энциклопедия антиотладочных приемов — трассировка — в погоне на TF, SEH на виражах выпуск #02h

крис касперски, ака мыщъх, a.k.a nezumi, a.k.a souriz, a.k.a elraton, no-email

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, где не действуют привычные законы и приходится долго и нудно ковыряться в недрах системы, чтобы угадать куда в следующий момент будет передано управление и как усмирить разбушевавшуюся защиту

введение

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 все равно 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, чем хвост! Но как в него попасть?! Отладчик не дает ответа на поставленный вопрос, а тупая трассировка требует нехилого количества времени…

Но SEH это ерунда. Начиная с XP появилась поддержка обработки векторных исключений (VEH), усиленная в Server 2003 и, соответственно, в Висле/Server 2008, о чем отладчики вообще не знают, открывая разработчикам защит огромные возможности для антиоладки и обламывая начинающих хакеров косяками. Мыщъх покажет 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 потрахаться все-таки придется. Секс будет. И его будет много!

Рисунок 0 мыщъх как он есть собственной персоной

SEH fundamentals

Архитектура структурных исключений подробно описана в десятках книг и сотнях статей. Настолько подробно, что в процессе чтения можно уснуть, поэтому краткое изложение основных концепций в мыщъхином 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 следующего кода XOR EAX,EAX/MOV EAX,[EAX] следующей выполняемой командой оказывается первая инструкция функции NTDLL.DLL!KiUserCallbackDispatcher. Сюрприз, да?!

В ходе своего выполнения KiUserCallbackDispatcher извлекает указатель на цепочку обработчиков структурных исключений, хранящийся по адресу FS:[00000000h], и вызывает первый обработчик через функцию ExecuteHandler (см. рис. 1), передавая ему параметры, указанные в листинге 2.

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

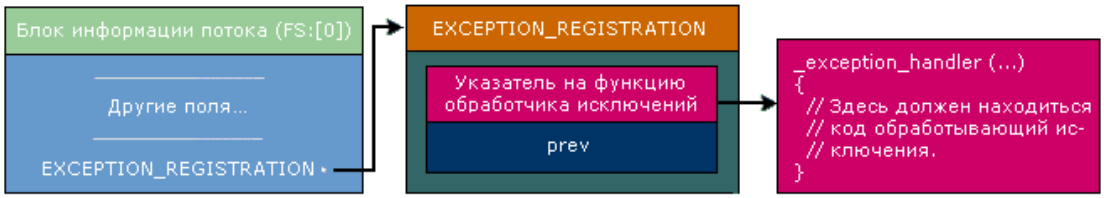

Рисунок 1 структура EXCEPTION_REGISTRATION на полях сражений

Список обработчиков структурных исключений представляет собой простой односвязанный список (см. листинг 1):

```
_EXCEPTION_REGISTRATION struc
        prev    dd      ?               ; // предыдущий обработчик, -1 - конец списка;
        handler dd      ?               ; // указатель на SEH-обработчик
_EXCEPTION_REGISTRATION ends
```

Листинг 1 формат списка обработчиков структурных исключений

Процедура обработки структурных исключений имеет следующий прототип (см. листинг 2) и возвращает одно из трех значений: EXCEPTION_CONTINUE_SEARCH, EXCEPTION_CONTINUE_EXECUTION или EXCEPTION_EXECUTE_HANDLER, описанных в MSDN.

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

Листинг 2 прототип процедуры-обработчика структурных исключений

Обработчики структурных исключений практически полностью реентерабельны, т. е. обработчик так же может генерировать исключения, корректно подхватываемые системой и начинающие раскрутку списка обработчиков с нуля. "Практически" — потому что если исключение возникает при попытке вызова обработчика (например, "благодаря" исчерпанию стека), ядро просто молчаливо прибивает процесс, но это уже дебри технических деталей, в которые мы пока не будем углубляться.

Важно отметить, что после установки своего собственного обработчика его нужно не забывать снимать, иначе можно 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, которые нам понадобятся для брачных игр со структурными исключениями.

