ВЗЛОМ

Энциклопедия антиотладочных приемов 7. Скрытая установка SEH-обработчиков

Крис Касперски , 01.12.2008 Комментарии 311 Добавить в закладки

Содержание статьи

Продолжая окучивать плодородную тему структурных исключений, поговорим о методах скрытой установки SEH-обработчиков, используемых для затруднения дизассемблирования/отладки подопытного кода, а также обсудим возможные контрмеры антиантиотладочных способов.

ПОСТАНОВКА ПРОБЛЕМЫ

Структурные исключения представляют собой мощное антиотладочное средство, в чем мы уже убедились на примере предыдущих выпусков. Там же мы познакомились и с техникой исследования программ, играющихся исключениями, работу с которыми достаточно трудно замаскировать.

Всякий раз, когда в тексте программы встречается конструкция `MOV FS:[0], xxx`, хакер сразу встает торчком — раз это `FS:[0]`, значит, программа устанавливает собственный SEH-обработчик и, судя по всему, сейчас будет бросать исключения. Теоретически возможно засунуть `MOV FS:[0], xxx` в самомодифицирующийся код, убрав его из дизассемблерных листингов, однако против аппаратной точки останова по записи на `MOV FS: [0], xxx` ничто не спасет. В момент установки нового SEH-обработчика отладчик тут же «всплывет», демаскируя защитный механизм. А `SetUnhandledExceptionFilter` вообще представляет собой API-функцию, экспортируемую KERNEL32.DLL, которую легко обнаружить любым API-шпионом, даже без анализа всего дизассемблерного кода!

Задача: установить собственный обработчик структурных исключений, но так, чтобы это как можно меньше бросалось в глаза и не палилось тривиальной установкой точек останова. Решением мы сейчас, собственно, и займемся, предложив широкий ассортимент антиотладочных трюков, один интереснее другого.

ПЕРЕЗАПИСЬ СУЩЕСТВУЮЩЕГО ОБРАБОТЧИКА

Вместо того чтобы устанавливать новый обработчик структурных исключений, некоторые (и достаточно многие) защиты предпочитают модифицировать указатель на существующий. Даже если приложение и не устанавливает никаких SEH-обработчиков, система все равно впихивает ему SEH-обработчик по умолчанию, смотрящий куда-то в дебри KERNEL32.DLL. На этом, кстати говоря, основан популярный примем поиска базового адреса загрузки KERNEL32.DLL, в котором нуждается shell-код, а также программы, написанные без использования таблицы импорта (из-за ошибки в системном загрузчике они работают только на XP и более поздних версиях).

