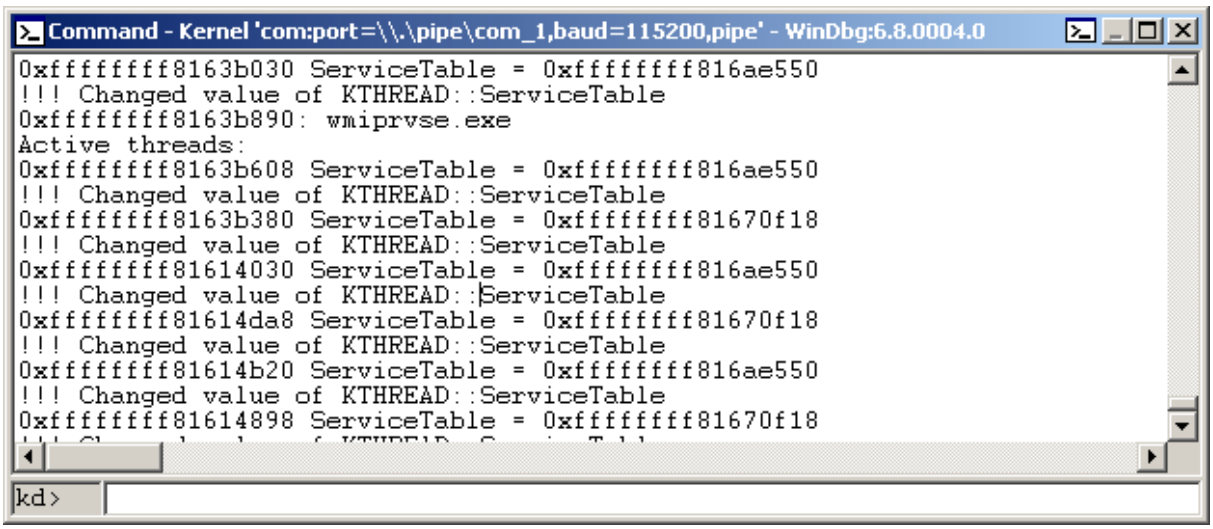


*Рис. 6. Поиск модификаций указателя KTHREAD::ServiceTable с помощью скрипта.*

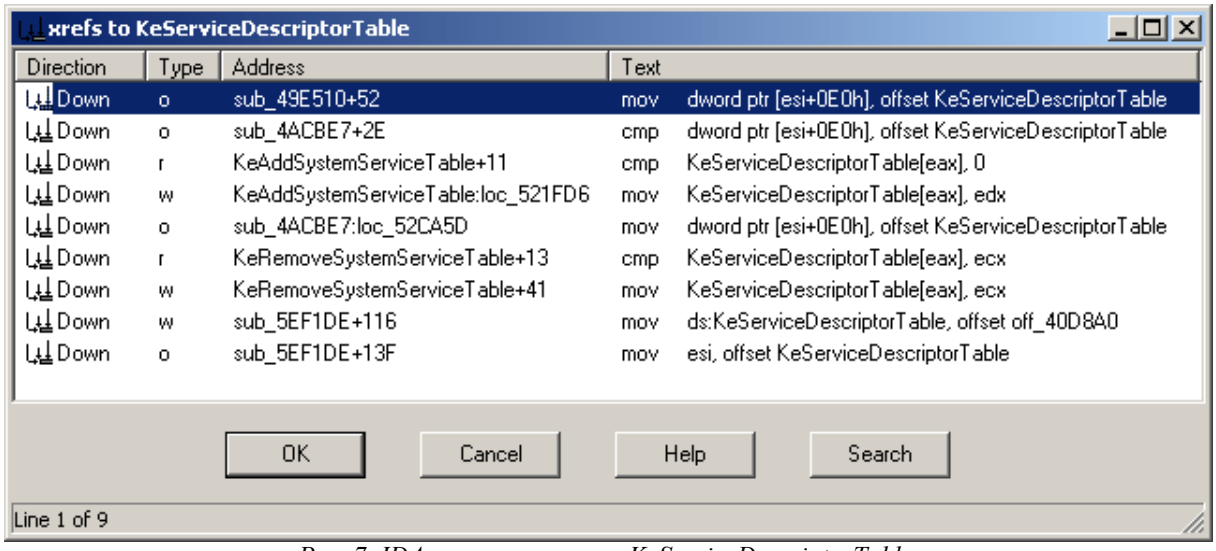


*Рис. 7. IDA нашла ссылки на KeServiceDescriptorTable.*

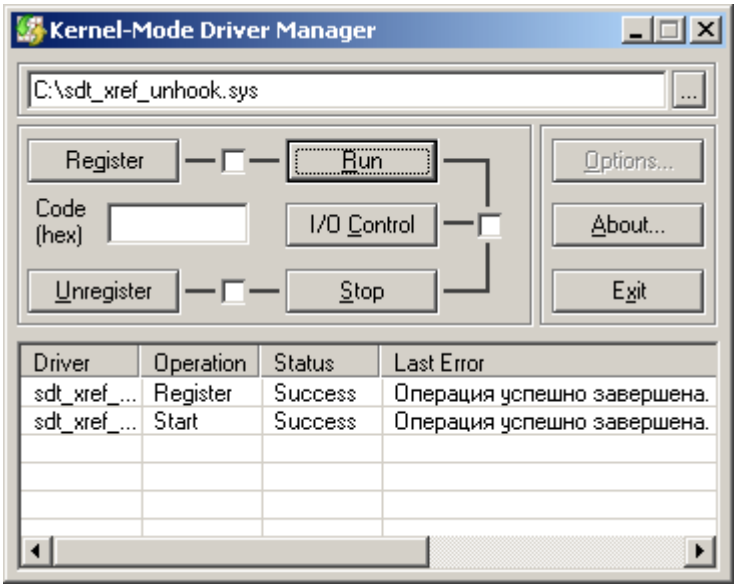


*Рис. 8. Загрузка драйвера с помощью утилиты KmdManager.*

## Модификация ссылок на KeServiceDescriptorTable в коде функций

Способ перехвата, описанный в предыдущем разделе, может быть всего лишь следствием подмены указателей на KeServiceDescriptorTable в коде ядра. Для осуществления такой подмены создают копии KeServiceDescriptorTable и KeServiceDescriptorTableShadow и исправляют все указатели на них в коде ядра (соответствующие смещения можно найти парсингом таблицы базовых поправок). Для проверки наличия или отсутствия такого перехвата достаточно просмотреть код двух основных функций: KeInitThread и PsConvertToGuiThread. Список всех ссылок на интересующую переменную можно получить при помощи IDA. (См. Рис. 7).

Для поиска и исправления таких модификаций можно использовать драйвер, который будет читать образ ядра с диска, искать ссылки на нужные переменные и исправлять их в загруженном образе ядра в случае обнаружения несоответствий. Полный исходный код драйвера доступен в приложении к статье.

Для загрузки драйверов удобно использовать утилиту KmdManager, которая входит в состав KmdKit от FourF. (См. Рис. 8).

Следует отметить, что хороший руткит может подсунуть подобному коду уже модифицированный образ ядра при попытке его чтения с диска. Поэтому наиболее достоверными являются результаты проверки, сделанной вручную с помощью отладчика.

## Сплайсинг кода диспетчера системных вызовов и функций-обработчиков

Сплайсинг – это способ перехвата, который основан на внесении изменений в машинный код перехватываемой функции. Самыми популярными мишенями для сплайсинг в диспетчере системных вызовов является nt!KiFastCallEntry и отсутствующая в отладочных символах метка KiSystemServiceRepeat. Однако не менее вероятна модификация и других участков кода диспетчера.

С самими Nt*-функциями, которые являются обработчиками соответствующих системных вызовов, дело обстоит несколько проще: перехват в подавляющем большинстве случаев устанавливается на начало функции, и он может представлять собой как обычную инструкцию jmp, так и целый блок инструкций, в результате исполнения которых будет выполнен переход на код обработчика перехвата.

Модификации кода ядра легче всего обнаружить методом сравнения кода в памяти с кодом образа ядра на диске. Для этих целей также можно использовать небольшой драйвер, приведенный в приложении (drivers/spl_detect). После запуска, он выводит в отладочный вывод информацию обо всех участках кода ядра в памяти, которые отличаются от соответствующих участков в файле ядра на диске.

Наличие модификаций в коде ядра не обязательно свидетельствует об активности руткита. Информация, приведенная на иллюстрации выше, вполне типична для чистой операционной системы. Найденные модификации – это так называемый self-patching: модификация ядром собственного кода. В качестве примеров самомодификации кода ядра можно привести функции, отображенные в Табл. 2.

В примере, приведенном на Рис. 9, патч nt!KiDispatchInterrupt+0x22a соответствует функции KeFlushCurrentTb, а патч nt!KeReleaseInStackQueuedSpinLockFromDpcLevel+0xc12 - функции KiSystemCallExitBranch.

