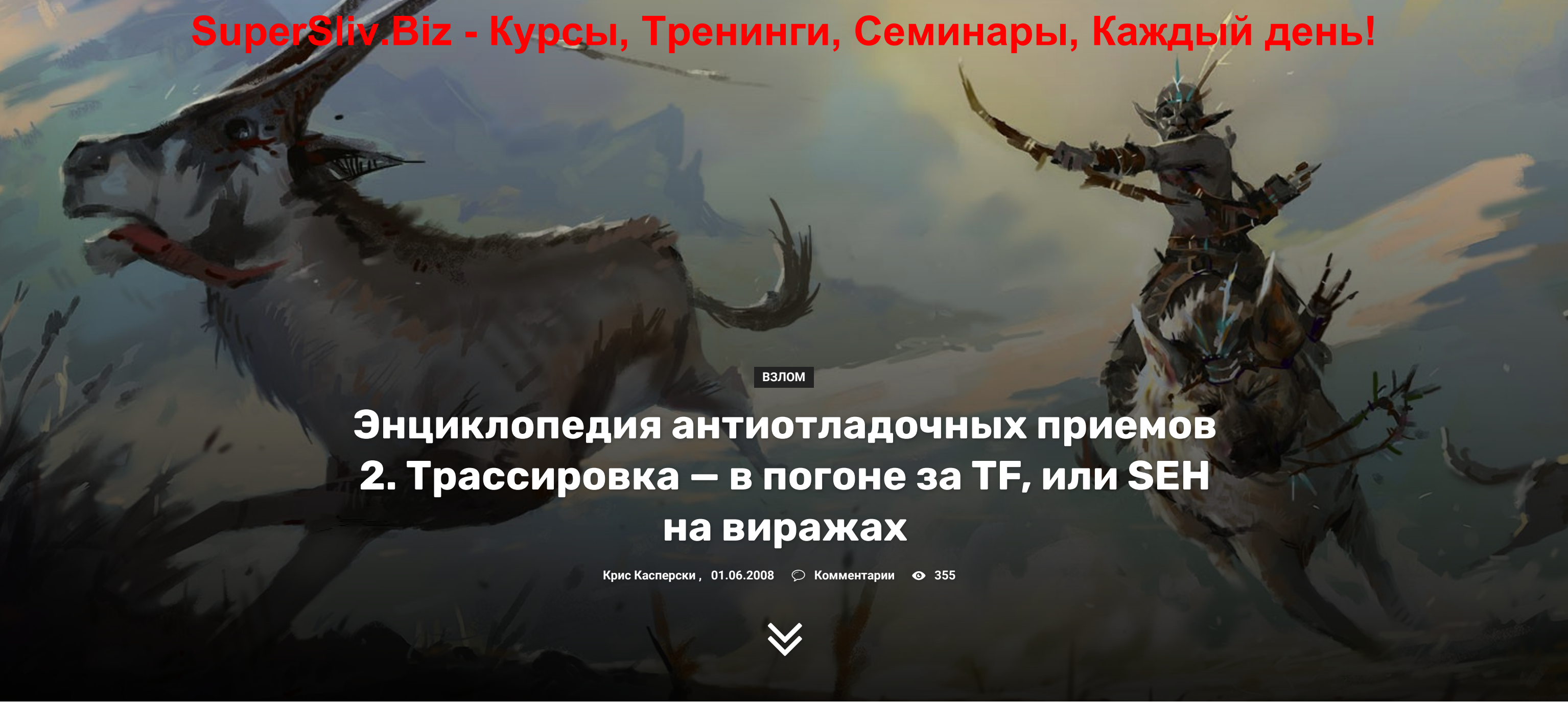


ВЗЛОМ

Энциклопедия антиотладочных приемов 2. Трассировка — в погоне за TF, или SEH на виражах

Крис Касперски , 01.06.2008 Комментарии 355

Содержание статьи

Охота на флаг трассировки подходит к концу, и дичь уже хрустит на зубах. Продолжив наши эксперименты с TF-битом, мы познакомимся со структурными и векторными исключениями, выводящими борьбу с отладчиками в вертикальную плоскость. Здесь не действуют привычные законы, и приходится долго и нудно ковыряться в недрах системы, чтобы угадать, куда будет передано управление и как усмирить разбушевавшуюся защиту.