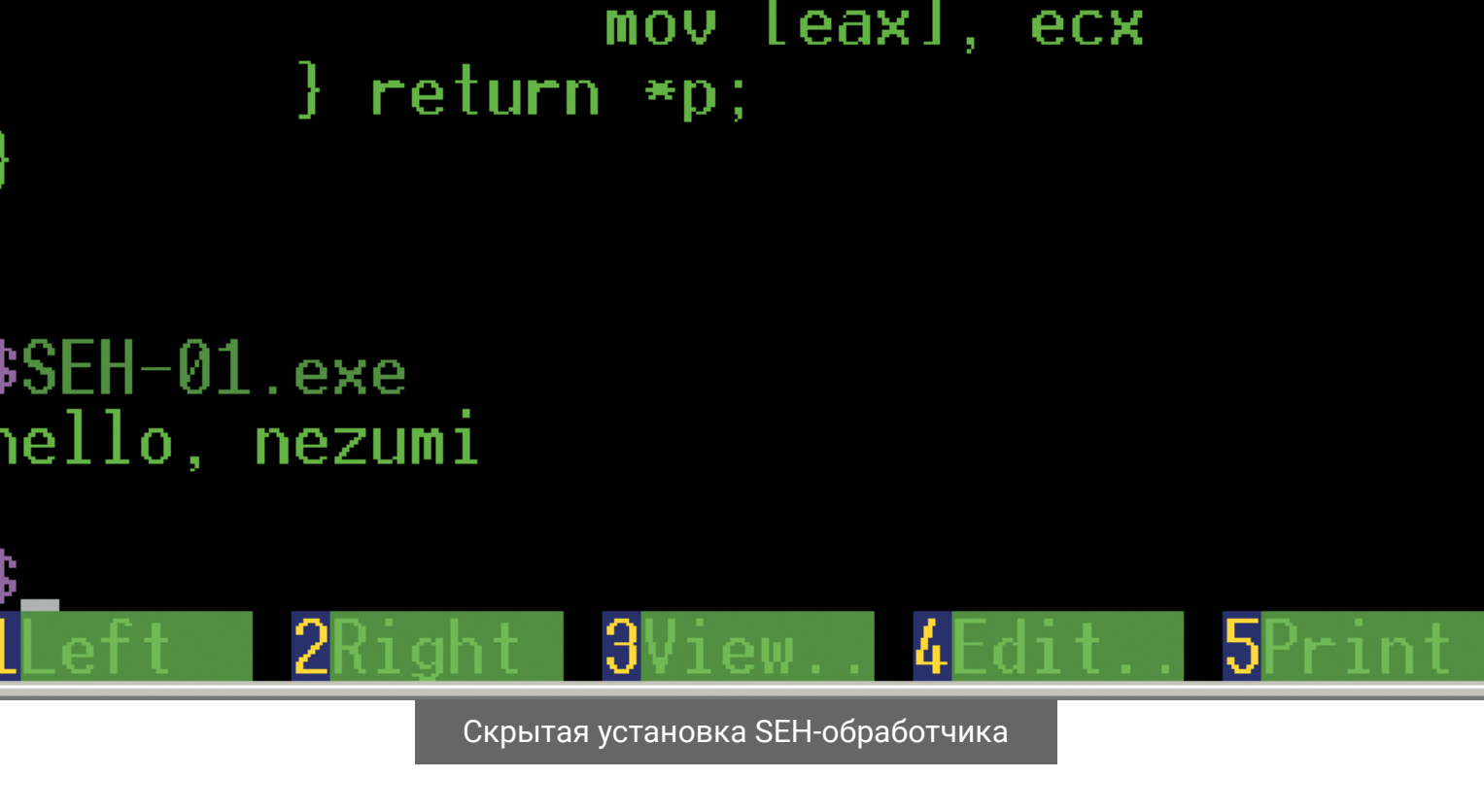
Обработчик по умолчанию не делает ничего полезного, и потому без него можно обойтись, «позаимствовав» указатель — на время или навсегда. Конкретный пример реализации приведен ниже.

Установка своего SEH-обработчика без перезаписи ячейки FS:[0]

```
souriz()
{
    printf("hello, nezumi\n"); ExitProcess(0);
}


main()
{
    int *p=0;
    __asm{
        mov eax, fs:[0]
        lea ecx, souriz
        add eax, 4
        mov [eax], ecx
    }
    return *p;
}
```

Внешне код очень похож на классический способ установки SEH-обработчика, но, присмотревшись внимательнее, мы видим, что в нашем примере модифицируется отнюдь не ячейка `FS:[0]`, а то, на что она указывает. Точка останова по записи на `FS:[0]` уже не сработает, однако сегментный регистр FS режет глаз, да и бряк на `FS:[0]` по доступу продолжает работать, а потому для эффективного противодействия хакеру требуются дополнительные уровни маскировки. Ну и чего мы сидим? Вперед!



Скрытая установка SEH-обработчика

ПРЯЧЕМ FS

Ослепить дизассемблеры совсем не трудно. Перезаписать указатель на системный SEH-обработчик можно и без явного использования сегментного регистра FS. Самое простое, что можно сделать, — скопировать его в любой другой сегментный регистр (например, GS). С точки зрения процессора, регистры FS и GS совершенно равноправны. Главное, чтобы в регистре содержался «правильный» селектор, а его название — уже дело десятое. Создавать новые селекторы мы не можем (точнее, можем, но это тема отдельного разговора), а вот загружать существующие — почему бы и нет?

Усиленный фрагмент защиты приведен ниже.

Прячем регистр FS от любопытных глаз

```
__asm{
        mov ax, fs
        mov gs, ax
    }
    …
    …
    …
    __asm{
        mov eax, gs:[0]
        lea ecx, souriz
        add eax, 4
        mov [eax], ecx
    }
```

Небольшое пояснение. Поскольку ни один известный мне компилятор не использует регистр GS для своих целей, его можно инициализировать в одной процедуре, а использовать — в другой. Единственное условие — обе процедуры должны принадлежать одному потоку, поскольку каждый поток обладает собственным регистровым контекстом.