VEH fundamentals

Начиная с XP появилась поддержка векторных исключений, являющаяся разновидностью SEH, однако, реализованная независимо от последней и работающая параллельно с ней. 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, вызывающую в свою очередь NTDLL.DLL!RtlCallVectoredExceptionHandlers, раскручивающую список векторных обработчиков с последующей передачей управления.

SEH и VEH концептуально очень схожи, предоставляют аналогичные возможности и вся разница между ними в том, что вместо ручного манипулирования со списками обработчиков теперь у нас есть API-функции AddVectoredExceptionHandler/RemoveVectoredExceptionHandler устанавливающие/удаляющие векторные обработчики из списка, указатель на который хранится в не экспортируемой переменной _RtlpCalloutEntryList внутри NTDLL.DLL (по одному

экземпляру на каждый процесс). Плюс, упростилось написание локальных/глобальных обработчиков исключений, что в случае с SEH представляет большую проблему. Однако, по прежнему, векторная обработка придерживается принципа "социального кодекса", то есть, все обработчики должны придерживаться определенных правил и ничто не мешает одному из них объявить себя самым главным и послать других на хрен.

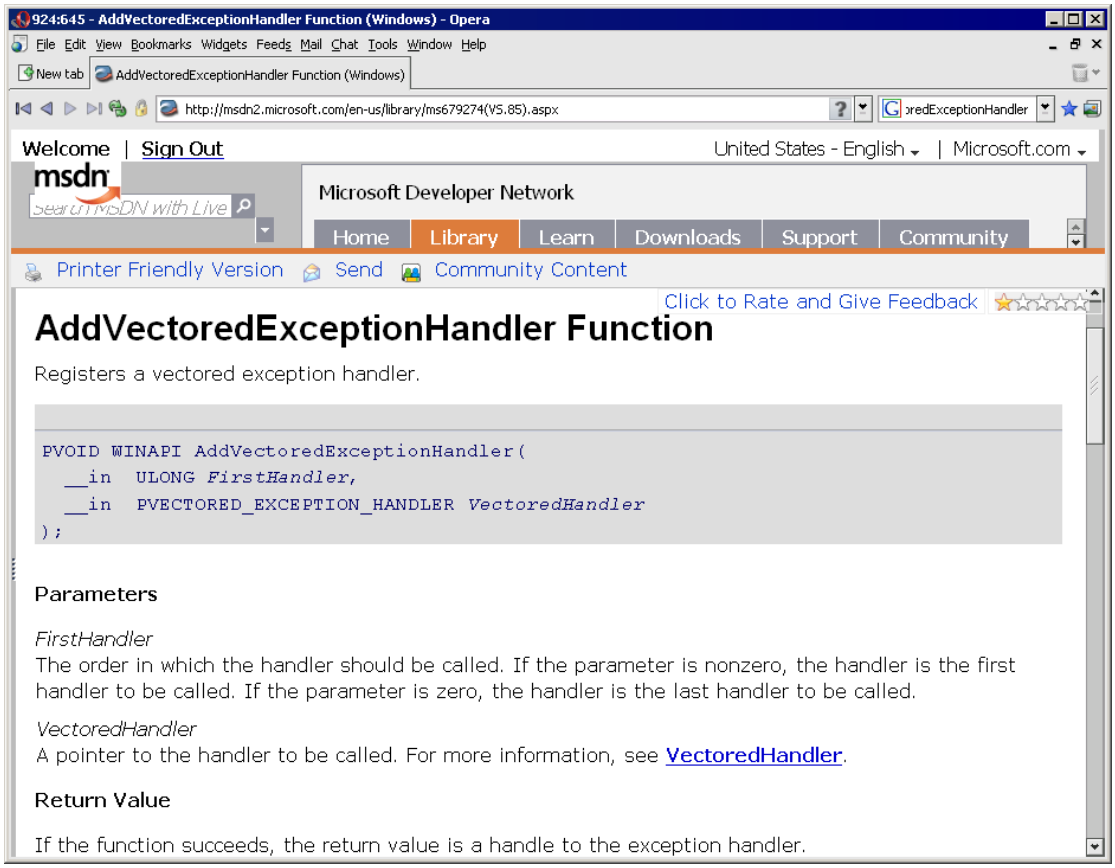

Рисунок 2 описание новых VEH-функций на MSDN

Поскольку, 9x/W2K системы все еще достаточно широко распространены, пользоваться векторной обработкой без особой на то нужны 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 же дезактивировать защитный модуль, работающий на базе VEH.