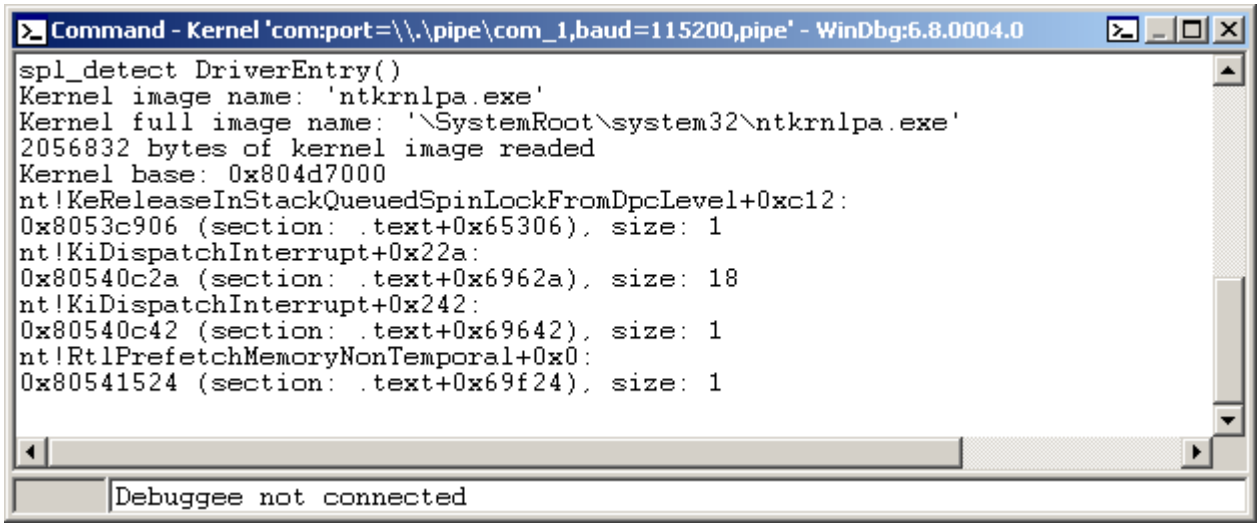


*Рис. 9. Поиск модификаций кода ядра.*

| Название функции | Откуда производится модификация кода | Цель патча |
|---|---|---|
| KeFlushCurrentTb | Ki386EnableGlobalPage | Патчинг выполняется в случае поддержки G-бита глобальности в PDE/PTE, который влияет на процесс кэширования страниц памяти. |
| RtlPrefetchMemoryNonTemporal | KiInitMachineDependent | Модифицируется в зависимости от того, поддерживает ли процессор prefetch-инструкцию или нет. |
| KiSystemCallExitBranch | KiEnableFastSyscallReturn<br>KiDisableFastSyscallReturn | Используется для переключения возможности возврата из обработчика системного вызова через sysexit, так как данная возможность может использоваться только тогда, когда инициализированы нужные для этой инструкции MSR-регистры на всех процессорах. |

*Таблица 2. Примеры самомодификации кода ядра*

## Файловая система

Перехват системных вызовов, помимо всего прочего, позволяет контролировать файловые операции с целью сокрытия файлов руткита или блокирования доступа к ним. Однако для достижения этих целей в руткитах применяется и ряд других способов. Для того, что бы уметь им противодействовать, рассмотрим организацию файловой подсистемы в Windows.

Файловая подсистема достаточно тесно переплетается с подсистемой ввода-вывода. Она имеет модульно-иерархическую структуру, функциональность отдельных уровней которой реализуется отдельными драйверами. Какие бывают типы драйверов относительно подсистемы ввода-вывода?

- Драйвера устройств (storage devices drivers) – это драйверы, объединяющие в себе функции управления конкретными аппаратными устройствами (такими как порты, шины, накопители). Большинство из них являются PnP драйверами: их загрузкой и управлением занимается PnP менеджер, для которого они представлены в виде дерева PnP устройств.
- Драйвера томов (storage volume drivers) – служат для управления томами, которые представляют собой разделы различных устройств хранения информации. Для взаимодействия с более низкоуровневыми частями дисковой подсистемы такой драйвер создает physical device object (PDO) - объект, представляющий определённый раздел. После того, как на данном разделе смонтирована файловая система, создаётся volume device object (VDO) - объект, представляющий смонтированный раздел для более высокоуровневых драйверов – драйверов файловой системы. Самим процессом монтирования управляет часть подсистемы ввода-вывода под названием «диспетчер монтирования» (mount manager). Когда PnP драйвера уведомляют его о нахождении нового тома в системе, диспетчер монтирования, в свою очередь, передаёт управление соответствующим механизмам драйверов разделов.
- Драйвера файловой системы (file system drivers) – реализуют работу с определёнными файловыми системами (такими как FAT32, NTFS, CDFS). Драйвера этого класса также создают два типа устройств: volume device object (VDO) и control device object (CDO). CDO является представлением конкретной файловой системы (в отличие от отдельного раздела). Эти устройства именованные (например, \Device\Ntfs). Для уведомления драйверов файловых систем о подключении нового раздела диспетчер ввода-вывода посылает им IRP запрос типа IRP_MJ_FILE_SYSTEM_CONTROL (IRP_MN_MOUNT_VOLUME). Получив этот IRP, драйвера файловых систем создают VDO для каждого отдельного раздела. VDO, в свою очередь, обрабатывают IRP запросы, которые диспетчер ввода-вывода отправляет им при вызове какой-либо файловой функции (таких как NtCreateFile, NtReadFile и других).

Набор всех этих драйверов, участвующих в обработке файловых операций и управлении устройствами, называется стеком драйверов устройств хранения информации. Их взаимодействие между собой обеспечивает специальная часть ядра Windows, называемая диспетчером ввода-вывода. Схема организации драйверов хранения информации выглядит так (см. Рис. 10).

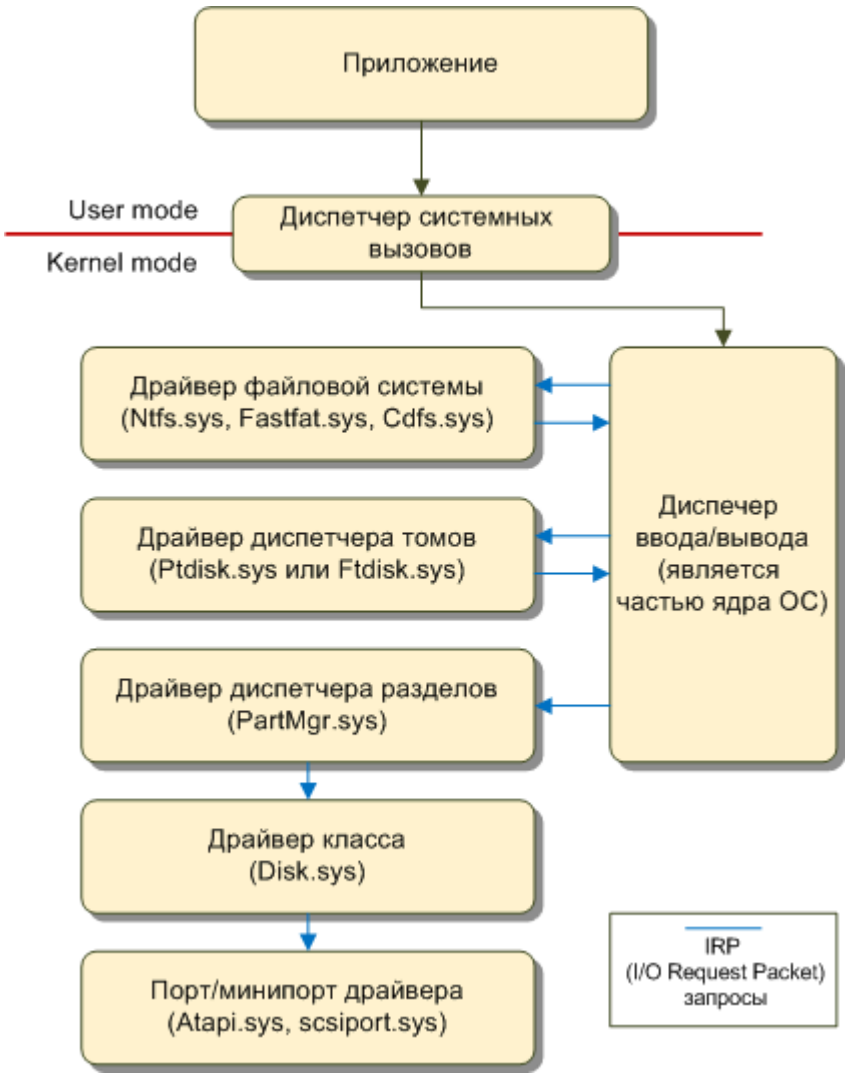


*Рис. 10. Условная схема взаимодействия драйверов хранения информации.*

При вызове какой-либо Native API функции, связанной с обработкой файлов (например NtCreateFile, NtWriteFile, NtQueryDirectoryFile), диспетчер ввода/вывода создаёт IRP-запрос, который отправляется драйверу файловой системы. Драйвер файловой системы выполняет парсинг главной файловой таблицы при открытии или перечислении файлов, а при чтении/записи в файл по определённому смещению транслирует это смещение в смещение тома, после чего отправляет запрос на чтение/запись сектора драйверу ниже по стеку. Драйвер диспетчера томов необходим для предоставления драйверу файловой системы устройства, управляющего работой конкретного тома (такие устройства, как правило, имеют имена вида \Device\HarddiskVolumeN). Драйвер диспетчера разделов отвечает за уведомление диспетчера Plug and Play о наличии разделов, благодаря чему драйвера томов получают соответствующие уведомления. В конечном итоге, запрос на чтение/запись сектора доставляется драйверу класса диска, который инициирует дисковый ввод-вывод на уровне секторов, поддерживаемый порт- и минипорт-драйверами, управляющими оборудованием. Драйвер класса диска также создаёт устройства, представляющие все разделы на физических дисках (пример имени такого устройства \Device\Harddisk0\DP(1)0x7e00-0x288c34200+1). Порт- и минипорт-драйверы, которые завершают обработку цепочки запроса на выполнение дисковой операции, довольно часто предоставляются разработчиками самого устройства. Однако в состав Windows входят и некоторые стандартные порт-драйвера: например, Scsiport.sys, отвечающий за работу с SCSI-шиной, или Atapi.sys – порт-драйвер для IDE.