Начинающих хакеров обращение к регистру GS дробит на части, сваливая в вертикальный штопор. Короче, это как обухом по голове. «Ольга» (в отличие от «Айса») не показывает значений сегментных регистров, чем серьезно осложняет ситуацию.

Опытных реверсеров таким макаром не проведешь, но никаких гарантий, что GS в данный момент содержит именно FS, а не, например, DS, у нас нет. А потому статический анализ становится неоднозначным и требует реконструкции последовательности вызываемых функций. Причем обращения к FS в явном виде может и не быть — его значение легко прочитать API-функцией `GetThreadContext`, на которую, конечно, нетрудно поставить точку останова, но точки останова — это уже динамический, а не статический анализ!

Самое интересное, что блок окружения потока, засунутый в селектор (который хранится в сегментном регистре FS), отображается на плоское адресное пространство, а значит, доступен для чтения и через остальные селекторы. Например, через сегментный регистр DS. На W2K блок окружения первичного потока начинается с адреса 7FFDB000h (7FFDE000h на XP), поэтому вместо `FS:[0]` допустимо использовать конструкцию `DS:[7FFDB000h]`. Чтобы избежать краха, надо отталкиваться от того факта, что в настоящем блоке окружения потока по смещению 30h байт от его начала расположен указатель на блок окружения процесса, лежащий на 1000h байт ниже. Благодаря чему мы можем найти указатель на SEH-обработчик даже на неизвестной операционной системе!

Конечно, реализация алгоритма существенно усложняется, но это даже хорошо, поскольку чем больше строк кода — тем дольше их будет анализировать хакер, особенно если эти строки бессмысленны сами по себе.

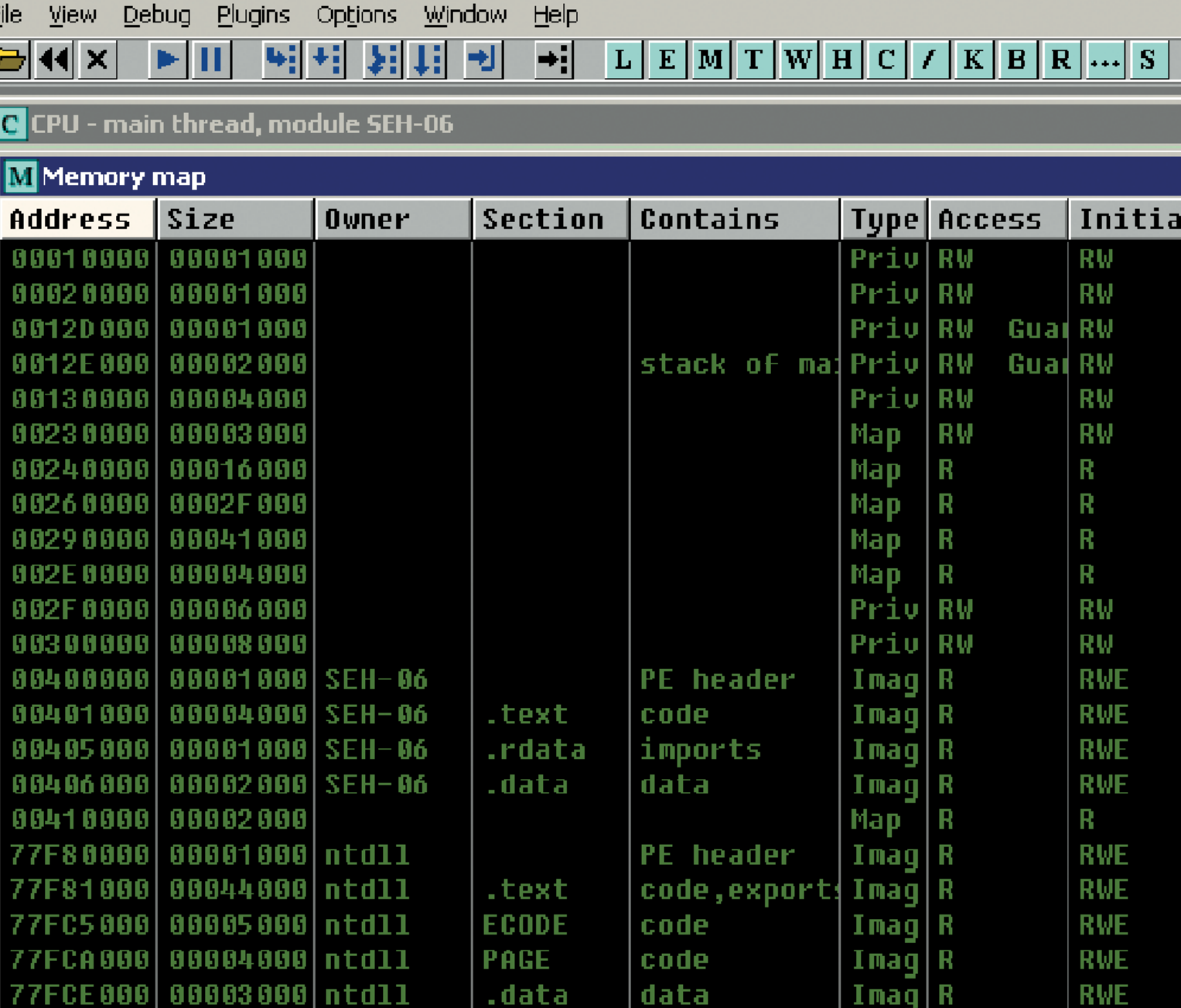
Поиск блока окружения потока в стеке

```
int a; int *p=0;
    unsigned char *pp = (unsigned char*) 0x7FFE0000;

    for(a = 0; a < 6; a++)
    {
        pp -= 0x1000;
        if (IsBadReadPtr(pp, 4)) continue;
        if (IsBadReadPtr((pp + 0x30), 4)) continue;
        if ( *((size_t*)(pp + 0x30)) == ((size_t) pp + 0x1000) )
        {
            *(size_t*) (*((size_t*)pp) + 4) = (size_t*) souriz;
            return *p;
        }
    } printf("not found\n");
```