ОК, теперь пару слов о новых API функциях. AddVectoredExceptionHandler (см. рис. 2) имеет следующий прототип (см. листинг 3) и принимает два параметра, первый из которых обычно равен нулю, а второй представляет указатель на обработчик векторных исключений, прототип которого показан в листинге 4.

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

Листинг 3 прототип API-функции AddVectoredExceptionHandler, добавляющий новый векторный обработчик в список

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

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

Листинг 4 прототип процедуры обработки векторных исключений

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

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

Листинг 5 прототип API-функции RemoveVectoredExceptionHandler, удаляющей векторный обработчик из списка

эксперимент #4 – ловля TF-бита на SEH

Продемонстрируем технику отладки программ, использующий структурные исключения, на примере следующего crackme (см листинг 6), генерирующего общую ошибку доступа к памяти путем обращения по нулевому указателю, и проверяющего значение флага трассировки в заранее установленном SEH-обработчике (на самом деле, здесь, для запутывания хакера, назначается сразу _два_ обработчика — первый обработчик ничего не делает и тупо возвращает управление, а вот второй — считывает регистровый контекст, извлекает оттуда содержимое флага трассировки и увеличивает значение EIP на два байта – длину инструкции mov eax, [eax], вызывавшей исключение).

```
#include <windows.h>
char noo[]="debugger is not detected";
char yes[]="debugger is detected :-)";

DWORD EFLAGZ; DWORD old_seh; char *p=noo;

nezumi()
{
        __asm{

        // int 03h

        mov ecx, fs:[0]               ; // save old SEH handler
        mov old_seh, ecx

        ; // set-up our SEH handler
        ; //-----------------------------------------------------------------------
        push offset main_handler      ; // the last seh-handler actually
        push -1                       ; // the last handler in the chain
        mov eax,esp                   ; // eax keeps the point to main_handler

        push offset dump_handler      ; // it's a dump handler to obfuscate code
        push eax                      ; // <- dump handler is the first seh-handler

        mov fs:[0], esp               ; // set-up our SEH handler chain
        ; //=======================================================================

        xor eax,eax                   ; // rise an exception...
        mov eax,[eax]                 ; // ...to pass control to dump_handler

        jmp end_asm                   ; // to the exit


main_handler:                         ; // good place to check TF-flag
        mov eax,[esp + 0Ch]           ; // EAX -> ContextRecord
        add [eax + 0B8h], 2           ; // change EIP to skip "mov eax,[eax]"
        mov eax, [eax+0C0h]           ; // EAX -> EFlags
        mov EFLAGZ, eax               ; // save EFlagz

        //mov ecx, [old_seh]          ; // don't try to resore SEH inside handler,
        //mov fs:[0], ecx             ; // coz it has no effect, do it ouside handlr

        xor     eax,eax               ; // EXCEPTION_CONTINUE_SEARCH
        retn                          ; // it's done!

dump_handler:                         ; // do nothing here (to confuse a hacker)
        xor     eax, eax              ;
        inc eax                       ; // EXCEPTION_EXECUTE_HANDLER
        retn                          ; // return to system to execute main_handler

end_asm:
        mov eax, old_seh              ; // safe place to restore the original SEH
```

```
        mov fs:[0], eax
        }

        // display the result
        if (EFLAGZ & (1<<8)) p = yes; MessageBox(0, p, p, MB_OK); ExitProcess(0);
}
```

Листинг 6 TF-SEH.c — ловля флага трассировки на структурные исключения

Для упрощения отладки из программы выкинуто все лишнее (и стартовый код в том числе), поэтому для ее сборки применяется специальный командный файл следующего содержания (см. листинг 7).