Отладчики обоих уровней — как ядерного, так и прикладного — совершенно не приспособлены для исследования программ, интенсивно использующих структурные исключения (они же structured exceptions, более известные как SEH, где последняя буква досталась в наследство от слова handling — обработка). И хотя OllyDbg делает некоторые шаги в этом направлении, без написания собственных скриптов/макросов не обойтись. Генерация исключения «телепортирует» нас куда-то внутрь NTDLL.DLL, в толщу служебного кода, выполняющего поиск и передачу управления на SEH-обработчик, который нас интересует больше всего. Как в него попасть? Отладчик не дает ответа, а тупая трассировка требует немало времени.

Впрочем, SEH — это ерунда. Начиная с XP появилась поддержка обработки векторных исключений (VEH), усиленная в Server 2003 и, соответственно, в Висте / Server 2008. Отладчики об этом вообще не знают, открывая разработчикам защит огромные возможности для антиотладки и обламывая начинающих хакеров косяками. Я покажу, как побороть SEH/VEH-штучки в любом отладчике типа Syser, Soft-Ice или WinDbg. К сожалению, OllyDbg содержит грубую ошибку в «движке» и для отладки SEH/VEH-программ не подходит. Ну не то чтобы совсем не подходит, но повозиться придется (секс будет — и много).

SEH FUNDAMENTALS

Архитектура структурных исключений подробно описана в десятках книг и сотнях статей. Настолько подробно, что, читая их, можно уснуть. Поэтому краткое изложение основных концепций, выполненное в моем фирменном стиле, не помешает.

Исключение, сгенерированное процессором, тут же перехватывается ядром операционной системы, которое его долго и нудно мутузит, но в конце концов возвращает управление на прикладной уровень, вызывая функцию `NTDLL.DLL!KiUserCallbackDispatcher`. При пошаговой трассировке отладчики прикладного/ядерного уровня пропускают ядерный код, сразу же оказываясь в `NTDLL.DLL!KiUserCallbackDispatcher`. То есть при трассировке кода `XOR EAX,EAX/MOV EAX,[EAX]` следующей выполняемой командой оказывается первая инструкция функции `NTDLL.DLL!KiUserCallbackDispatcher`. Сюрприз, да?

При выполнении `KiUserCallbackDispatcher` извлекает указатель на цепочку обработчиков структурных исключений, хранящийся по адресу `FS:[00000000h]`, и вызывает первый обработчик через функцию ExecuteHandler, передавая ему специальные параметры.

В зависимости от значения, возвращенного обработчиком, функция `KiUserCallbackDispatcher` либо продолжает «раскручивать» список структурных исключений, либо останавливает «раскрутку», возвращая управление коду, породившему исключение. Исходя из типа исключения (trap или fault) управление передается либо машинной команде, сгенерировавшей исключение, либо следующей инструкции (подробнее об этом можно прочитать в мануалах Intel).

Список обработчиков структурных исключений представляет собой простой односвязный список.

Формат списка обработчиков структурных исключений

```
_EXCEPTION_REGISTRATION struc
    prev    dd  ?       ; // Предыдущий обработчик, -1 — конец списка
    handler dd  ?       ; // Указатель на SEH-обработчик
_EXCEPTION_REGISTRATION ends
```

Обработка структурных исключений имеет следующий прототип и возвращает одно из трех значений, описанных в MSDN: `EXCEPTION_CONTINUE_SEARCH`, `EXCEPTION_CONTINUE_EXECUTION` или `EXCEPTION_EXECUTE_HANDLER`.

Прототип обработчика структурных исключений

```
handler(PEXCEPTION_RECORD pExcptRec, PEXCEPTION_REGISTRATION pExcptReg,
    CONTEXT * pContext, PVOID pDispatcherContext,   FARPROC handler);
```

Обработчики структурных исключений практически полностью реентерабельны — обработчик также может генерировать исключения, корректно подхватываемые системой и начинающие раскрутку списка обработчиков с нуля. «Практически» — потому что, если исключение возникает при попытке вызова обработчика (например, из-за исчерпания стека), ядро просто молчаливо прибивает процесс. Но это уже дебри технических деталей, в которые мы пока не будем углубляться.

После установки своего собственного обработчика не забывай его снимать, иначе есть шанс получить весьма неожиданный результат. Причем система игнорирует попытку снять обработчик внутри самого обработчика, и это нужно делать только за пределами его тела.

