

энциклопедия антиотладочных приемов — скрытая установка SEH-обработчиков выпуск #07h

крис касперски, aka мыщѣх, a.k.a nezumi, a.k.a souriz, a.k.a elraton, no-email

продолжая окучивать плодородную почву темы структурных исключений, поговорим о методах скрытой установки SEH-обработчиков, используемых для затруднения дизассемблирования/отладки подопытного кода, а так же обсудим возможные контрмеры анти-анти-отладочных способов.

введение — 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, играющих исключениями, работу с которыми достаточно трудно замаскировать.

Всякий раз когда в тексте программы встречается конструкция `MOV FS:[0], xxx`, хакер сразу встает торчком, издавая звук выпускаемого воздуха — 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 саомодифицирующийся код, убрав его из дизассемблерных листингов, однако, против аппаратной точки останова по записи на `MOV FS:[0], xxx` ничего не спасет и 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, чем 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 (между прочим, достаточно многие) защиты предпочитают модифицировать указатель на уже существующий. Даже если приложение и не устанавливает никаких SEH-обработчиков, система все равно вписывает ему SEH-обработчик по умолчанию, смотрящий куда-то в дебри `KERNEL32.DLL`, на чем, кстати говоря, основан популярный прием поиска базового адреса загрузки `KERNEL32.DLL`, в котором нуждается `shell`-код, а так же 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:

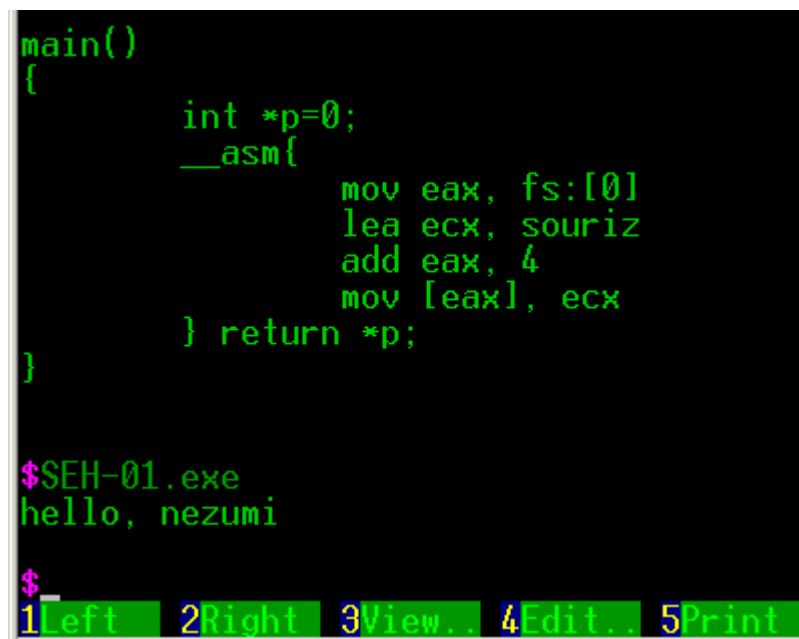
```
souriz()
{
    printf("hello, nezumi\n"); ExitProcess(0);
}

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

Листинг 1 установка своего SEH-обработчика без перезаписи ячейки FS:[0]

Внешне этот код очень похож на классический способ установки SEH-обработчика, однако, присмотревшись повнимательнее, мы видим, что в нашем примере модифицируются 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!



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

$SEH-01.exe
hello, nezumi

$
1Left 2Right 3View.. 4Edit.. 5Print
```

Рисунок 1 скрытая установка SEH-обработчика

прячем FS

Ослепить дизассемблеры совсем нетрудно. Перезаписать указатель на системный SEH-обработчик можно и без явного использования сегментного регистра FS. Самое простое, что только 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 — дело десятое. Создавать новые селекторы мы не можем (точнее, можем, но это тема отдельного разговора), но загружать уже существующие — почему бы и нет?!

Усиленный фрагмент защиты приведен ниже:

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

Листинг 2 прячем регистр FS от любопытных глаз

Небольшое пояснение. Поскольку, ни один известный мне компилятор не использует регистр GS для своих целей, то 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. Или серпом по яйцам (ес-но, для тех, у кого они есть, а девушки среди хакеров нет-нет, да встречаются). Кстати, на счет девушек. Ольга (в отличие от Айс) не показывает значений сегментных регистров, чем серьезно осложняет ситуацию.

Опытных реверсеров таким макаром уже не проведешь, однако, 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 прочесть API-функцией `GetThreadContext`, на которую, конечно, легко поставить точку останова, но точки останова это уже динамический, а не статический анализ!