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

Листинг 7 TF-SEH.bat — сборка программы без стартового кода

Компилируем программу и загружаем ее в любой подходящий отладчик (например, Soft-Ice), а если же загрузка обламывается (известный глюк Soft-Ice), раскомментируем строку с командой "int 03h", пересобираем программу, пишем в Soft-Ice: "i3here on" и запускаем программу еще раз. 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.

Приехали! Дальше продолжать трассировку нет смысла, нужно найти способ _быстро_ найти адрес структурного обработчика. А как его найти?! Например, можно посмотреть, что находится в памяти по указателю FS:[00000000h]:

```
: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
```

Листинг 8 определение списка адресов SEH-обработчиков путем просмотра fs:0

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:

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

Листинг 9 дизассемблерный листинг первого SEH-обработчика в цепочке

Опс, 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 расположенную рядом с первой, однако, так бывает далеко не везде и не всегда, но это и не важно. Главное, что мы получили адрес очередного обработчика — 0040028Ah.

```
: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
```

Листинг 10 дизассемблерный листинг второго SEH-обработчика в цепочке

Ага, а вот тут уже кажется содержится что-то интересное! Возвращаясь к прототипу функции handler (см. листинг 2), определяем что по смещению 0Ch относительно верхушки стека расположена структура Context. Следовательно, в регистр EAX грузиться регистровый контекст. А дальше… какой-то из регистров увеличивается на два байта. Но как узнать какой?! В этом 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:

```
: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
```

Листинг 11 чтение глобальной переменной, хранящей регистр флагов

Все ясно! Защита анализирует содержимое регистра флагов и, если бит трассировки взведен, заключает, что программа находится под отладкой (см. рис. 3). Как можно это обломать?! Возможные варианты: сбросить бит трассировки в обработчике исключений путем модификации ячейки [ESP+0C]->0Ch в отладчике. Чтобы автоматизировать этот 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, однако, при этом перестанут работать самотрассирующие программы, отладчики прикладного уровня и еще много чего, поэтому, ручная работа все же предпочтительнее автоматической.