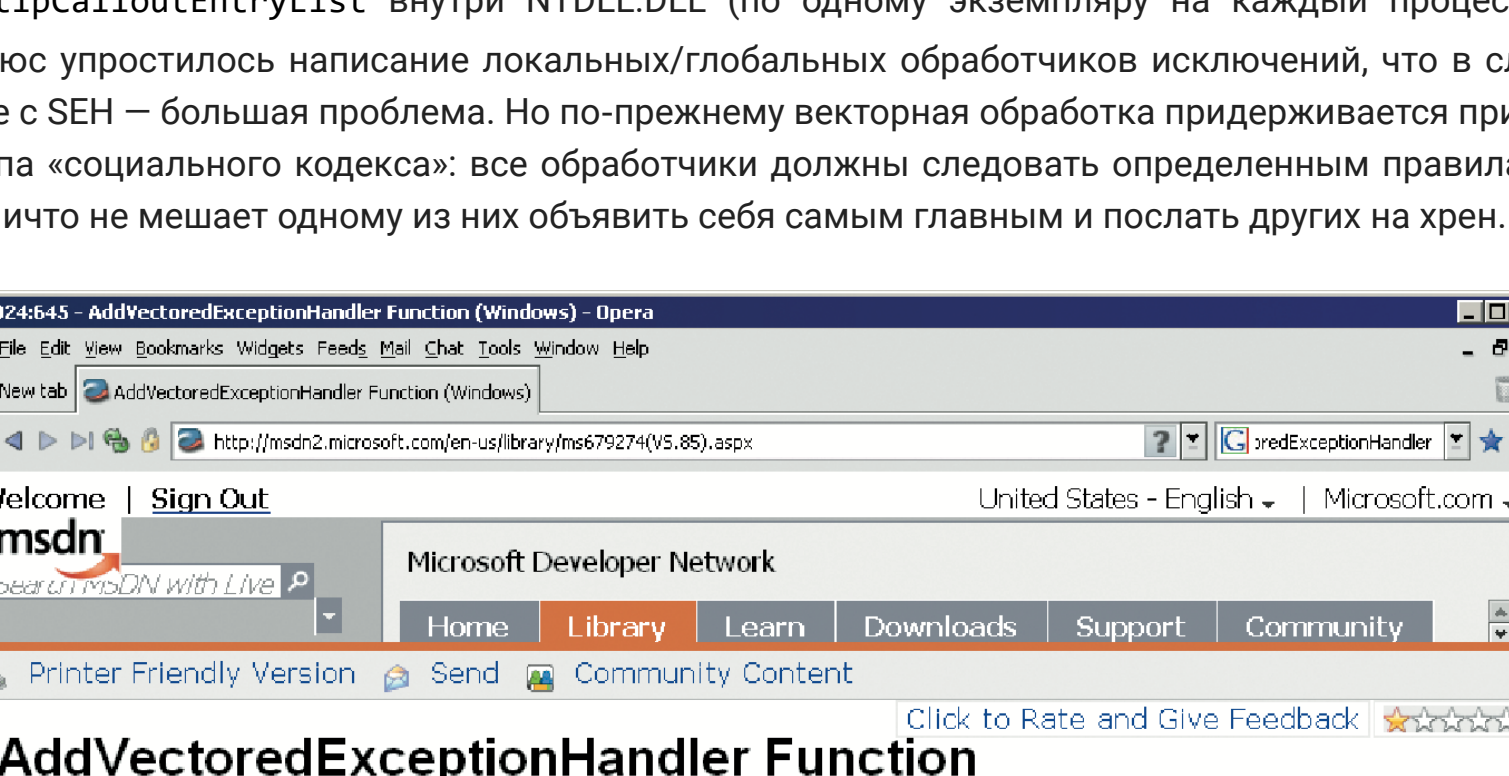
Вот абсолютный минимум знаний, который нам понадобится для брачных игр со структурными исключениями.

VEH FUNDAMENTALS

Начиная с XP появилась поддержка векторных исключений — разновидность SEH, однако реализованная независимо от нее и работающая параллельно. Другими словами, добавление нового векторного обработчика никак не затрагивает SEH-цепочку, и, соответственно, наоборот.

Механизм обработки векторных исключений работает по тому же принципу, что и SEH, вызывая уже знакомую нам функцию `NTDLL.DLL!KiUserCallbackDispatcher`. В свою очередь, она вызывает `NTDLL.DLL!RtlCallVectoredExceptionHandlers`, раскручивающую список векторных обработчиков с последующей передачей управления.

SEH и VEH концептуально очень схожи. Они предоставляют одинаковые возможности, и вся разница между ними в том, что вместо ручного манипулирования со списками обработчиков теперь у нас есть API-функции `AddVectoredExceptionHandler` и `RemoveVectoredExceptionHandler`, устанавливающие векторные обработчики и удаляющие их из списка, указатель на который хранится в неэкспортируемой переменной `_RtlpCalloutEntryList` внутри NTDLL.DLL (по одному экземпляру на каждый процесс). Плюс упростилось написание локальных/глобальных обработчиков исключений, что в случае с SEH — большая проблема. Но по-прежнему векторная обработка придерживается принципа «социального кодекса»: все обработчики должны следовать определенным правилам и ничто не мешает одному из них объявить себя самым главным и послать других на хрен.