Очевидно, что руткиту удобнее всего устанавливать свой перехват на уровне драйверов файловых систем – все файловые операции можно перехватывать «как есть», без необходимости реализовывать самостоятельный парсинг структур файловой системы в данных перехватываемых запросов. Хотя это не исключает и возможность функционирования перехватов и на более низких уровнях: можно довольно легко получить посекторную карту размещения файла на диске, после чего блокировать чтение/запись этих секторов хоть на уровне минипорт-драйвера. Независимо от уровня, на котором устанавливается перехват в стеке устройств хранения информации, он может осуществляться тремя основными способами:

- Присоединение к целевому устройству устройства-фильтра.
- Подмена указателей на IRP (или Fast IO) обработчики в дескрипторе драйвера, управляющего целевым устройством.
- Сплайсинг кода IRP (или Fast IO) обработчиков драйвера, управляющего целевым устройством.

Техника сплайсинга и методы ее обнаружения рассматривалась в прошлой главе на примере ядра. Однако, два первых способа касаются исключительно техник фильтрации IRP запросов, из-за чего требуют детального рассмотрения.

## Устройства-фильтры

Для фильтрации IRP запросов к определённому устройству разработчики NT архитектуры предусмотрели возможность присоединения (аттача) к нему других устройств, которые будут играть роль фильтра относительно друг друга. В этом случае структуры DEVICE_OBJECT, описывающие устройства, будут объединяться в односвязный список по полю AttachedDevice, а называется этот список device stack. При отправке IRP-запроса какому-либо устройству функция IoCallDriver получает указатель на вершину этого стека и вызывает функцию-обработчик, принадлежащую крайнему устройству в списке. Это устройство, в свою очередь, может завершить полученный IRP запрос вызовом IoCompleteRequest, либо передать его устройству ниже по стеку, вызвав IoSkipCurrentIrpStackLocation и IoCallDriver.

Отладчик WinDBG располагает мощными средствами для работы с объектами ядра типа «драйвер» и «устройство». Рассмотрим пример обнаружения и нейтрализации руткита, выполняющего аттач устройства-фильтра к Volume Device Object (VDO) драйвера файловой системы NTFS.

Получим список всех устройств интересующего нас драйвера файловой системы:

```
kd> !drvobj \FileSystem\Ntfs
Driver object (8192aa18) is for:
 \FileSystem\Ntfs
Driver Extension List: (id , addr)

Device Object list:
818f8030  8192a8f0
```

Для просмотра стека устройств, ассоциированного с каждым конкретным устройством этого драйвера, используем команду !devstack:

```
kd> !devstack 818f8030
  !DevObj   !DrvObj            !DevExt   ObjectName
  81749030                     00000000
> 818f8030  \FileSystem\Ntfs  818f80e8
kd> !devstack 8192a8f0
  !DevObj   !DrvObj            !DevExt   ObjectName
> 8192a8f0  \FileSystem\Ntfs  00000000  Ntfs
```

VDO обычно представлен безымянным устройством (818f8030). На приведенном выше примере к нему присоединено устройство-фильтр, принадлежащее руткиту (81749030). Перед отсоединением этого фильтра важно убедиться в том, что на данный момент времени устройствами не обрабатывается какой-либо IRP запрос, иначе вмешательство в их стек может повлечь за собой крах операционной системы:

```
kd> !devobj 818f8030
Device object (818f8030) is for:
  \FileSystem\Ntfs DriverObject 8192aa18
Current Irp 00000000 RefCount 0 Type 00000008
Flags 00000000
DevExt 818f80e8 DevObjExt 818f8890
ExtensionFlags (0000000000)
AttachedDevice (Upper) 81749030
Device queue is not busy.
```

Указатель на Current Irp должен быть нулевым - в противном случае, требуется возобновить работу системы по F5 из окна отладчика, и повторить все манипуляции через некоторое время, по завершению всех операций дискового ввода-вывода. Отсоединяется фильтр (или вся цепочка, при наличии нескольких присоединенных устройств) посредством обнуления поля AttachedDevice в структуре, описывающей VDO устройство.

```
kd> dt _DEVICE_OBJECT 818f8030
ntdll!_DEVICE_OBJECT
   +0x000 Type             : 3
   +0x002 Size             : 0x860
   +0x004 ReferenceCount   : 0
   +0x008 DriverObject : 0x8192aa18 _DRIVER_OBJECT
   +0x00c NextDevice   : 0x8192a8f0 _DEVICE_OBJECT
   +0x010 AttachedDevice
                       : 0x81749030 _DEVICE_OBJECT
   +0x014 CurrentIrp   : (null)
   +0x018 Timer        : (null)
   +0x01c Flags        : 0
   +0x020 Characteristics  : 0
   +0x024 Vpb          : (null)
   +0x028 DeviceExtension  : 0x818f80e8
   +0x02c DeviceType   : 8
   +0x030 StackSize    : 7 ''
   +0x034 Queue        : __unnamed
   +0x05c AlignmentRequirement : 1
   +0x060 DeviceQueue  : _KDEVICE_QUEUE
   +0x074 Dpc          : _KDPC
   +0x094 ActiveThreadCount : 0
   +0x098 SecurityDescriptor : (null)
   +0x09c DeviceLock   : _KEVENT
   +0x0ac SectorSize   : 0x200
   +0x0ae Spare1       : 1
   +0x0b0 DeviceObjectExtension
                     : 0x818f8890 _DEVOBJ_EXTENSION
   +0x0b4 Reserved     : (null)
kd> ed 818f8030+0x010 0
```

## Минифильтр-драйвера файловой системы

Просматривая стек устройств драйвера файловой системы на реальном компьютере, можно заметить, что к VDO почти всегда присоединено устройство, принадлежащее драйверу fltMgr.sys. Данный драйвер реализует функциональность минифильтр-драйверов файловой системы (file system mini filter drivers), предоставляя им API функции вида FltXxx. [4] Это API также может

быть использовано руткитом для перехвата событий обращения к дисковой подсистеме. Для отключения всех минифильтров файловой системы достаточно отсоединить принадлежащее fltMgr.sys устройство от VDO устройства драйвера файловой системы. На практике подобные манипуляции совершенно безопасны, и могут привести разве что к временной неработоспособности службы восстановления системы.

## Подмена указателей на IRP обработчики

Для обработки обращений к созданным устройствам драйвер на этапе инициализации ассоциирует со своим объектом набор IRP обработчиков. Эти функции вызываются диспетчером ввода-вывода при осуществлении определённых операций с устройством (таких как открытие, закрытие, чтение, запись и т.д.), а также по факту некоторых системных событий (таких как завершение работы системы или монтирование раздела жесткого диска). Адреса этих обработчиков хранятся в поле MajorFunction структуры DRIVER_OBJECT, которая описывает конкретный загруженный драйвер. Данное поле является массивом указателей с фиксированным размером IRP_MJ_MAXIMUM_FUNCTION + 1. Константа IRP_MJ_MAXIMUM_FUNCTION определена в заголовочных файлах Driver Development Kit-а как 27. IRP-обработчики имеют следующий тип:

```
typedef
NTSTATUS
(*PDRIVER_DISPATCH) (
    IN struct _DEVICE_OBJECT *DeviceObject,
    IN struct _IRP *Irp
);
```

Для получения контроля над нужным устройством руткит может подменить значения этих указателей на свои и, далее, либо модифицировать буферы с данными, которые передаются в IRP запросах к устройству, либо вообще не передавать управления оригинальным обработчикам, завершая «нежелательные» IRP запросы.

Для просмотра таблицы IRP обработчиков нужного устройства используем команду !drvobj со значением параметра Flags равным 2 (см. Листинг 7).

Для обнаружения активных перехватов в данном случае достаточно обращать внимание на те адреса, для которых либо не определилось корректно символьное имя функции (при условии, что отладочные символы для целевого драйвера корректно загружены), либо адрес функции указывает в модуль, отличный от самого драйвера или ядра. На приведенном выше примере для драйвера файловой системы NTFS руткитом перехвачены обработчики IRP_MJ_CREATE (для контроля над операциями создания и открытия файлов) и IRP_MJ_DIRECTORY_CONTROL (для контроля над операциями получения содержимого каталога). Задачу восстановления оригинальных адресов IRP обработчиков для драйверов файловых систем облегчает тот факт, что все их имена присутствуют в отладочных символах. Имена для основных IRP обработчиков драйверов файловых систем NTFS и FAT приведены в Таблице 3.