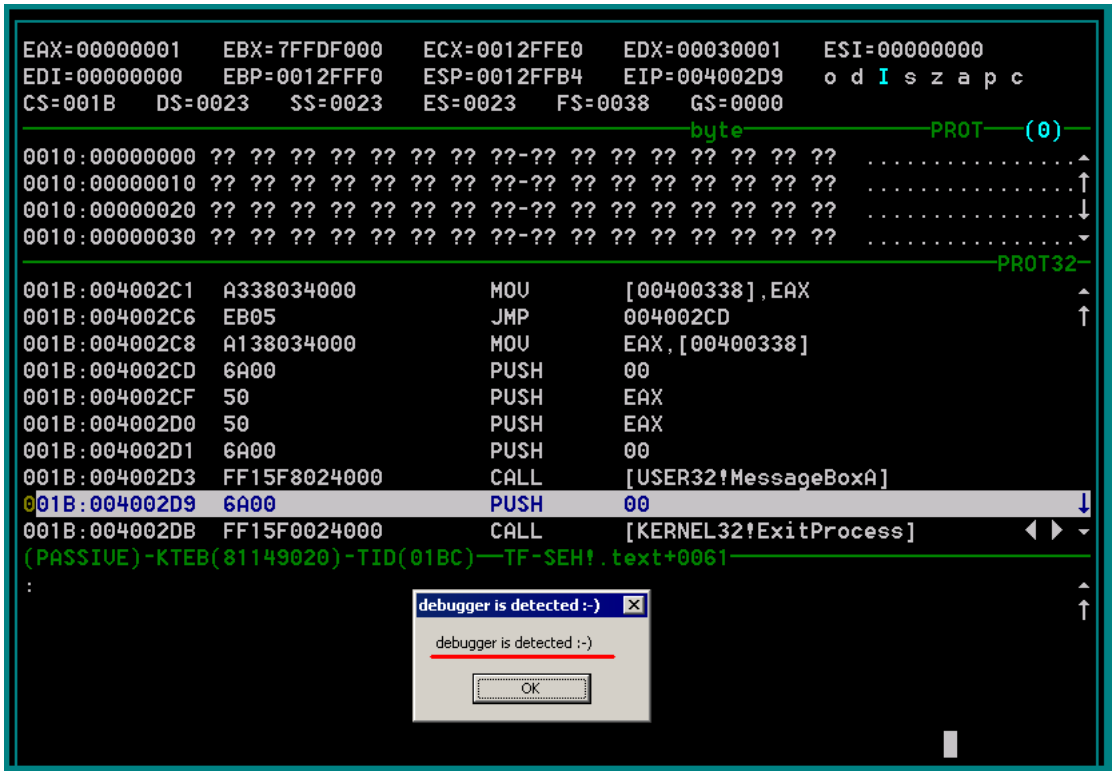

Рисунок 3 отладчик успешно обнаружен ;-)

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

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

>>> врезка context helper

Программируя на языке Си, мы используем готовые определения, даже не задумываясь по каким смещениям расположены интересующие нас поля, перекладывая эту заботу на компилятор.

Дизассемблируя программу, мы сталкиваемся с обратной задачей и хотя IDA Pro поддерживает определение структур, позволяя преобразовать произвольный указатель к регистровому контексту, это не лучший выход из ситуации, поскольку в отладчике нам все равно придется иметь дело с константами.

Чтобы не ломать голову какому регистру они принадлежат, достаточно распечатать смещение всех полей структуры CONTEXT, определить которые можно, например, следующим путем:

```
CONTEXT *ContextRecord=0;              // see WINNT.H
printf("EFlagz :-> %02Xh\n",&ContextRecord->EFlags);
printf("EIP    :-> %02Xh\n",&ContextRecord->Eip);
```

Листинг 12 context-field.c – определение смещение полей структуры CONTEXT (см. рис. 4)