Описание новых VEH-функций на MSDN

Поскольку 9x/W2K-системы еще достаточно широко распространены, пользоваться векторной обработкой без особой на то нужды могут только дураки. Во всяком случае, необходимо использовать динамическую загрузку векторных функций, экспортируемых библиотекой KERNEL32.DLL, и, если их там не окажется, либо выдать сообщение об ошибке, либо дезактивировать защитный модуль, работающий на базе VEH.

Теперь пара слов о новых API-функциях. `AddVectoredExceptionHandler` имеет следующий прототип и принимает два параметра, первый из которых обычно равен нулю, а второй представляет собой указатель на обработчик векторных исключений:

Прототип API-функции AddVectoredExceptionHandler, добавляющий новый векторный обработчик в список

```
PVOID WINAPI AddVectoredExceptionHandler(
    ULONG FirstHandler,
    PVECTORED_EXCEPTION_HANDLER VectoredHandler);
```

Функция `AddVectoredExceptionHandler` определена в файле winnt.h, поставляемом с новыми версиями SDK, да и то только в том случае, если в программе определен макрос `_WIN32_WINNT` со значением 0x0500 или большим. Если у нас нет свежего SDK, то определить прототип можно и самостоятельно, прямо по месту использования функции.

Прототип процедуры обработки векторных исключений

```
LONG CALLBACK VectoredHandler(PEXCEPTION_POINTERS ExceptionInfo);
```

Для удаления установленных векторных обработчиков из списка можно воспользоваться API-функцией `RemoveVectoredExceptionHandler`, где `Handler` — указатель на обработчик:

Прототип API-функции RemoveVectoredExceptionHandler, удаляющей векторный обработчик из списка

```
ULONG WINAPI RemoveVectoredExceptionHandler(PVOID Handler);
```

ЭКСПЕРИМЕНТ #4 — ЛОВЛЯ TF-БИТА НА SEH

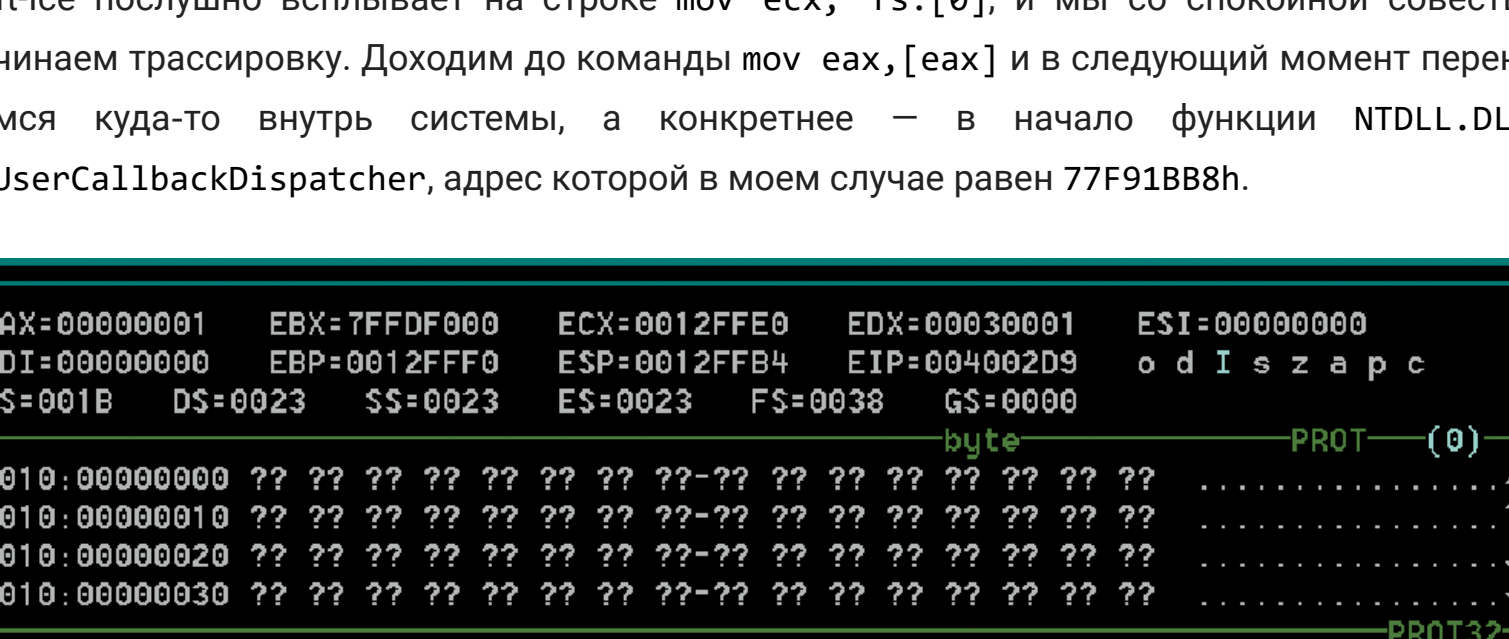
Продемонстрируем технику отладки программ, использующих структурные исключения, на примере crackme (его сорцы ищи среди приложенных к статье файлов). Он генерирует общую ошибку доступа к памяти путем обращения по нулевому указателю и проверяет значение флага трассировки в заранее установленном SEH-обработчике. На самом деле для запутывания хакера назначается сразу два обработчика: первый ничего не делает и тупо возвращает управление, а второй считывает регистровый контекст, извлекает оттуда содержимое флага трассировки и увеличивает значение EIP на два байта — длину инструкции `mov eax, [eax]`, вызывавшей исключение.