Сам процесс восстановления указателей на обработчики выглядит так (на примере перехваченных обработчиков, которые фигурировали в предыдущем выводе команды !drvobj):

```
kd> !drvobj \FileSystem\Ntfs
Driver object (8192b1f8) is for:
 \FileSystem\Ntfs
Driver Extension List: (id , addr)

Device Object list:
818f7030  818f9040
kd> dt _DRIVER_OBJECT 8192b1f8
nt!_DRIVER_OBJECT
   +0x000 Type             : 4
   +0x002 Size             : 168
   +0x004 DeviceObject
            : 0x818f7030 _DEVICE_OBJECT
   +0x008 Flags            : 0x92
   +0x00c DriverStart      : 0xf9675000
   +0x010 DriverSize       : 0x8c480
   +0x014 DriverSection    : 0x819f17e0
   +0x018 DriverExtension
            : 0x8192b2a0 _DRIVER_EXTENSION
   +0x01c DriverName
            : _UNICODE_STRING "\FileSystem\Ntfs"
   +0x024 HardwareDatabase
            : 0x8066e9d8 _UNICODE_STRING
   "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch
            : 0xf9694820 _FAST_IO_DISPATCH
   +0x02c DriverInit
            : 0xf96fa204 long Ntfs!GsDriverEntry+0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : (null)
   +0x038 MajorFunction
            : [28] 0xf9ef4d2c     long  +0
kd> ed 8192b1f8+0x038+4*0x00 Ntfs!NtfsFsdCreate
kd> ed 8192b1f8+0x038+4*0x0c
                  Ntfs!NtfsFsdDirectoryControl
```

Значения 0x00 и 0x0c в двух последних командах ed являются номерами нужных нам обработчиков - IRP_MJ_CREATE и IRP_MJ_DIRECTORY_CONTROL, соответственно.

| Обработчик | Имя для Ntfs.sys | Имя для Fastfat.sys |
|---|---|---|
| IRP_MJ_CREATE | Ntfs!NtfsFsdCreate | Fastfat!FatFsdCreate |
| IRP_MJ_CLOSE | Ntfs!NtfsFsdClose | Fastfat!FatFsdClose |
| IRP_MJ_READ | Ntfs!NtfsFsdRead | Fastfat!FatFsdRead |
| IRP_MJ_WRITE | Ntfs!NtfsFsdWrite | Fastfat!FatFsdWrite |
| IRP_MJ_QUERY_INFORMATION | Ntfs!NtfsFsdDispatchWait | Fastfat!FatFsdQueryInformation |
| IRP_MJ_SET_INFORMATION | Ntfs!NtfsFsdSetInformation | Fastfat!FatFsdSetInformation |
| IRP_MJ_DIRECTORY_CONTROL | Ntfs!NtfsFsdDirectoryControl | Fastfat!FatFsdDirectoryControl |
| IRP_MJ_FILE_SYSTEM_CONTROL | Ntfs!NtfsFsdFileSystemControl | Fastfat!FatFsdFileSystemControl |
| IRP_MJ_DEVICE_CONTROL | Ntfs!NtfsFsdDispatch | Fastfat!FatFsdDeviceControl |

*Таблица 3. Имена основых IRP-обработчиков драйверов файловых систем*

Иногда бывает необходимо восстановить адреса IRP обработчиков драйвера, для которого нет отладочных символов. Для этого загрузим исполняемый файл драйвера в дизассемблер с целью ручного нахождения адресов. Обычно заполнению массива MajorFunction структуры DRIVER_OBJECT предшествует создание устройства, которое находится в коде анализируемого файла около вызова импортируемой функции IoCreateDevice. В большинстве случаев эти манипуляции происходят в непосредственной близости к точке входа модуля. Листинг 8 демонстрирует процедуру заполнения массива IRP обработчиков в коде драйвера Ntfs.sys.

Дизассемблирование и поиск нужного участка кода можно выполнить и в отладчике. Однако, очень часто во время линковки исполняемого модуля функции, выполняющие подобные «одноразовые» инициализационные операции, помещаются в секцию файла, имеющую discardable атрибут. Это, в свою очередь, значит, что после успешной загрузки драйвера и вызова его точки входа такая секция будет выгружена из памяти с целью экономии системных ресурсов.

## Другие способы перехвата IRP запросов

Помимо уже рассмотренных способов перехвата IRP запросов к устройству есть и некоторые другие типичные техники, не имеющие отношения к особенностям функционирования диспетчеров системных сервисов и ввода-вывода, но, тем не менее, весьма часто применяющиеся в руткитах. Для работы с IRP запросами ядром Windows, среди всего прочего, используются две важные функции: IofCallDriver (отправка IRP произвольному драйверу) и IofCompleteRequest (завершение IRP запроса).

Взглянем на их код:

```
kd> u nt!IofCallDriver
nt!IofCallDriver:
804ede00 ff2500b45480 jmp dword ptr
                    [nt!pIofCallDriver (8054b400)]
...
```

```
kd> u nt!IofCompleteRequest
nt!IofCompleteRequest:
804ede90 ff2504b45480 jmp dword ptr
                [nt!pIofCompleteRequest (8054b404)]
```

Он представляет собой небольшие заглушки, которые передают управление по адресам, хранящимся в глобальных переменных ядра pIofCallDriver и pIofCompleteRequest.

```
kd> dd nt!pIofCallDriver L1
8054b400  804eddc8
kd> u 804eddc8
nt!IopfCallDriver:
804eddc8 fe4a23          dec     byte ptr
[edx+23h]
804eddcb 8a4223          mov     al,byte ptr
[edx+23h]
...
kd> dd nt!pIofCompleteRequest L1
8054b404  804f02c0
kd> u 804f02c0
nt!IopfCompleteRequest:
804f02c0 8bff            mov     edi,edi
804f02c2 55              push    ebp
804f02c3 8bec            mov     ebp,esp
804f02c5 83ec10          sub     esp,10h
```

На «чистой» операционной системе эти указатели проинициализированы адресами fastcall функций IopfCallDriver и IopfCompleteRequest, соответственно. Инициализацию переменных pIofCallDriver и pIofCompleteRequest выполняет функция IopSetIoRoutines, которая вызывается в коде ядра ближе к началу IoInitSystem.

```
; int __stdcall IopSetIoRoutines()
_IopSetIoRoutines@0 proc near
xor     eax, eax
cmp     _pIofCallDriver, eax
jnz     short loc_8068414A
mov     _pIofCallDriver, offset @IopfCallDriver@8
loc_8068414A:
cmp     _pIofCompleteRequest, eax
jnz     short loc_8068415C
mov     _pIofCompleteRequest, offset
                          @IopfCompleteRequest@8
loc_8068415C:
cmp     _pIoAllocateIrp, eax
jnz     short loc_8068416E
mov     _pIoAllocateIrp, offset
```

```
                                _IopAllocateIrpPrivate@8
loc_8068416E:
cmp     _pIoFreeIrp, eax
jnz     short locret_80684180
mov     _pIoFreeIrp, offset _IopFreeIrp@4
locret_80684180:
retn
_IopSetIoRoutines@0 endp
```

Эти манипуляции с указателями позволяет установить несколько простых перехватов, которые будут контролировать обработку всех IRP запросов в системе. Так как подобные перехваты устанавливаются обычным патчингом указателя, по которому осуществляется переход, они имеют следующие преимущества:

- В отличие от сплайсинга, они совершенно безопасны и могут быть в любой момент сняты и установлены вновь без риска вызвать крах операционной системы.
- Не детектируются большинством публично доступными антируткитов, оснащённых функцией поиска модификаций кода.

Снимаются подобные перехваты следующими командами:

```
kd> ed nt!pIofCallDriver nt!IopfCallDriver
kd> ed nt!pIofCompleteRequest
nt!IopfCompleteRequest
```

## Объекты, типы объектов и их дескрипторы

В NT архитектуре все ресурсы операционной системы представлены в виде объектов ядра, а подсистема, которая управляет объектами, называется диспетчером объектов. Диспетчер объектов, так же как и диспетчер системных вызовов, тоже может являться целью руткита. Взять диспетчер объектов под контроль можно перехватом нужных Ob* функций ядра (которые и составляют код диспетчера объектов). На практике кроме подобных перехватов встречаются и более изощрённые приёмы.

Каждый объект представляет собой структуру, находящуюся в памяти режима ядра, которую условно можно разделить на две составные части: заголовок (служебная структура, которая содержит информацию для диспетчера объекта) и тело объекта (заполняется в зависимости от нужд подсистемы, создавшей объект). Заголовок объекта описывается структурой OBJECT_HEADER. Важной характеристикой абсолютно любого объекта является его тип. Тип - это тоже указатель на структуру (она называется OBJECT_TYPE), причём тип объекта сам по себе является экземпляром объекта типа «тип», а структура OBJECT_TYPE есть не что иное, как тело этого объекта типа «тип». Для программиста все эти особенности совершенно прозрачны: в большинстве случаев, работа с объектами происходит через системные сервисы, которые ссылаются на тот или иной объект по ассоциированному с ним дескриптору (HANDLE), а внутренние механизмы диспетчеризации объектов скрыты от посторонних глаз и рук. Все объекты типа «тип» отображаются на пространство имён диспетчера объектов в директории ObjectTypes. Всего их существует пара десятков (см. Листинг 9).

Структура, описывающая заголовок объекта, выглядит следующим образом:

```
typedef struct _OBJECT_HEADER
{
    LONG_PTR PointerCount;
    union {
        LONG_PTR HandleCount;
        PVOID NextToFree;
    };
    POBJECT_TYPE Type;
    UCHAR NameInfoOffset;
    UCHAR HandleInfoOffset;
    UCHAR QuotaInfoOffset;
    UCHAR Flags;

    union {
        POBJECT_CREATE_INFORMATION
                                    ObjectCreateInfo;
        PVOID QuotaBlockCharged;
    };

    PSECURITY_DESCRIPTOR SecurityDescriptor;
    QUAD Body;

} OBJECT_HEADER,
*POBJECT_HEADER;
```

Важным её полем является указатель Type, который ссылается на тело объекта «тип» (структура OBJECT_TYPE).