Самое интересное — блок окружения потока, засунутый в селектор, хранящийся в сегментном регистре FS, отображается на плоское адресное пространство и потому доступен для чтения и через остальные селекторы, например, через сегментный регистр DS. На W2K блок окружения первичного потока начинается с адреса 7FFDB000h и 7FFDE000h на XP, поэтому (не без риска, конечно), вместо `FS:[0]` допустимо использовать конструкцию `DS:[7FFDB000h]`, а чтобы избежать краха, 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, тем более если эти строки бессмысленны сами по себе.

```
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");
```

Листинг 3 поиск блока окружения потока в стеке

Во-первых, мы обошлись без ассемблерных вставок, реализовав алгоритм на чистом Си (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_ неочевиден, к тому же, сам указатель на `souriz` можно зашифровать, помешав дизассемблерам реконструировать перекрестные ссылки. Как это сделать на Си (без ассемблерных вставок) описывалось в 1Еh выпуске сишных трюков.

Существуют и другие способы поиска указателя на блок окружения потока. Рассмотрим только два самых популярных из них. 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, причем 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]`, то есть без FS можно по любому обойтись.

OllyDbg - SEH-06.exe

File View Debug Plugins Options Window Help

LEMTWHC / KBR ... S

CPU - main thread, module SEH-06

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial
00010000	00001000				Priv	RW	RW
00020000	00001000				Priv	RW	RW
0012D000	00001000				Priv	RW	Guar
0012E000	00002000			stack of ma	Priv	RW	Guar
00130000	00004000				Priv	RW	RW
00230000	00003000				Map	RW	RW
00240000	00016000				Map	R	R
00260000	0002F000				Map	R	R
00290000	00041000				Map	R	R
002E0000	00004000				Map	R	R
002F0000	00006000				Priv	RW	RW
00300000	00008000				Priv	RW	RW
00400000	00001000	SEH-06		PE header	Imag	R	RWE
00401000	00004000	SEH-06	.text	code	Imag	R	RWE
00405000	00001000	SEH-06	.rdata	imports	Imag	R	RWE
00406000	00002000	SEH-06	.data	data	Imag	R	RWE
00410000	00002000				Map	R	R
77F80000	00001000	ntdll		PE header	Imag	R	RWE
77F81000	00044000	ntdll	.text	code, exports	Imag	R	RWE
77FC5000	00005000	ntdll	ECODE	code	Imag	R	RWE
77FCA000	00004000	ntdll	PAGE	code	Imag	R	RWE
77FCE000	00003000	ntdll	.data	data	Imag	R	RWE
77FD1000	00001000	ntdll	EDATA		Imag	R	RWE
77FD2000	00028000	ntdll	.rsrc	resources	Imag	R	RWE
77FFA000	00003000	ntdll	.reloc	relocations	Imag	R	RWE
79430000	00001000	KERNEL32		PE header	Imag	R	RWE
79431000	00059000	KERNEL32	.text	code, imports	Imag	R	RWE
7948A000	00004000	KERNEL32	.data	data	Imag	R	RWE
7948E000	00052000	KERNEL32	.rsrc	resources	Imag	R	RWE
794E0000	00004000	KERNEL32	.reloc	relocations	Imag	R	RWE
7F6F0000	00007000				Map	R E	R E
7FFB0000	00024000				Map	R	R
7FFDE000	00001000			data block	Priv	RWE	RWE
7FFDF000	00001000				Priv	RWE	RWE
7FFFE000	00001000				Priv	R	R

Рисунок 2 блок окружения потока на карте памяти процесса

Указатель на блок окружения потока так же 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 походе никак не заботит и они используют его сплошь и рядом.

И что в итоге? Мы рассмотрели множество приемов скрытного 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 справляется не только с дизассемблерами, но и с отладчиками!