Для упрощения отладки из программы выкинули все лишнее (и стартовый код в том числе), поэтому для ее сборки применяется специальный командный файл следующего содержания:

TF-SEH.bat — сборка программы без стартового кода

```
cl /Ox /c TF-SEH.c
link TF-SEH.obj /ALIGN:16 /DRIVER /FIXED /ENTRY:nezumi
    /SUBSYSTEM:CONSOLE KERNEL32.LIB USER32.lib
```

Компилируем программу и загружаем ее в любой подходящий отладчик (например, Soft-Ice). Если загрузка обламывается (известный глюк Soft-Ice), раскомментируем строку с командой `int 03h`, пересоберем программу, напишем в Soft-Ice `i3here on` и запустим все по новой. Soft-Ice послушно всплывает на строке `mov ecx, fs:[0]`, и мы со спокойной совестью начинаем трассировку. Доходим до команды `mov eax,[eax]` и в следующий момент переносимся куда-то внутрь системы, а конкретнее — в начало функции `NTDLL.DLL!KiUserCallbackDispatcher`, адрес которой в моем случае равен 77F91BB8h.



Отладчик успешно обнаружен

Приехали! Дальше продолжать трассировку нет смысла, нужен способ быстро найти адрес структурного обработчика. Например, можно посмотреть, что находится в памяти по указателю `FS:[00000000h]`:

Определяем список адресов SEH-обработчиков, просматривая fs:0

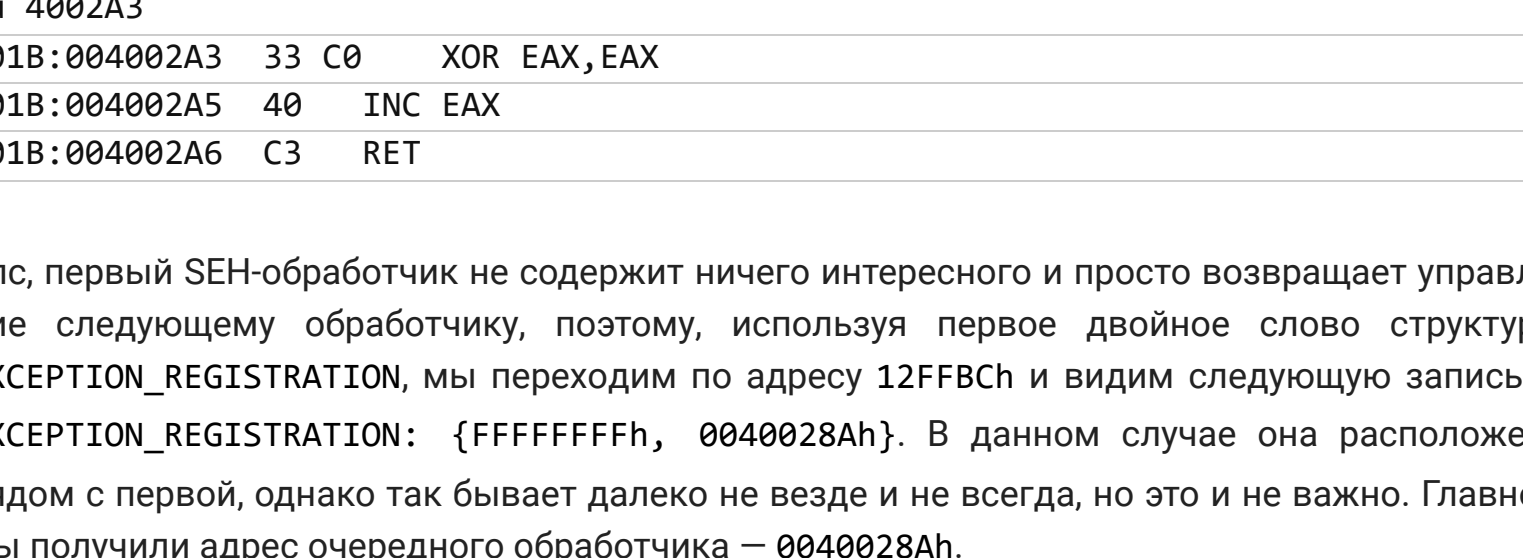
```
:dd      ; <- Отображать двойные слова
:d fs:0      ; <- Смотрим, что находится в fs:[00000000h]
0038:00000000 0012FFB4  00130000  0012D000  00000000

:d ss:12FFB4     ; Смотрим структуру EXCEPTION_REGISTRATION
:d ss:012FFB4
0023:0012FFB4 0012FFBC  004002A3  FFFFFFFF  0040028A
0023:0012FFC4 79458989  FFFFFFFF  0012FA34  7FFDF000
```