Так выглядит объект типа «драйвер» и тело объекта, который описывает этот тип:

```
kd> !drvobj \Driver\Beep
Driver object (81f2e678) is for:
 \Driver\Beep
Driver Extension List: (id , addr)
Device Object list:
81f2e4f8
kd> !object 81f2e678
Object: 81f2e678  Type: (821b85b8) Driver
    ObjectHeader: 81f2e660 (old version)
    HandleCount: 0  PointerCount: 3
    Directory Object: e13a0948  Name: Beep
kd> dt _OBJECT_TYPE 821b85b8
ntdll!_OBJECT_TYPE
   +0x000 Mutex          : _ERESOURCE
   +0x038 TypeList
          : _LIST_ENTRY [ 0x821b82b8 - 0x81dd7018 ]
   +0x040 Name           : _UNICODE_STRING "Driver"
   +0x048 DefaultObject  : 0x805588c0
   +0x04c Index          : 0x1a
   +0x050 TotalNumberOfObjects : 0x62
   +0x054 TotalNumberOfHandles : 0
   +0x058 HighWaterNumberOfObjects : 0x62
   +0x05c HighWaterNumberOfHandles : 1
   +0x060 TypeInfo
                         : _OBJECT_TYPE_INITIALIZER
   +0x0ac Key            : 0x76697244
   +0x0b0 ObjectLocks    : [4] _ERESOURCE
```

Наибольший интерес представляет структура OBJECT_TYPE_INITIALIZER, которая является частью OBJECT_TYPE:

```
typedef struct _OBJECT_TYPE_INITIALIZER
{
    USHORT Length;
    BOOLEAN UseDefaultObject;
    BOOLEAN CaseInsensitive;
    ULONG InvalidAttributes;
    GENERIC_MAPPING GenericMapping;
    ULONG ValidAccessMask;
    BOOLEAN SecurityRequired;
    BOOLEAN MaintainHandleCount;
    BOOLEAN MaintainTypeList;
    POOL_TYPE PoolType;
    ULONG DefaultPagedPoolCharge;
    ULONG DefaultNonPagedPoolCharge;
    OB_DUMP_METHOD DumpProcedure;
    OB_OPEN_METHOD OpenProcedure;
    OB_CLOSE_METHOD CloseProcedure;
    OB_DELETE_METHOD DeleteProcedure;
    OB_PARSE_METHOD ParseProcedure;
    OB_SECURITY_METHOD SecurityProcedure;
    OB_QUERYNAME_METHOD QueryNameProcedure;
    OB_OKAYTOCLOSE_METHOD OkayToCloseProcedure;

} OBJECT_TYPE_INITIALIZER,
*POBJECT_TYPE_INITIALIZER;
```

В конце этой структуры находятся указатели на методы обьекта, которые представляют собой функции, вызываемые диспетчером на определённом этапе жизни обьекта. (См. Табл. 3).

Как правило, у каждого типа объекта есть свой собственный набор методов, которые определяются подсистемой, создавшей данный тип. Для многих типов объектов некоторые из этих методов не определены, например, вот так выглядит структура OBJECT_TYPE_INITIALIZER для типа объекта «устройство»:

```
kd> !object \ObjectTypes\Device
Object: 819b8ad0  Type: (819f1428) Type
    ObjectHeader: 819b8ab8 (old version)
    HandleCount: 0  PointerCount: 1
    Directory Object: e10004a0  Name: Device
kd> dt _OBJECT_TYPE 819b8ad0 TypeInfo.
ntdll!_OBJECT_TYPE
      +0x060 TypeInfo  :
      +0x000 Length    : 0x4c
      +0x002 UseDefaultObject : 0x1 ''
      +0x003 CaseInsensitive : 0x1 ''
      +0x004 InvalidAttributes : 0x100
      +0x008 GenericMapping : _GENERIC_MAPPING
      +0x018 ValidAccessMask : 0x1f01ff
      +0x01c SecurityRequired : 0 ''
      +0x01d MaintainHandleCount : 0 ''
      +0x01e MaintainTypeList : 0x1 ''
      +0x020 PoolType  : 0 ( NonPagedPool )
      +0x024 DefaultPagedPoolCharge : 0
      +0x028 DefaultNonPagedPoolCharge : 0xe8
      +0x02c DumpProcedure : (null)
      +0x030 OpenProcedure : (null)
      +0x034 CloseProcedure : (null)
      +0x038 DeleteProcedure : 0x80577d44
      +0x03c ParseProcedure : 0x80576964
      +0x040 SecurityProcedure : 0x80577f2a
      +0x044 QueryNameProcedure : (null)
      +0x048 OkayToCloseProcedure : (null)
```

С позиции руткита, существует два способа перехвата этих методов:

- Непосредственная подмена указателя на функцию-обработчик метода.
- Подмена указателя на тип в заголовке уже существующего объекта.

Для автоматизации выявления подобных перехватов удобнее всего воспользоваться драйвером, который выводит в отладочный вывод список методов для всех типов объектов в системе (см. приложение: drivers/ob_check_functions).

Второй способ перехвата методов - подмена указателя на тип в заголовке уже существующего объекта - более сложен в детектировании. Он ориентирован, в первую очередь, на перехват методов для одного, уже существующего, конкретного объекта и реализуется в несколько шагов:

1. Руткит получает указатель на нужный ему целевой обьект (например, на обьект типа «устройство» \Device\Harddisk0\DR0).
2. Руткит создаёт свою копию структуры OBJECT_TYPE для типа «устройство», в которой подменяет адрес нужного метода указателем на свой обработчик.
3. В заголовке целевого объекта, в поле Type (указатель на тип данного объекта) помещается адрес созданной в п.2 копии структуры.

Таким образом, легитимные структуры, описывающие типы объектов, остаются нетронутыми – указатели на методы модифицируются исключительно в их копиях. Для детектирования подобных перехватов будет достаточно перечислить все объекты, проверив в их заголовке значение поля Type. Получение указателей на все объекты можно реализовать посредством вызова функции ZwQuerySystemInformation с параметром SystemInformationClass, равным SystemObjectInformation.

Ядро Windows ведёт учет всех создаваемых объектов только тогда, когда в GlobalFlag установлен бит FLG_MAINTAIN_OBJECT_TYPELIST (0x4000). Если данный бит сброшен (а по умолчанию он сброшен почти всегда) - функция ZwQuerySystemInformation вернёт статус ошибки. Для установки этого бита удобнее всего использовать утилиту gflags.exe, входящую в состав Debugging Tools For Windows (см. рис. 11). После установки бита FLG_MAINTAIN_OBJECT_TYPELIST необходимо выполнить перезагрузку для того, что бы изменения вступили в силу. Значение GlobalFlags можно также устанавливать вручную: для этого необходимо модифицировать параметр GlobalFlag (типа REG_DWORD) в ключе реестра HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager.

| Имя метода | Когда вызывается |
|---|---|
| Open | При открытии дескриптора объекта. |
| Close | При закрытии дескриптора объекта. |
| Delete | Перед удалением объекта. |
| Parse | При поиске диспетчером объектов имени, существующего во вторичном пространстве имён объектов. |
| Security | При чтении или изменении параметров защиты объекта, существующего во вторичном пространстве имён объектов. |
| QueryName | При запросе имени объекта, существующего во вторичном пространстве имён объектов. |
| OkayToClose | Перед удалением объекта и вызовом метода Delete. В теле самого метода подсистема может разрешить или запретить процедуру удаления объекта. |

*Таблица 3. Методы объекта*

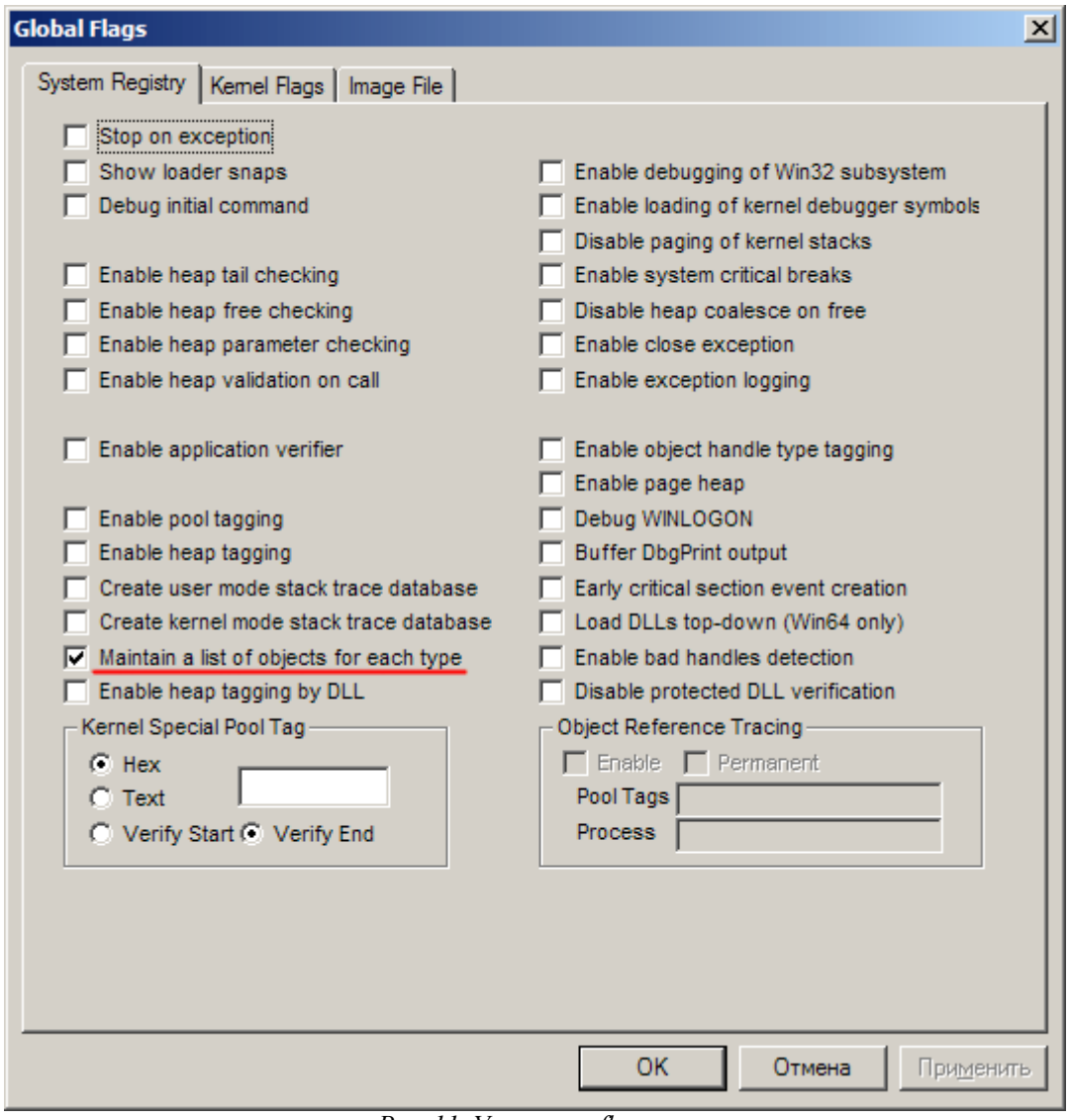


*Рис. 11. Утилита gflags.exe*

Код драйвера, который выявляет модификации поля Type в заголовке объекта и выводит информацию об найденных аномалиях в отладочный вывод, также находится в приложении к статье (drivers/ob_check_objects).