Во-первых, мы обошлись без ассемблерных вставок, реализовав алгоритм на чистом C (с тем же успехом можно использовать паскаль). Во-вторых, вместо характерного FS в программе появилась куча констант, смысл которых понятен только посвященным, да и то не без пристального анализа, сопровождаемого глубокой медитацией. В-третьих, факт передачи управления на функцию `souriz` по `return *p (где p == 0)` совершенно не очевиден. К тому же сам указатель на `souriz` можно зашифровать, помешав дизассемблерам реконструировать перекрестные ссылки. Как это сделать на C (без ассемблерных вставок), описывалось в 1Eh выпуске сишных трюков.

Существуют и другие способы поиска указателя на блок окружения потока. Рассмотрим только два самых популярных. Просматривая карту памяти (а просмотреть ее можно с помощью API-вызова VirtualQuery), даже удав заметит, что блоки окружения процесса и потока лежат в своих собственных секциях памяти с атрибутами Private и правами на чтение/запись. Размер каждого блока равен 1000h, плюс ко всему указатель на блок окружения процесса расположен по смещению 30h байт от блока окружения потока. То есть если `*((size_t*)(block_1+30h)) == block_2`, то block_1 — блок окружения потока, а block_2 — блок окружения процесса и `MOV EAX, FS:[0]` равносильно `MOV EAX, block_1/MOV EAX, [EAX]`. Вывод: без FS можно по-любому обойтись.



Блок окружения потока на карте памяти процесса

Указатель на блок окружения потока также находится в стеке потока, куда его кладет операционная система. В W2K/XP это третье двойное слово от вершины. И хотя в последующих версиях его местоположение может измениться, вирусов это обстоятельство, похоже, никак не заботит, и они используют его сплошь и рядом.