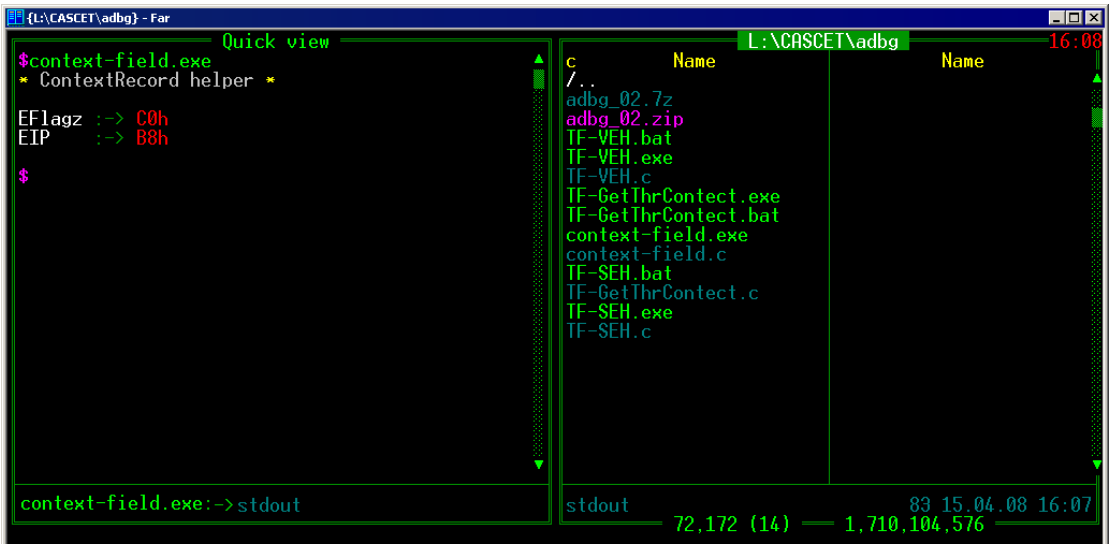

Рисунок 4 результат работы Context Record Helper'a

эксперимент #5 – ловля TF-бита на VEH

А теперь испытаем векторные исключения, поддерживаемые, как уже говорилось, начиная с XP (см. листинг 13):

```
char noo[]="debugger is not detected";
char yes[]="debugger is detected :-)";
DWORD EFLAGZ; char *p=noo;

// define VEH-func prototypes
typedef LONG (NTAPI *PVECTORED_EXCEPTION_HANDLER)(struct _EXCEPTION_POINTERS*);
typedef PVOID (WINAPI *AddVEH)(ULONG, PVECTORED_EXCEPTION_HANDLER);

// our vector-handler
LONG CALLBACK VectoredHandler(PEXCEPTION_POINTERS ExceptionInfo)
{
        EFLAGZ=ExceptionInfo->ContextRecord->EFlags; // save EFlags
        ExceptionInfo->ContextRecord->Eip+=2;        // skip mov eax,[eax]
        return EXCEPTION_CONTINUE_EXECUTION;         // continue execution
}
```

```
souriz()
{
        AddVEH AddVectoredExceptionHandler;          // pointer to the func

        // get AddVectoredExceptionHandler address (if there is one)
        AddVectoredExceptionHandler =
                (PVOID(WINAPI *)(ULONG, PVECTORED_EXCEPTION_HANDLER))
                                GetProcAddress(LoadLibrary("KERNEL32.DLL"),
                                                "AddVectoredExceptionHandler");

        if (!AddVectoredExceptionHandler)            // check if we got it or not
        {
                MessageBox(0,"VEH works only on XP or S2K3!",0,MB_OK); ExitProcess(0);
        }

        // set-up a new vectored exception handler
        AddVectoredExceptionHandler(0,VectoredHandler);
}

nezumi()
{
        __asm{
                // int 03h                           ; // for soft-ice
        }
        souriz();                                    ; // do everything has to be done

        __asm{
                xor eax,eax
                mov eax,[eax]                        ; // rise an exception
        }

        // display the result
        if (EFLAGZ & (1<<8)) p = yes; MessageBox(0, p, p, MB_OK); ExitProcess(0);
}
```

Листинг 13 TF-VEH.c – ловля флага трассировки на структурные исключения

Как и в предыдущем примере из программы выкинут стартовый код и она собирается аналогичным образом. Загрузив исполняемый файл в отладчик, делаем step over через функцию souriz (первый CALL), поскольку, она нам неинтересна и трассируем пару машинных команд xor eax,eax/mov eax,[eax], генерирующих исключение, перебрасывающее отладчик на функцию NTDLL.DLL!KiUserExceptionDispatcher. Как теперь определить — какой обработчик будет вызван?! Вопрос достаточно актуален, особенно если программа попеременно использует как структурные, так и векторные исключения, поэтому попытка установки точки останова на API-функцию AddVectoredExceptionHandler с последующим отслеживанием адресов всех обработчиков, может и не привести к желаемому эффекту…

К счастью существует универсальный способ определения адресов действительных обработчиков, одинаково хорошо работающий как со структурными, так и с векторными исключениями, описанный во врезке "капкан для исключений".

>>> врезка капкан для исключений

Передача управления на обработчики исключений, установленные пользователем (в смысле, программистом) происходит внутри функции NTDLL.DLL!KiUserExceptionDispatcher, вызывающей служебную неэкпортируемую процедуру, адрес которой варьируется в зависимости от версии Windows и установленных сервис-паков, однако, в рамках конкретно взятой версии он всегда постоянен — достаточно определить его один-единственный раз, записать на бумажку, приклеенную на стену, и устанавливать точку останова всякий раз, когда мы хотим отследить момент передачи управления на пользовательский обработчик.