Ага, мы видим, что в `FS:[00000000h]` содержится адрес `0012FFB4h`, переходя по которому мы обнаруживаем структуру `EXCEPTION_REGISTRATION: {0012FFBC, 004002A3}`, где первое двойное слово — указатель на следующий SEH-обработчик, а второе — указатель на сам обработчик:

Дизассемблерный листинг первого SEH-обработчика в цепочке

```
:u 4002A3
001B:004002A3  33 C0    XOR EAX,EAX
001B:004002A5  40   INC EAX
001B:004002A6  C3   RET
```

Упс, первый SEH-обработчик не содержит ничего интересного и просто возвращает управление следующему обработчику, поэтому, используя первое двойное слово структуры `EXCEPTION_REGISTRATION`, мы переходим по адресу `12FFBCh` и видим следующую запись — `EXCEPTION_REGISTRATION: {FFFFFFFFh, 0040028Ah}`. В данном случае она расположена рядом с первой, однако так бывает далеко не везде и не всегда, но это и не важно. Главное, мы получили адрес очередного обработчика — `0040028Ah`.



Определение адреса машинной инструкции, передающей управление SEH-обработчику

Дизассемблерный листинг второго SEH-обработчика в цепочке

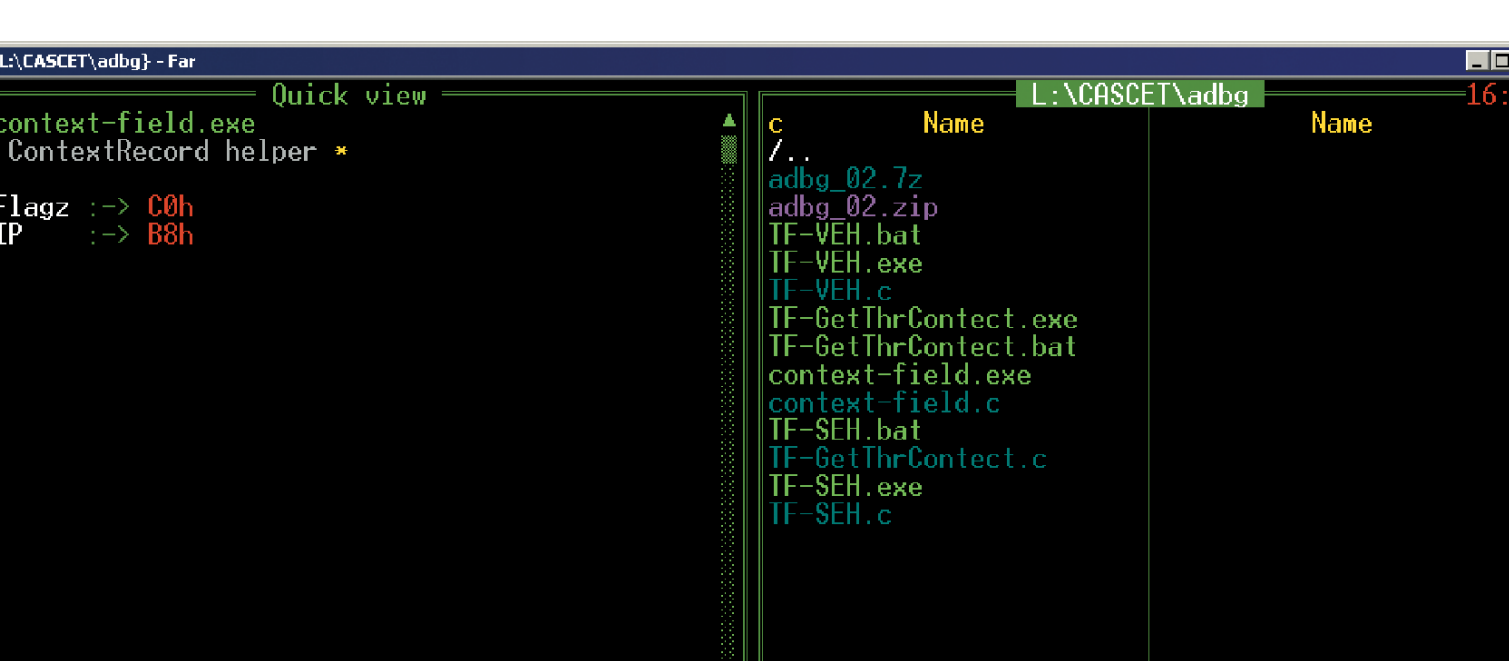
```
:u 40028A
001B:0040028A  8B 44 24 0C       MOV EAX, [ESP + 0C]
001B:0040028E  80 80 B8 00 00 00 02 ADD BYTE PTR [EAX + 000000B8], 02
001B:00400295  8B 80 C0 00 00 00     MOV EAX, [EAX + 000000C0]
001B:0040029B  A3 3C 03 40 00        MOV [0040033C], EAX
001B:004002A0  33 C0             XOR EAX, EAX
001B:004002A2  C3             RET
```

Ага, а вот тут, кажется, содержится что-то интересное! Вернувшись к прототипу функции handler, определяем, что по смещению 0Ch относительно верхушки стека расположена структура Context. Следовательно, в регистр EAX грузится регистровый контекст. А дальше... какой-то из регистров увеличивается на два байта. Но как узнать, какой? Нам поможет