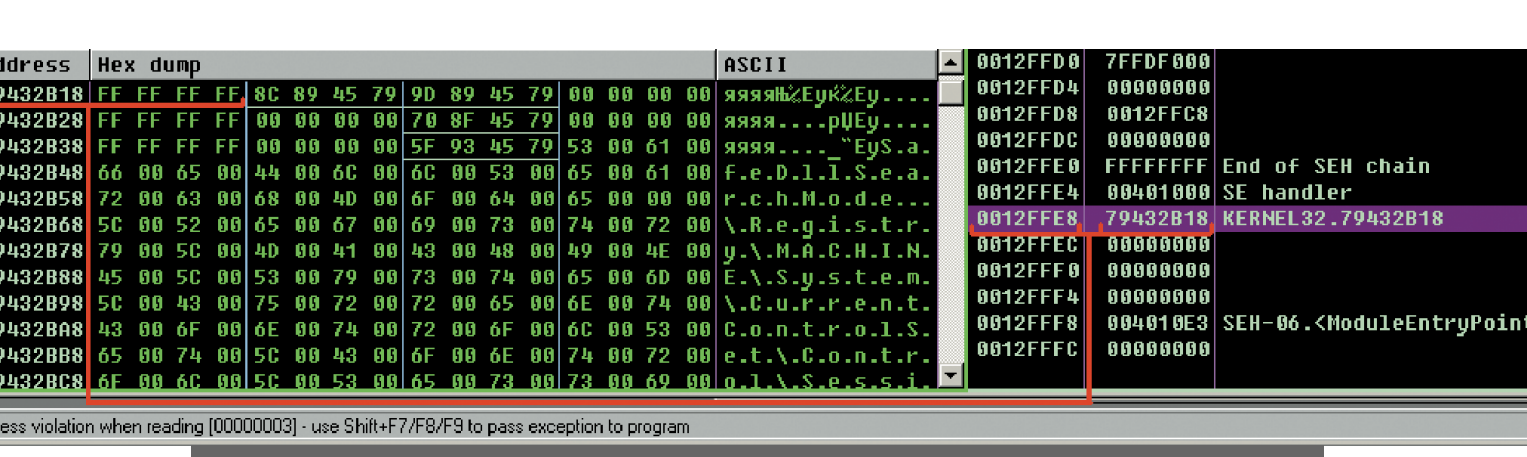
И что в итоге? Мы рассмотрели множество приемов скрытого обращения к ячейке FS:0, однако все они действуют только против дизассемблеров, а отладчики просто ставят сюда точку останова по доступу, и все обращения к FS:0 немедленно палятся. Независимо от того, какой адрес используется — смещение 0 по селектору FS или же смещение 7FFDB000h по селектору DS.

Непорядок! Хорошая защита должна справляться не только с дизассемблерами, но и с отладчиками.

КРАЖА ЧУЖИХ ОБРАБОТЧИКОВ

Системный обработчик структурных исключений расположен на дне стека потока — и обращаться к блоку окружения для его поисков совсем не обязательно, поскольку местоположение обработчика непостоянно и зависит от версии операционной системы. С учетом этого мы должны выработать эвристический алгоритм поиска.

Системный обработчик, назначаемый по умолчанию, есть не что иное, как функция `__except_handler3`, расположенная в недрах KERNEL32.DLL и не экспортируемая наружу, но присутствующая в отладочных символах. Которые теоретически можно в любой момент скачать с серверов Microsoft, но практически такое решение будет слишком громоздким, неудобным, ненадежным, да и довольно «прозрачным» для хакера.