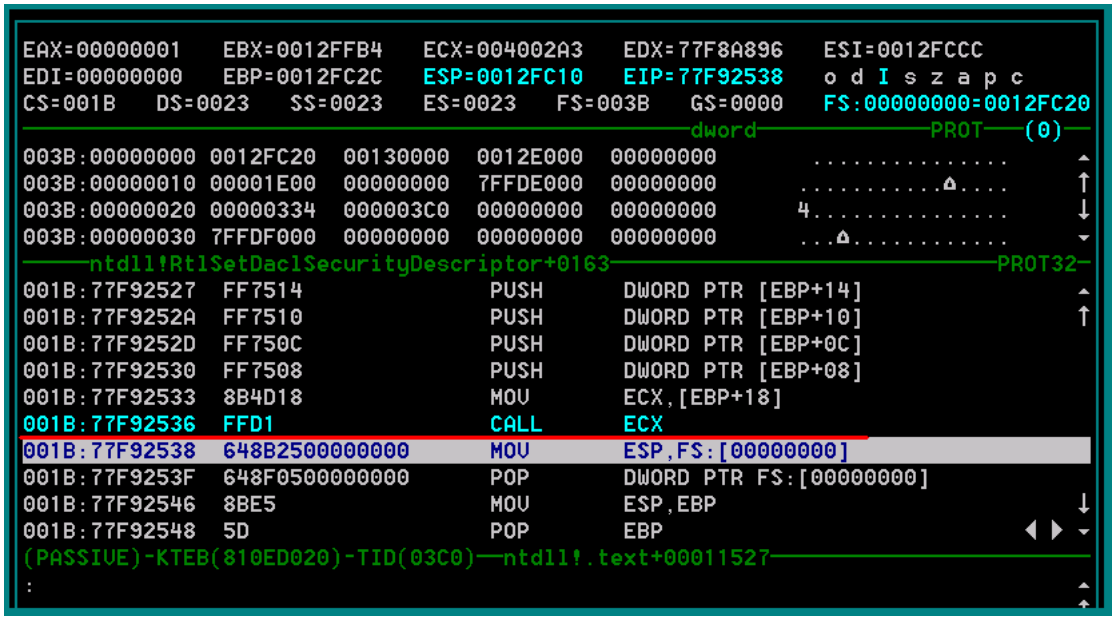

Рисунок 5 определение адреса машинной инструкции, передающей управление SEH-обработчику (в данном случае это инструкция CALL ECX, расположенная по адресу 77F92536h)

Техника определения проста до безобразия. Начнем со структурных исключений. Возвращаемся к листингу 6. Загрузив программу в отладчик, устанавливаем точку останова на метку dump_handler, после чего говорим отладчику "Run", дожидаясь его всплытия. Трассируем обработчик вплоть до выхода из него по команде RETN, попадая в служебную системную функцию, вызвавшую обработчик (см. рис. 5).

Мы увидим приблизительно следующий код:

```
.text:77F68CE5        push    [ebp+arg_C]
.text:77F68CE8        push    [ebp+arg_8]
.text:77F68CEB        push    [ebp+arg_4]
.text:77F68CEE        push    [ebp+arg_0]
.text:77F68CF1        mov     ecx, [ebp+arg_10]
.text:77F68CF4        call    ecx                     ; call seh_handler()
.text:77F68CF6        mov     esp, large fs:0         ; <- сюда мы входим
.text:77F68CFD        pop     large dword ptr fs:0
```

Листинг 14 системный код, вызывающий пользовательский SEH-обработчик

Как нетрудно догадаться, call ecx (расположенная на мыщъхиной машине по адресу 77F68CF4h) – и есть команда, передающая управление на SEH-обработчик. Запоминаем (записываем на бумажку) ее адрес и рулим. В смысле устанавливаем сюда точку останова по исполнению и мониторим вызов SEH-обработчиков.