кража чужих обработчиков

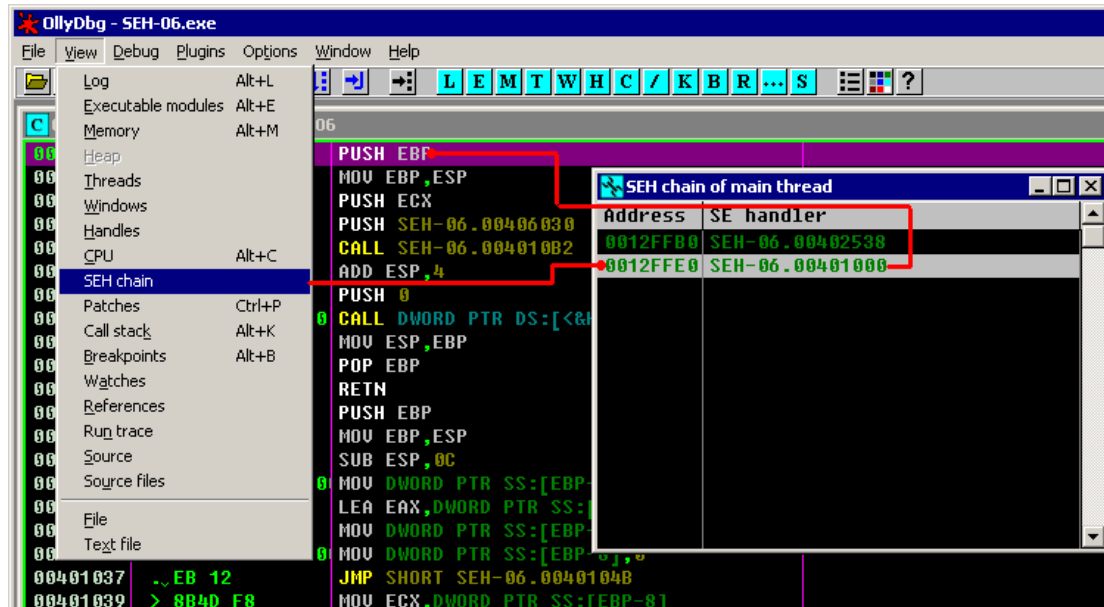


Рисунок 4 просмотр SEH-цепочек в Ольге

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.

```
.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
```

Листинг 5 дизассемблерный листинг API-функции SetUnhandledExceptionFilter из W2K

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 импортирования SetUnhandledExceptionFilter/UnhandledExceptionFilter или получения эффективного адреса путем ручного разбора таблицы экспорта KERNEL32.DLL. Да, конечно, ручной разбор с использованием хэш-сум вместо имен API-функций до некоторой степени скрывает наши намерения от хакера, однако, нет ничего тайного что бы ни стало явным. Даже если выбранный хэш-алгоритм математически необратим, запустив программу под отладчиком всегда можно установить какой именно API функции какой хэш соответствует.

```

.text:77E2D187 public SetUnhandledExceptionFilter
.text:77E2D187 SetUnhandledExceptionFilter proc near
.text:77E2D187
.text:77E2D187 var_220 = dword ptr -220h
.text:77E2D187 loTopLevelExceptionFilter = dword ptr 4
.text:77E2D187 arg_4 = dword ptr 8
.text:77E2D187
.text:77E2D187 ; FUNCTION CHUNK AT .text:77E18ECE SIZE 00000007 BYTES
.text:77E2D187 ; FUNCTION CHUNK AT .text:77E2D31E SIZE 00000048 BYTES
.text:77E2D187 ; FUNCTION CHUNK AT .text:77E574F0 SIZE 0000000E BYTES
.text:77E2D187 ; FUNCTION CHUNK AT .text:77E63669 SIZE 00000027 BYTES
.text:77E2D187
.text:77E2D187 mov edi, edi
.text:77E2D189 push ebp
.text:77E2D18A mov ebp, esp
.text:77E2D18C sub esp, 220h
.text:77E2D192 push ebx
.text:77E2D193 push esi
.text:77E2D194 mov esi, [ebp+arg_4]
.text:77E2D197 test esi, esi
.text:77E2D199 jz loc_77E18ECE
.text:77E2D19F lea eax, [ebp+var_220]
.text:77E2D1A5 push eax
.text:77E2D1A6 push esi
.text:77E2D1A7 call sub_77E2D248
.text:77E2D1AC test eax, eax
.text:77E2D1AE jz loc_77E574F0
.text:77E2D1B4 loc_77E2D1B4: ; CODE XREF: SetUnhandledExceptionFilter+10
.text:77E2D1B4 test esi, esi
.text:77E2D1B6 jz loc_77E18ECE
.text:77E2D1BC mov ebx, dword_77ECBFD4
.text:77E2D1C2 test ebx, ebx
.text:77E2D1C4 jz loc_77E2D31E
.text:77E2D1CA loc_77E2D1CA: ; CODE XREF: SetUnhandledExceptionFilter+1D
.text:77E2D1CA ; SetUnhandledExceptionFilter+1D
.text:77E2D1CA push esi
.text:77E2D1CB call RtlEncodePointer
.text:77E2D1D0 push offset unk_77ECBFD0

```

Рисунок 5 дизассемблерный листинг API-функции SetUnhandledExceptionFilter из Висты, как видно, со времен W2K ее реализация сильно усложнилась

К тому же, 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/RtlDecodePointer имена которых говорят сами за себя, однако, все это существенно усложняет реализацию защиты, что делает ее экономически нецелесообразной, вынуждая нас 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 хануть 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.

То есть, нам нужно найти CALL func и заменить func адресом нашей функции my_func, выполняющий некоторые действия и при необходимости возвращающую управление оригинальной func. Анализ кода __XcptFilter обнаруживает вызов _xcptlookup, осуществляемый в основном блоке кода, т.е. _не_ "шунтируемый" никакими ветвлениями, что очень хорошо:

```

.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

```

Листинг 6 дизассемблерный фрагмент библиотечной функции __XcptFilter

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 так же не покажет ничего подозрительного, поскольку IDA-Pro не проверяет целостность библиотечных функций, и никто из хакеров не тратит время на их анализ, а потому предложенный прием оказывается весьма живучим в плане взлома.