Теоретически, руткит может обойти такое детектирование либо путём перехвата функции ZwQuerySystemInformation, либо путем исключения своего объекта из двусвязного списка, находящегося в теле объекта типа «тип», которым данная функция манипулирует для получения нужной информации. По этим причинам действительно качественная антируткит-утилита должна уметь своими силами находить в неподкачиваемом пуле структуры, описывающие заголовки объектов.

## Преодолеваем практические сложности

Очень часто при борьбе с руткитами недостаточно только детектирования и снятия установленных перехватов: например, руткит может проверять состояние перехватов, и в случае их отсутствия – устанавливать заново. Кроме того, некоторые руткиты так же восстанавливают свои файлы, ключи и параметры системного реестра при их удалении антивирусной защитой.

В подавляющем большинстве случаев все манипуляции по восстановлению объектов или перехватов осуществляются в отдельном потоке, работающем в контексте процесса System. Такой поток создаётся вызовом функций PsCreateSystemThread или IoAllocateWorkItem / IoQueueWorkItem из драйвера руткита, а в коде этого потока, в бесконечном цикле с фиксированной задержкой между итерациями по таймеру выполняется все полезные манипуляции. Очевидно, что для препятствия этим манипуляциям будет достаточно завершить поток с помощью отладчика. Найти нужный поток руткита можно, перечислив все потоки процесса System, после чего, просматривая их стеки вызовов на предмет адресов, указывающих в драйвер руткита, идентифицировать нужные. (См. Листинг 10 – на примере руткита Rootkit.Win32.TDSS).

Красным шрифтом в листинге выделены принадлежащие драйверу руткита адреса в стеке вызовов, по присутствию которых была идентифицирована принадлежность потока. Зная идентификатор потока, его можно завершить с помощью достаточно мощного менеджера процессов (такого как Process Explorer). Однако из-за того, что в активной среде зараженной операционной системы руткит может этому препятствовать, найденные потоки желательно завершать исключительно с помощью отладчика.

К сожалению, штатные возможности WinDBG не позволяют завершать потоки и процессы при удалённой отладке. Самый простой способ завершения потока состоит из нескольких простых шагов:

1. Просматриваем стек вызовов нужного потока, начиная с самого верха, с целью найти первое значение адреса возврата (RetAddr), относящееся к драйверу руткита (на примере, показанном выше, это f7a094a5).
2. Записываем по найденному адресу вызов функции ядра PspExitThread.
3. Обнуляем флаг ActiveExWorker в структуре ETHREAD, которая описывает нужный поток руткита.
4. Возобновляем выполнение отлаживаемой операционной системы нажатием F5 в окне WinDbg).

После того, как поток возобновит своё выполнение, перейдя по хранящемуся в стеке адресу возврата, будет выполнен вызов функции PspExitThread, завершающей поток. Таким образом, мы заставим поток руткита завершить самого себя. Вот так выполнение этих шагов будет выглядеть в Command Window отладчика:

```
kd> x nt!PspExitThread
805c6bb0 nt!PspExitThread =
kd> eb f7a094a5 e8
kd> ed f7a094a5+1 805c6bb0-f7a094a5-5
kd> dt _ETHREAD 821cada8
ntdll!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1c0 CreateTime       : _LARGE_INTEGER 0x0
   +0x1c0 NestedFaultCount : 0y00
   +0x1c0 ApcNeeded        : 0y0
...
   +0x24c SameThreadPassiveFlags : 0
   +0x24c ActiveExWorker   : 0y0
   +0x24c ExWorkerCanWaitUser : 0y0
   +0x24c MemoryMaker      : 0y0
   +0x250 SameThreadApcFlags : 0
   +0x250 LpcReceivedMsgIdValid : 0y0
   +0x250 LpcExitThreadCalled : 0y0
   +0x250 AddressSpaceOwner : 0y0
   +0x254 ForwardClusterOnly : 0 ''
   +0x255 DisablePageFaultClustering : 0 ''
kd> eb 821cada8+0x24c 0
```

## Послесловие

Отладчик WinDBG является мощным инструментом диагностики, с помощью которого можно гарантированно выявлять любого рода аномалии в работающей операционной системе, так как при удалённой отладке её компоненты не участвуют в процессе съема и обработки информации и не могут ее исказить. В теории, руткит может активно препятствовать работе не только защитных средств, но и отладчика. Однако, в этом случае факт компрометации системы будет слишком очевиден, что в случае, к примеру, с целевыми руткитами, совершенно недопустимо. На данный момент, самые высокотехнологичные вредоносные программы снабжены функциями детектирования удалённого отладчика, которое легко обходится в силу самой своей специфики. Весьма существенным минусом WinDBG является маломощность его встроенного скриптового языка; данный недостаток можно исправить посредством интеграции в него интерпретатора python-а или perl-а как дополнительного модуля.

## Список литературы

[1] Руководство по настройке канала связи между виртуальной машиной и отладчиком
http://silverstr.ufies.org/lotr0
[2] Offline документация к отладчику windbg (команды as и ad)
[3] Таблица номеров системных вызовов для разных версии Windows
http://www.metasploit.com/users/opcode/syscalls.html
[4] MSDN: FltXxx (Minifilter Driver) Routines
http://msdn.microsoft.com/en-us/library/aa488414.aspx

## Приложение

Исходный код примеров и скрипты WinDbg:
http://www.nobunkum.ru/files/materials.rar

## Листинг 1. Код функции PsSetCreateThreadNotifyRoutine

```
nt! PsSetCreateThreadNotifyRoutine:
805c47a4 8bff            mov     edi,edi
805c47a6 55              push    ebp
805c47a7 8bec            mov     ebp,esp
805c47a9 53              push    ebx
805c47aa 57              push    edi
805c47ab 33ff            xor     edi,edi
; NULL
805c47ad 57              push    edi
; Указатель на нотификатор
805c47ae ff7508          push    dword ptr [ebp+8]
; Создаём и инициализируем EX_CALLBACK_ROUTINE_BLOCK
805c47b1 e87ad30300      call    nt!ExAllocateCallBack
805c47b6 8bd8            mov     ebx,eax
805c47b8 3bdf            cmp     ebx,edi
805c47ba 7507            jne     805c47c3
; Выход с кодом STATUS_INSUFFICIENT_RESOURCES в случае ошибки
805c47bc b89a0000c0      mov     eax,0C000009Ah
805c47c1 eb2a            jmp     805c47ed
805c47c3 56              push    esi
; указатель на список нотификаторов
805c47c4 bea0935580      mov     esi,
offset nt!PspCreateThreadNotifyRoutine (805593a0)
805c47c9 6a00            push    0
805c47cb 53              push    ebx
805c47cc 56              push    esi
; добавляем новый нотификатор в список
805c47cd e88ed30300      call    nt!ExCompareExchangeCallBack
805c47d2 84c0            test    al,al
805c47d4 751d            jne     805c47f3
805c47d6 83c704          add     edi,4
805c47d9 83c604          add     esi,4
; 20h / 4 = 8 = PSP_MAX_CREATE_THREAD_NOTIFY
; максимальное количество нотификаторов в списке
805c47dc 83ff20          cmp     edi,20h
805c47df 72e8            jb      805c47c9
805c47e1 53              push    ebx
805c47e2 e88f320200      call    nt!SeFreePrivileges
805c47e7 b89a0000c0      mov     eax,0C000009Ah
805c47ec 5e              pop     esi
805c47ed 5f              pop     edi
805c47ee 5b              pop     ebx
805c47ef 5d              pop     ebp
805c47f0 c20400          ret     4
```

## Листинг 2. Код функции PsSetLoadImageNotifyRoutine

```
nt!PsSetLoadImageNotifyRoutine:
805c4a54 8bff            mov     edi,edi
805c4a56 55              push    ebp
805c4a57 8bec            mov     ebp,esp
805c4a59 53              push    ebx
805c4a5a 57              push    edi
805c4a5b 33ff            xor     edi,edi
; NULL
805c4a5d 57              push    edi
; Указатель на нотификатор
805c4a5e ff7508          push    dword ptr [ebp+8]
; Создаём и инициализируем EX_CALLBACK_ROUTINE_BLOCK
805c4a61 e8cad00300      call    nt!ExAllocateCallBack
805c4a66 8bd8            mov     ebx,eax
805c4a68 3bdf            cmp     ebx,edi
805c4a6a 7507            jne     805c4a73
; Выход с кодом STATUS_INSUFFICIENT_RESOURCES в случае ошибки
805c4a6c b89a0000c0      mov     eax,0C000009Ah
805c4a71 eb2a            jmp     805c4a9d
805c4a73 56              push    esi
; указатель на список нотификаторов
805c4a74 be80935580      mov     esi,
offset nt!PspLoadImageNotifyRoutine (80559380)
805c4a79 6a00            push    0
805c4a7b 53              push    ebx
805c4a7c 56              push    esi
; добавляем новый нотификатор в список
805c4a7d e8ded00300      call    nt!ExCompareExchangeCallBack
805c4a82 84c0            test    al,al
805c4a84 751d            jne     805c4aa3
805c4a86 83c704          add     edi,4
805c4a89 83c604          add     esi,4
; 20h / 4 = 8 = PSP_MAX_LOAD_IMAGE_NOTIFY
; максимальное количество нотификаторов в списке
805c4a8c 83ff20          cmp     edi,20h
805c4a8f 72e8            jb      805c4a79
805c4a91 53              push    ebx
805c4a92 e8df2f0200      call    nt!SeFreePrivileges (805e7a76)
805c4a97 b89a0000c0      mov     eax,0C000009Ah
805c4a9c 5e              pop     esi
805c4a9d 5f              pop     edi
805c4a9e 5b              pop     ebx
805c4a9f 5d              pop     ebp
805c4aa0 c20400          ret     4
```