А теперь — тоже самое только для векторных исключений. Загружаем в отладчик программу TF-VEH.exe, устанавливаем точку останова на VectoredHandler, говорим "Run" и затем, дождавшись, всплытия отладчика, трассируем функцию пока не встретим RETN:

```
.text:77F68C81        lea     eax, [ebp+var_8]
.text:77F68C84        push    eax
.text:77F68C85        call    dword ptr [esi+8]       ; call VectoredHandler()
.text:77F68C88        cmp     eax, 0FFFFFFFFh         ; ← сюда мы выходим
```

Листинг 15 системный код, вызывающий пользовательский VEH-обработчик

Как и следовало ожидать, адрес call'a, вызывающего VEH-обработчик, отличается от его SEH-собрата и в данном случае равен 77F68C85h. Установив точки останова по исполнению на эти два адреса (77F68CF4h и 77F68C85h), мы легко и быстро сломаем любую защиту.

>>> врезка ошибка OllyDbg

При возникновении исключения (не важно 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 нам самостоятельно отслеживать момент передачи управления на SEH/VEH-обработчик, а Shift-F9 выполняет обработчик на "автопилоте" и останавливает отладчик только по выходу из него. В случае двух наших crackme это будет команда, расположенная непосредственно за mov eax,[eax].

Сказанное справедливо только для случая, если флаг трассировки был 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 повторный вызов обработка (см. рис. 6). В нашем случае это приводит к увеличению регистра EIP еще на два байта и, как следствие, к краху программы. В OllyDbg 2.00c ошибка данная ошибка до сих пор не исправлена, что ужасно напрягает.

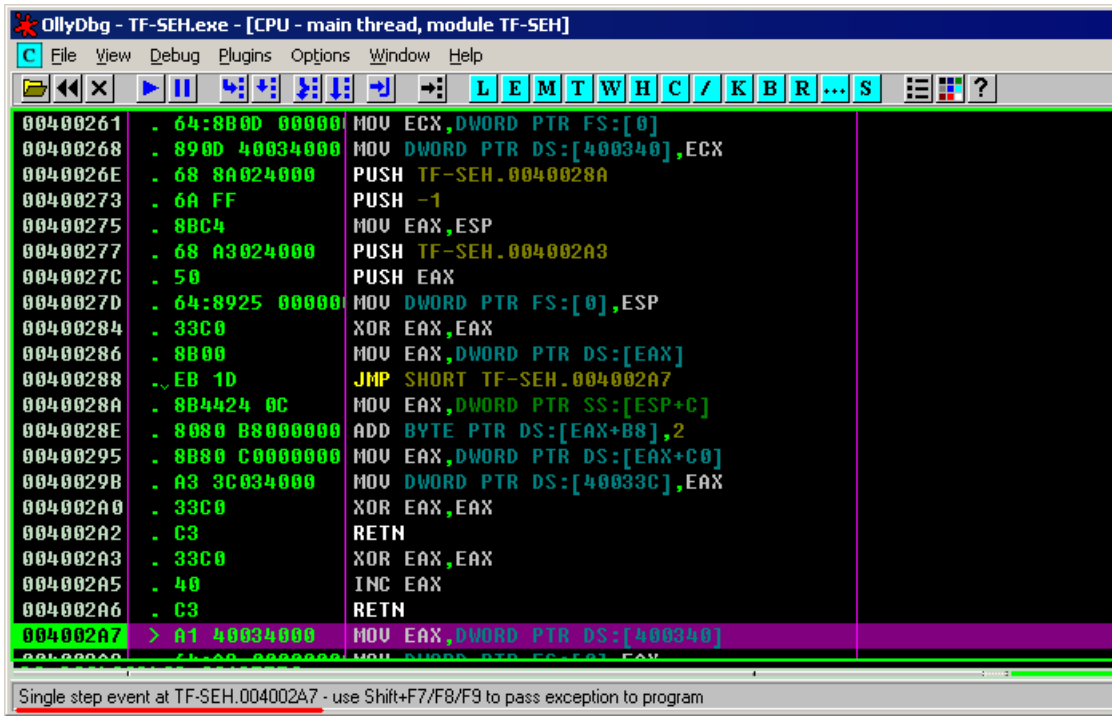

Рисунок 6 генерация "левого" исключения из-за ошибки в отладочном "движке"