Указатель на системный SEH-обработчик лежит на дне стека потока

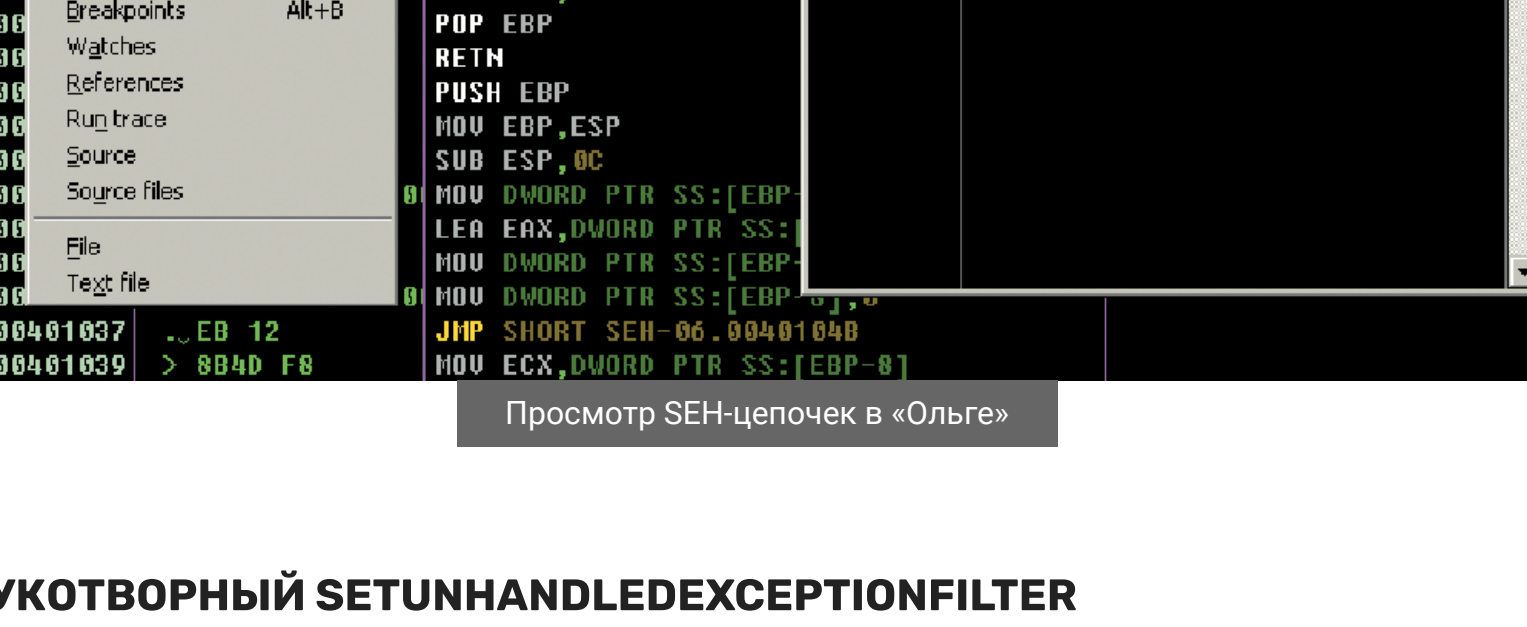
Хорошо, будем отталкиваться от того, что `__except_handler3` смотрит в KERNEL32.DLL и что перед ним всегда расположено двойное слово FFFFFFFFh, а после него — указатель на секцию данных KERNEL32.DLL, опять-таки содержащий в себе двойное слово FFFFFFFFh. Последнее обстоятельство системно-зависимо, но справедливо как для W2K, так и для XP, а потому его можно использовать без особых опасений.

Практический пример приведен ниже:

Прямой поиск указателя на SEH-обработчик в стеке

```
for (a=0;a<69;a++,pp++)
{
    if (IsBadReadPtr((pp+2), 4)) break;
    if (*pp == 0xFFFFFFFF)
    {
        if (IsBadReadPtr(*(pp + 2), 4)) continue;
        if (*((unsigned int*)*(pp + 2)) == 0xFFFFFFFF)
        {
            *(pp + 1) = (unsigned int*) souriz;
            return *p;
        }
    }
} printf("not found\n");
```

Точка останова на FS:0 на этот раз идет лесом и не срабатывает, поскольку обращения к этой ячейке памяти уже не происходит. К тому же разобраться, что именно ищет программа в стеке, можно после серии экспериментов (ну или чтения этой статьи). Способов поиска системного обработчика исключений намного больше одного. Это существенно усложняет задачу хакера, и универсальных «отмычек» тут нет, что в плане защиты очень даже хорошо. Однако просмотр цепочки обработчиков структурных исключений (в «Ольге» через меню View → SEH Chain) немедленно разоблачает хакнутый обработчик, на который несложно установить точку останова на исполнение со всеми вытекающими отсюда последствиями.



Просмотр SEH-цепочек в «Ольге»