context helper, описанный в соответствующей врезке, с помощью которого мы узнаем, что это EIP. А вот по смещению C0h в регистровом контексте содержится EFlags, сохраняемый в глобальной переменной `0040033Ch`, на которую при желании можно поставить аппаратную точку останова на чтение/запись, чтобы посмотреть, что с ней происходит в дальнейшем:

Чтение глобальной переменной, хранящей регистр флагов

```
:bpm 40033C RW
:x
Break due to BPMB #0023:0040033C RW DR3  (ET=1.48 milliseconds)
  MSR LastBranchFromIp=00400288
    MSR LastBranchToIp=004002A7

001B:004002B2  A1 3C 03 40 00        MOV EAX,[0040033C]
001B:004002B7  F6 C4 01      TEST     AH,01        ; TF бит
001B:004002BA  74 0C             JZ  004002C8
```

Все ясно! Защита анализирует содержимое регистра флагов и, если бит трассировки взведен, заключает, что программа находится под отладкой. Как это можно обломать? Первый вариант — сбросить бит трассировки в обработчике исключений, модифицировав ячейку `[ESP+0C]->0Ch` в отладчике. Чтобы автоматизировать процесс, можно создать условную точку останова на функцию `NTDLL.DLL!KiUserExceptionDispatcher (PEXCEPTION_RECORD pExcptRec, CONTEXT *pContext)`, всегда сбрасывая TF-бит по адресу `pContext -> EFlags`, что позволит надежно скрыть отладчик от защиты. При этом перестанут работать самотрассирующие программы, отладчики прикладного уровня и еще много чего, поэтому ручная работа все же предпочтительнее автоматической.



Результат работы Context Record Helper'a

Второй вариант (совершенно не универсальный, но надежный) — изменить условный переход по адресу `004002BAh` на безусловный, чтобы он всегда рапортовал защите о сброшенном флаге трассировки. Естественно, это прокатит только с данной программой — за отказ от универсальности приходится платить.