## Листинг 3. Код установки нотификатора

```
nt!CmRegisterCallback+0x14:
8055ee6b 68434d6362      push    62634D43h
8055ee70 33db            xor     ebx,ebx
8055ee72 6a38            push    38h
8055ee74 43              inc     ebx
8055ee75 53              push    ebx
; выделение памяти под контекст нотификатора
; указатель на контекст хранится в параметре Context, структуры
; EX_CALLBACK_ROUTINE_BLOCK
8055ee76 e80552feff      call    nt!ExAllocatePoolWithTag
8055ee7b 8bf0            mov     esi,eax
8055ee7d 33ff            xor     edi,edi
8055ee7f 3bf7            cmp     esi,edi
8055ee81 0f848a000000    je      8055ef11
; Context
8055ee87 56              push    esi
; Указатель на функцию-нотификатор
8055ee88 ff7508          push    dword ptr [ebp+8]
; Создаём и инициализируем EX_CALLBACK_ROUTINE_BLOCK
8055ee8b e8a02c0a00      call    nt!ExAllocateCallBack
8055ee90 3bc7            cmp     eax,edi
8055ee92 8945fc          mov     dword ptr [ebp-4],eax
8055ee95 7509            jne     8055eea0
nt!CmRegisterCallback+0x3e:
8055eea0 56              push    esi
; Генерируем идентификатор, который будет возвращён в
; параметре Cookie
8055eea1 e8927bf9ff      call    nt!KeQuerySystemTime
; инициализируем Context
8055eea6 8b0e            mov     ecx,dword ptr [esi]
8055eea8 8b4510          mov     eax,dword ptr [ebp+10h]
8055eeab 8908            mov     dword ptr [eax],ecx
8055eead 8b4e04          mov     ecx,dword ptr [esi+4]
8055eeb0 894804          mov     dword ptr [eax+4],ecx
8055eeb3 8d4608          lea     eax,[esi+8]
8055eeb6 894004          mov     dword ptr [eax+4],eax
8055eeb9 8900            mov     dword ptr [eax],eax
8055eebb 8d4624          lea     eax,[esi+24h]
8055eebe 895e10          mov     dword ptr [esi+10h],ebx
8055eebe 895e10          mov     dword ptr [esi+10h],ebx
8055eec1 885e1c          mov     byte ptr [esi+1Ch],bl
8055eec4 897e14          mov     dword ptr [esi+14h],edi
8055eec7 897e18          mov     dword ptr [esi+18h],edi
8055eeca c6461e04        mov     byte ptr [esi+1Eh],4
8055eece 897e20          mov     dword ptr [esi+20h],edi
8055eed1 894004          mov     dword ptr [eax+4],eax
8055eed4 8900            mov     dword ptr [eax],eax
8055eed6 8b450c          mov     eax,dword ptr [ebp+0Ch]
8055eed9 894630          mov     dword ptr [esi+30h],eax
; указатель на список нотификаторов
8055eedc bb20005580      mov     ebx,offset nt!CmpCallBackVector
; добавляем новый нотификатор в список
8055eee9 e8722c0a00      call    nt!ExCompareExchangeCallBack
```

## Листинг 4. Скрипт WinDbg для вывода списка нотификаторов

```
.echo Create Process Notifiers:
$$ нотификаторы на создание процессов
r $t0 = nt!PspCreateProcessNotifyRoutine
$$ перебираем EX_CALLBACK элементы массива
.for (r $t2 = 0; @$t2 < 8; r $t2 = @$t2 + 1)
{
    $$ отбрасываем младшие 3 бита,
    $$ которые хранят количество ссылок на указатель
    r $t3 = @@c++((*(long *)(@$t0 + @$t2 * 4)) & 0xfffffff8);

    .if (@$t3)
    {
        $$ если указатель не равен нулю,
        $$ выводим поле Function из EX_CALLBACK_ROUTINE_BLOCK
        dps @@c++((long *)(@$t3 + 4)) L 1
    }
}

.echo Create Thread Notifiers:
$$ нотификаторы на создание потоков
r $t0 = nt!PspCreateThreadNotifyRoutine
.for (r $t2 = 0; @$t2 < 8; r $t2 = @$t2 + 1)
{
    r $t3 = @@c++((*(long *)(@$t0 + @$t2 * 4)) & 0xfffffff8);

    .if (@$t3)
    {
        dps @@c++((long *)(@$t3 + 4)) L 1
    }
}

.echo Load Image Notifiers:
$$ нотификаторы на загрузку исполняемых образов
r $t0 = nt!PspLoadImageNotifyRoutine
.for (r $t2 = 0; @$t2 < 8; r $t2 = @$t2 + 1)
{
    r $t3 = @@c++((*(long *)(@$t0 + @$t2 * 4)) & 0xfffffff8);

    .if (@$t3)
    {
        dps @@c++((long *)(@$t3 + 4)) L 1
    }
}

.echo Registry Callbacks:
$$ нотификаторы на события системного реестра
r $t0 = nt!CmpCallBackVector
.for (r $t2 = 0; @$t2 < 0x64; r $t2 = @$t2 + 1)
{
    r $t3 = @@c++((*(long *)(@$t0 + @$t2 * 4)) & 0xfffffff8);

    .if (@$t3)
    {
        dps @@c++((long *)(@$t3 + 4)) L 1
    }
}
```

### Листинг 5. Вызов функции ExReferenceCallBackBlock перед вызовом нотификатора

```
Nt!PspCreateThread+0x3b1:
; циклический перебор списка нотификаторов
805c4f41 c745ace0935580  mov     dword ptr [ebp-54h],
        offset nt!PspCreateProcessNotifyRoutine (805593e0)
805c4f48 c7458408000000  mov     dword ptr [ebp-7Ch],8
805c4f4f ff75ac          push    dword ptr [ebp-54h]
; получаем EX_CALLBACK_ROUTINE_BLOCK для этого нотификатора
805c4f52 e8ddcc0300      call    nt!ExReferenceCallBackBlock
805c4f57 8bf8            mov     edi,eax
805c4f59 85ff            test    edi,edi
805c4f5b 741f            je      805c4f7c
805c4f5d 57              push    edi
805c4f5e e8bff0fcff      call    nt!IopGetRelationsTaggedCount
; Create = TRUE
805c4f63 6a01            push    1
; ProcessId
805c4f65 ffb384000000    push    dword ptr [ebx+84h]
; ParentId
805c4f6b ffb34c010000    push    dword ptr [ebx+14Ch]
; вызов нотификатора
805c4f71 ffd0            call    eax
805c4f73 57              push    edi
805c4f74 ff75ac          push    dword ptr [ebp-54h]
; декремент количества ссылок на EX_CALLBACK_ROUTINE_BLOCK
805c4f77 e8e4cd0300      call    nt!ExDereferenceCallBackBlock
805c4f7c 8345ac04        add     dword ptr [ebp-54h],4
805c4f80 ff4d84          dec     dword ptr [ebp-7Ch]
; переход к следующему элементу списка
805c4f83 75ca            jne     805c4f4f
```

## Листинг 6. Скрипт WinDbg для поиска модифицированных значений KTHREAD::ServiceTable

```
$$ указатель на список процессов
r $t0 = nt!PsActiveProcessHead
$$ перечисляем активные процессы
.for (r $t1 = poi(@$t0);
      (@$t1 != 0) & (@$t1 != @$t0);
      r $t1 = poi(@$t1))
{
    r? $t2 = #CONTAINING_RECORD(@$t1, nt!_EPROCESS,
        ActiveProcessLinks)
    as /x ${/v:$Procc} @$t2
    $$  Get image name into $ImageName.
    as /ma ${/v:$ImageName} @@c++(&@$t2->ImageFileName[0])

    .block
    {
        .echo ${$Procc}: ${$ImageName}
        .echo Active threads:
    }

    $$ указатель на список активных потоков этого процесса
    r? $t3 = (nt!_LIST_ENTRY *)&@$t2->ThreadListHead;

    $$ перечисляем потоки
    .for (r $t4 = poi(@$t3);
          (@$t4 != 0) & (@$t4 != @$t3);
          r $t4 = poi(@$t4))
    {
        r? $t5 = #CONTAINING_RECORD(@$t4, nt!_ETHREAD,
            ThreadListEntry)

        r? $t5 = (nt!_KTHREAD *)@$t5
        $$ получаем значение KTHREAD::ServiceTable
        r $t6 = @@c++(@$t5->ServiceTable)
        r $t7 = nt!KeServiceDescriptorTable
        r $t8 = nt!KeServiceDescriptorTableShadow

        as /x ${/v:$Kthread} @$t5
        as /x ${/v:$SDT} @$t6

        .block
        {
            .echo ${$Kthread} ServiceTable = ${$SDT}
        }

        $$ проверяем указатель
        .if ((@$t6 != @$t7) & (@$t6 != @$t8))
        {
            .block
            {
                .echo !!! Changed value of KTHREAD::ServiceTable
            }
        }
        ad ${/v:$Kthread}
        ad ${/v:$SDT}
    }
    ad ${/v:$ImageName}
    ad ${/v:$Procc}
}
```