РУКОТВОРНЫЙ SETUNHANDLEDEXCEPTIONFILTER

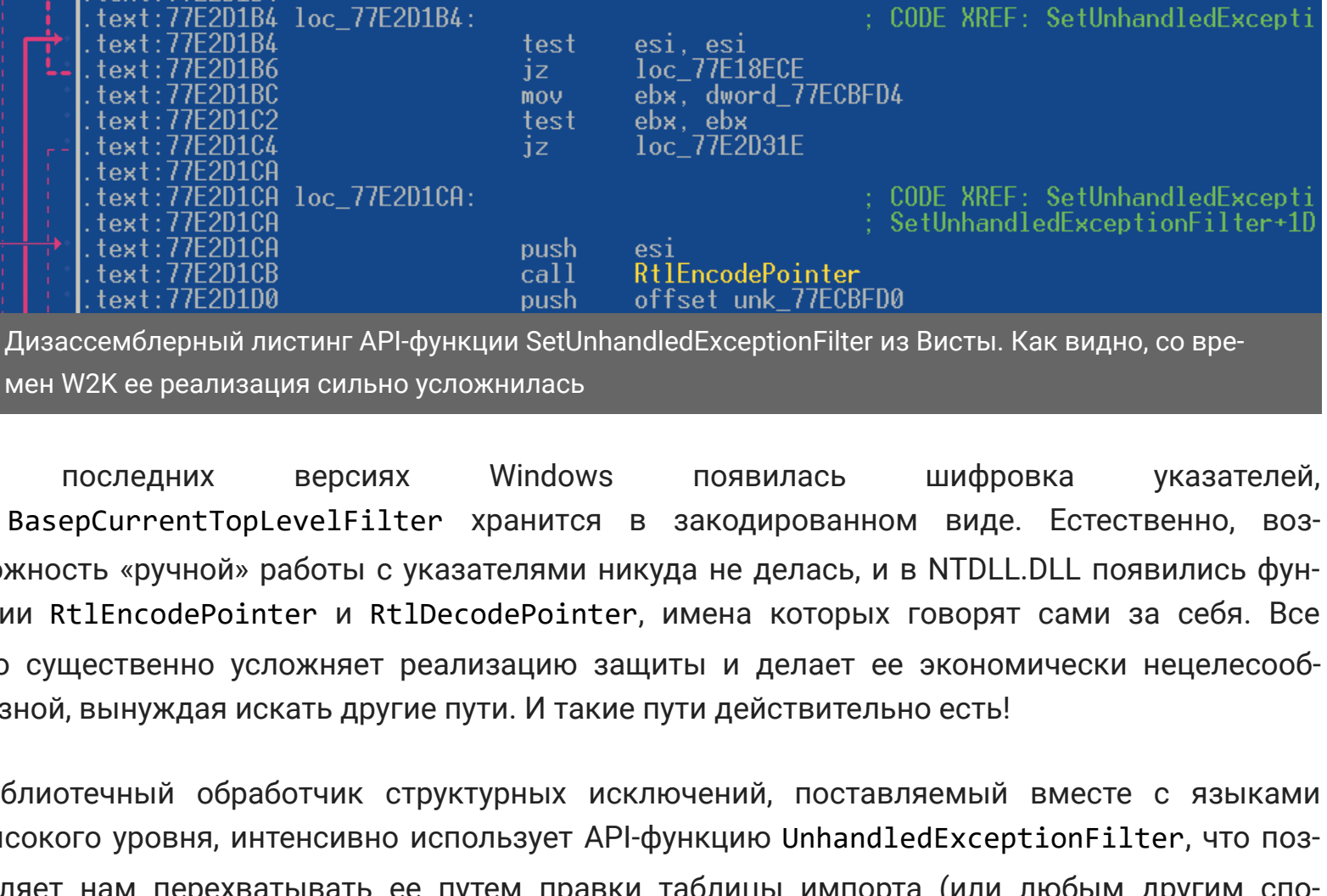
API-функция `SetUnhandledExceptionFilter`, как уже отмечалось в предыдущих выпусках, сама по себе представляет проблему для отладчиков, поскольку установленный ею фильтр исключений верхнего уровня при запуске программы под отладчиком не выполняется и приходится использовать разнообразные плагины для «Ольги», чтобы заставить систему считать, что никакого отладчика здесь нет. Или же, как вариант, насильственно включать фильтр верхнего уровня в цепочку обработчиков структурных исключений.

Самый большой недостаток функции `SetUnhandledExceptionFilter` в том, что ее вызов очень трудно замаскировать, но трудно еще не значит невозможно. К тому же реализация функции проста как движок от «запора». Фактически она всего лишь устанавливает глобальную переменную `BasepCurrentTopLevelFilter`, хранящуюся внутри KERNEL32.DLL и используемую только функцией `UnhandledExceptionFilter`.