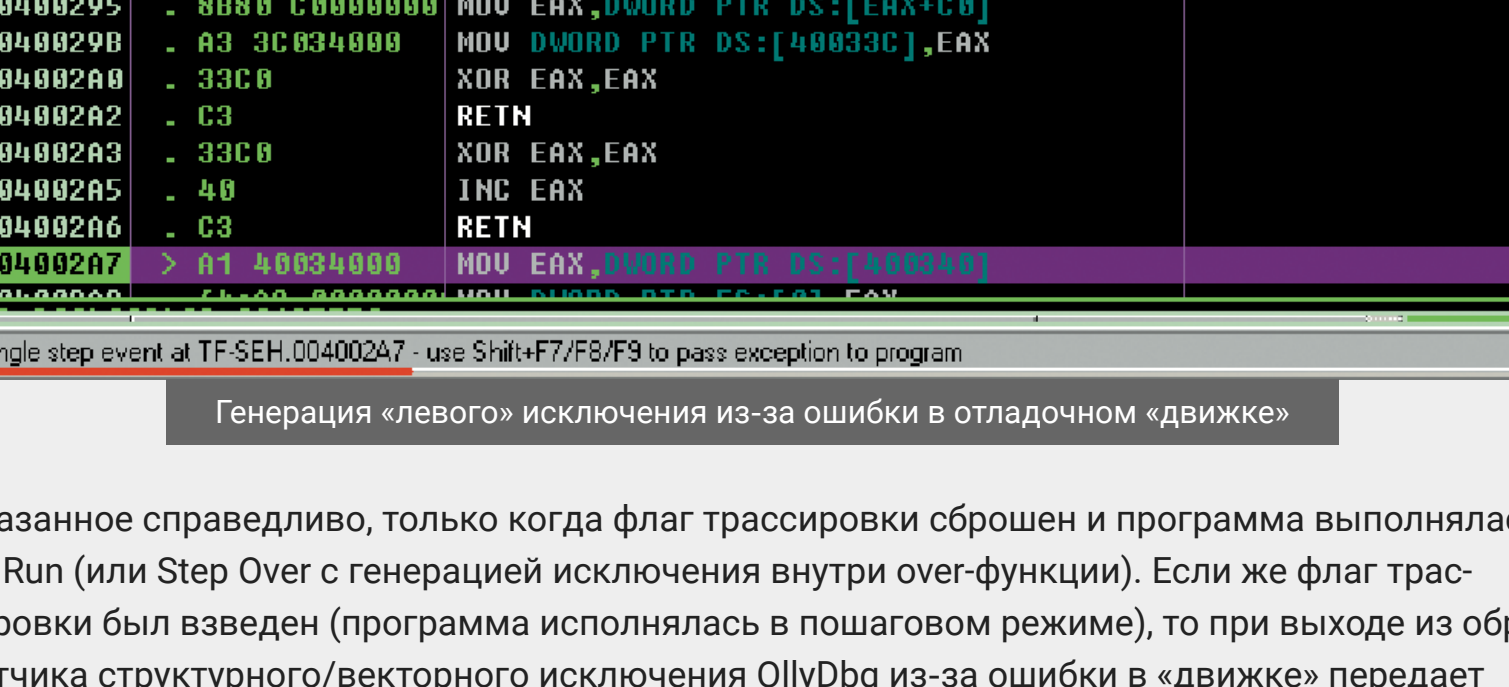
Попытка применить OllyDbg приводит к краху, поскольку он не вполне корректно обрабатывает исключения (как структурные, так и векторные). Подробности — в одноименной врезке.

WWW

Архив с исходниками и другими файлами из статьи

Ошибка OllyDbg

При возникновении исключения (неважно какого) отладчик OllyDbg останавливает выполнение программы, предлагая нам нажать Shift-F7/F8/F9 для продолжения. Первые две комбинации перебрасывают нас в начало `NTDLL.DLL!KiUserExceptionDispatcher`, предоставляя возможность самостоятельно отслеживать момент передачи управления на SEH/VEH-обработчик. А Shift-F9 выполняет обработчик на «автопилоте» и останавливает отладчик только на выходе из него. В случае двух наших crackme это будет команда, расположенная непосредственно за `mov eax,[eax]`.



Генерация «левого» исключения из-за ошибки в отладочном «движке»

Сказанное справедливо, только когда флаг трассировки сброшен и программа выполнялась по Run (или Step Over с генерацией исключения внутри over-функции). Если же флаг трассировки был взведен (программа исполнялась в пошаговом режиме), то при выходе из обработчика структурного/векторного исключения OllyDbg из-за ошибки в «движке» передает программе трассировочное исключение INT 01h, вызывая обработчик повторно. В нашем случае это приводит к увеличению регистра EIP еще на два байта и, как следствие, к краху программы. В OllyDbg 2.00c указанная ошибка до сих пор не исправлена, что ужасно напрягает.