## Листинг 7. Вывод таблицы IRP обработчиков в WinDbg

```
kd> !drvobj \FileSystem\Ntfs 2
Driver object (8192aa18) is for:
 \FileSystem\Ntfs
DriverEntry:   f96fa204       Ntfs!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      f9f2ad2c asfxz+0xd2c
[01] IRP_MJ_CREATE_NAMED_PIPE           804f320e nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       f969a320 Ntfs!NtfsFsdClose
[03] IRP_MJ_READ                        f9677ee4 Ntfs!NtfsFsdRead
[04] IRP_MJ_WRITE                       f9676bca Ntfs!NtfsFsdWrite
[05] IRP_MJ_QUERY_INFORMATION           f969b4d1 Ntfs!NtfsFsdDispatchWait
[06] IRP_MJ_SET_INFORMATION             f9678a58 Ntfs!NtfsFsdSetInformation
[07] IRP_MJ_QUERY_EA                    f969b4d1 Ntfs!NtfsFsdDispatchWait
[08] IRP_MJ_SET_EA                      f969b4d1 Ntfs!NtfsFsdDispatchWait
[09] IRP_MJ_FLUSH_BUFFERS               f96a0a68 Ntfs!NtfsFsdFlushBuffers
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    f969b61c Ntfs!NtfsFsdDispatch
[0b] IRP_MJ_SET_VOLUME_INFORMATION      f969b61c Ntfs!NtfsFsdDispatch
[0c] IRP_MJ_DIRECTORY_CONTROL           f9f2adc4 asfxz+0xdc4
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         f96a26d5 Ntfs!NtfsFsdFileSystemControl
[0e] IRP_MJ_DEVICE_CONTROL              f969b61c Ntfs!NtfsFsdDispatch
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     804f320e nt!IopInvalidDeviceRequest
[10] IRP_MJ_SHUTDOWN                    f9689621 Ntfs!NtfsFsdShutdown
[11] IRP_MJ_LOCK_CONTROL                f96eeb11 Ntfs!NtfsFsdLockControl
[12] IRP_MJ_CLEANUP                     f969acee Ntfs!NtfsFsdCleanup
[13] IRP_MJ_CREATE_MAILSLOT             804f320e nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              f969b61c Ntfs!NtfsFsdDispatch
[15] IRP_MJ_SET_SECURITY                f969b61c Ntfs!NtfsFsdDispatch
[16] IRP_MJ_POWER                       804f320e nt!IopInvalidDeviceRequest
[17] IRP_MJ_SYSTEM_CONTROL              804f320e nt!IopInvalidDeviceRequest
[18] IRP_MJ_DEVICE_CHANGE               804f320e nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 f969b4d1 Ntfs!NtfsFsdDispatchWait
[1a] IRP_MJ_SET_QUOTA                   f969b4d1 Ntfs!NtfsFsdDispatchWait
[1b] IRP_MJ_PNP                         f96b9f3f Ntfs!NtfsFsdPnp
...
```

## Листинг 8. Процедура заполнения массива IRP обработчиков (ntfs.sys)

```
push    offset aNtfs_0  ; "\\Ntfs" INIT:000952CF
lea     eax, [ebp+DeviceName]
push    eax             ; DestinationString
call    ds:__imp__RtlInitUnicodeString@8 ; RtlInitUnicodeString(x,x)
lea     eax, [ebp+DeviceObject]
push    eax             ; DeviceObject
push    ebx             ; Exclusive
push    ebx             ; DeviceCharacteristics
push    8               ; DeviceType
lea     eax, [ebp+DeviceName]
push    eax             ; DeviceName
push    ebx             ; DeviceExtensionSize
push    esi             ; DriverObject
call    ds:__imp__IoCreateDevice@28 ; IoCreateDevice(x,x,x,x,x,x,x)
cmp     eax, ebx
jl      loc_958F9
mov     dword ptr [esi+7Ch], offset _NtfsFsdLockControl@8
mov     dword ptr [esi+68h], offset _NtfsFsdDirectoryControl@8
mov     dword ptr [esi+50h], offset _NtfsFsdSetInformation@8
mov     dword ptr [esi+38h], offset _NtfsFsdCreate@8
mov     dword ptr [esi+40h], offset _NtfsFsdClose@8
mov     dword ptr [esi+44h], offset _NtfsFsdRead@8
mov     dword ptr [esi+48h], offset _NtfsFsdWrite@8
mov     dword ptr [esi+5Ch], offset _NtfsFsdFlushBuffers@8
mov     dword ptr [esi+6Ch], offset _NtfsFsdFileSystemControl@8
mov     dword ptr [esi+80h], offset _NtfsFsdCleanup@8
mov     dword ptr [esi+78h], offset _NtfsFsdShutdown@8
mov     dword ptr [esi+0A4h], offset _NtfsFsdPnp@8
mov     dword ptr [esi+28h], offset _NtfsFastIoDispatch
mov     eax, offset _NtfsFsdDispatchWait@8
mov     [esi+4Ch], eax
mov     [esi+0A0h], eax
mov     [esi+9Ch], eax
mov     [esi+58h], eax
mov     [esi+54h], eax
mov     eax, offset _NtfsFsdDispatch@8
mov     [esi+64h], eax
mov     [esi+60h], eax
mov     [esi+8Ch], eax
mov     [esi+88h], eax
mov     [esi+70h], eax
```

### Листинг 9. Список типов объектов

```
kd> !object \ObjectTypes
Object: e10012a0  Type: (821f1118) Directory
    ObjectHeader: e1001288 (old version)
    HandleCount: 0  PointerCount: 25
    Directory Object: e10005e0  Name: ObjectTypes

    Hash Address  Type          Name
    ---- -------  ----          ----
     00  821f1118 Type          Directory
     01  821cca98 Type          Thread
         821c95c0 Type          Mutant
     03  82129040 Type          FilterCommunicationPort
     05  821b8958 Type          Controller
     07  821f12e8 Type          Type
         821c9960 Type          Event
         821c8ca0 Type          Profile
     09  821cc040 Type          SymbolicLink
         821c9790 Type          EventPair
         821c8510 Type          Section
     10  821c8730 Type          Desktop
     11  821c8e70 Type          Timer
     12  821c8900 Type          WindowStation
         821ef040 Type          File
     16  821b85b8 Type          Driver
     18  821c8ad0 Type          KeyedEvent
         821b27a0 Type          WmiGuid
     19  821cce70 Type          Token
         821b8788 Type          Device
     20  821cb040 Type          DebugObject
     21  821b83e8 Type          IoCompletion
     22  821ccc68 Type          Process
     24  821b8b28 Type          Adapter
     26  821c57c8 Type          Key
     28  821cc8c8 Type          Job
     31  821f0748 Type          Port
         821f0578 Type          WaitablePort
     32  821c93f0 Type          Callback
     33  821283e0 Type          FilterConnectionPort
     34  821c8040 Type          Semaphore
```

## Листинг 10. Поток руткита Rootkit.Win32.TDSS

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 821cc5f0  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00701000  ObjectTable: e1000c78  HandleCount: 333.
    Image: System
...
kd> !process 821cc5f0
PROCESS 821cc5f0  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00701000  ObjectTable: e1000c78  HandleCount: 333.
    Image: System
    VadRoot 821c81e8 Vads 4 Clone 0 Private 3. Modified 184. Locked 0.
    DeviceMap e1006008
    Token                             e10016e0
    ElapsedTime                       00:01:35.906
    UserTime                          00:00:00.000
    KernelTime                        00:00:08.375
    QuotaPoolUsage[PagedPool]         0
    QuotaPoolUsage[NonPagedPool]      0
    Working Set Sizes (now,min,max)  (61, 0, 345) (244KB, 0KB, 1380KB)
    PeakWorkingSetSize                510
    VirtualSize                       1 Mb
    PeakVirtualSize                   2 Mb
    PageFaultCount                    3221
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      7
...
        THREAD 821cada8  Cid 0004.0024  Teb: 00000000 Win32Thread: 00000000
                    WAIT: (UserRequest) KernelMode Alertable
            f9ccbad0  NotificationEvent
        IRP List:
            81d6a540: (0006,0094) Flags: 00000000  Mdl: 81d4e8a0
        Not impersonating
        DeviceMap                 e1006008
        Owning Process            821cc5f0       Image:         System
        Wait Start TickCount      4732           Ticks: 1406 (0:00:00:21.968)
        Context Switch Count      21
        UserTime                  00:00:00.000
        KernelTime                00:00:00.000
        Start Address nt!ExpWorkerThread (0x80533cd0)
        Stack Init f9ccd000 Current f9ccba68 Base f9ccd000 Limit f9cca000 Call 0
        Priority 14 BasePriority 12 PriorityDecrement 0 DecrementCount 0
        ChildEBP RetAddr
        f9ccba80 8050017a nt!KiSwapContext+0x2e (FPO: [Uses EBP] [0,0,4])
        f9ccba8c 804f99be nt!KiSwapThread+0x46 (FPO: [0,0,0])
        f9ccbab4 f7a094a5 nt!KeWaitForSingleObject+0x1c2 (FPO: [Non-Fpo])
WARNING: Frame IP not in any known module. Following frames may be wrong.
        f9ccbae4 f7a098e0 0xf7a094a5
        f9ccbb08 f7a0b502 0xf7a098e0
        f9cccd74 80533dd0 0xf7a0b502
        f9cccdac 805c4a28 nt!ExpWorkerThread+0x100 (FPO: [Non-Fpo])
        f9cccddc 80540fa2 nt!PspSystemThreadStartup+0x34 (FPO: [Non-Fpo])
        00000000 00000000 nt!KiThreadStartup+0x16
```