Дизассемблерный листинг API-функции SetUnhandledExceptionFilter из W2K

```
.text:7945BC45  _SetUnhandledExceptionFilter@4 proc near
.text:7945BC45
.text:7945BC45  lpTopLevelExceptionFilter= dword ptr  4
.text:7945BC45
.text:7945BC45 8B 4C 24 04      mov ecx, [esp+lpTopLevelExceptionFilter]
.text:7945BC49 A1 F0 A1 48 79       mov eax, _BasepCurrentTopLevelFilter
.text:7945BC4E 89 0D F0 A1 48 79    mov _BasepCurrentTopLevelFilter, ecx
.text:7945BC54 C2 04 00      retn    4
.text:7945BC54  _SetUnhandledExceptionFilter@4 endp
```

Все, что нам нужно, — это найти `BasepCurrentTopLevelFilter` внутри `SetUnhandledExceptionFilter` (или `UnhandledExceptionFilter`) и прописать сюда указатель на свой собственный обработчик исключений. К сожалению, это не избавляет нас от необходимости импортировать `SetUnhandledExceptionFilter/UnhandledExceptionFilter` или получить эффективный адрес путем ручного разбора таблицы экспорта KERNEL32.DLL. Да, конечно, ручной разбор с использованием хеш-сум вместо имен API-функций до некоторой степени скрывает наши намерения от хакера. Увы, нет ничего тайного, что не стало бы явным. Даже если выбранный хеш-алгоритм математически необратим, запустив программу под отладчиком, всегда можно установить, какой именно API-функции какой хеш соответствует.



Дизассемблерный листинг API-функции SetUnhandledExceptionFilter из Висты. Как видно, со времен W2K ее реализация сильно усложнилась

В последних версиях Windows появилась шифровка указателей, и `BasepCurrentTopLevelFilter` хранится в закодированном виде. Естественно, возможность «ручной» работы с указателями никуда не делась, и в NTDLL.DLL появились функции `RtlEncodePointer` и `RtlDecodePointer`, имена которых говорят сами за себя. Все это существенно усложняет реализацию защиты и делает ее экономически нецелесообразной, вынуждая искать другие пути. И такие пути действительно есть!

Библиотечный обработчик структурных исключений, поставляемый вместе с языками высокого уровня, интенсивно использует API-функцию `UnhandledExceptionFilter`, что позволяет нам перехватывать ее путем правки таблицы импорта (или любым другим способом). Конечно, модификация импорта — грязный трюк, привлекающий к себе внимание, поэтому лучше хакнуть непосредственно саму библиотечную функцию обработки исключений. В случае MS VC эта функция носит имя `__XcptFilter`. Первые байты трогать нежелательно — иначе IDA-Pro ее не распознает; впрочем, байт байту рознь. IDA-Pro пропускает относительные вызовы, поскольку они непостоянны и подвержены сезонным вариациям.

Нам нужно найти `CALL func` и заменить `func` адресом нашей функции `my_func`, выполняющей некоторые действия и при необходимости возвращающей управление оригинальной `func`. Анализ кода `__XcptFilter` обнаруживает вызов `_xcptlookup`, осуществляемый в основном блоке кода, то есть не «шунтируемый» никакими ветвлениями, что очень хорошо:

Дизассемблерный фрагмент библиотечной функции __XcptFilter

```
.text:00401C9A __XcptFilter proc near
.text:00401C9A
.text:00401C9A arg_0        = dword ptr  8
.text:00401C9A ExceptionInfo    = dword ptr  0Ch
.text:00401C9A
.text:00401C9A          push    ebp
.text:00401C9B          mov ebp, esp
.text:00401C9D          push    ebx
.text:00401C9E          push    [ebp+arg_0]
.text:00401CA1          call    _xcptlookup ; _xcptlookup ? my_invisible_seh
.text:00401CA6          test    eax, eax
```

Обнаружить наш обработчик исключений практически невозможно. Он отсутствует в SEH-цепочке (точнее, присутствует, но прячется внутри обработчика, устанавливаемого RTL языка высокого уровня), и «Ольга» в упор его не видит. Конечно, при пошаговой трассировке хакерский обработчик будет выявлен — вот только трассировать мегабайты системного и библиотечного кода никто не будет. Дизассемблирование также не покажет ничего подозрительного, поскольку IDA-Pro не проверяет целостность библиотечных функций и никто [illegible] ма живучим в плане взлома.