**Централизованная обработка исключений**

Беляев Алексей

Когда приложение перестает работать в офисе у разработчика, найти ошибку и исправить ее не составляет труда. Когда же приложение отказывает у клиента, то трудно найти общий язык с огорченным пользователем, и понять что ты сделал не так...

**Введение**

Что такое ошибка? Отвечая на этот вопрос кратко, можно сказать, что ошибка – это отклонение от описанного поведения. Для разработчика это означает, что необходимо искать и исправлять причину этого отклонения. Для программиста контроля качества ПО это означает, что необходимо доработать тесты и включить их в базовый цикл тестирования приложения. Для руководства это означает увеличение времени и затрат на разработку продукта.

Ошибки бывают разные, одни воспроизводятся легко, другие трудно. На поиск одних тратится немного времени, на поиск других уходят дни. Основная проблема поиска ошибки зачастую связана с недостатком информации по ее воспроизведению или состоянии приложения в момент возникновения ошибки. Если бы разработчик имел информацию о том, какая строчка программы содержит ошибку, ему не составило бы труда исправить ее.

Единственный способ избежать ошибок в программах – это писать код без ошибок. Но человек не может не делать ошибок, поэтому в любой программе они есть. Единственное, что разработчик может постараться сделать – это минимизировать количество ошибок, а также облегчить их поиск и исправление. В этой статье рассматривается способ, помогающий ускорить поиск и исправление ошибок.

**Windows и необработанные исключения**

Когда в приложении, работающем под управлением ОС Windows (от 9х до ХР), возникает необработанное исключение, операционная система обрабатывает его, создает dump-файл и записывает в него информацию, анализируя которую можно восстановить состояние приложения и быстро найти ошибку. К информации, которую сохраняет операционная система, относится:

информация о потоках;

информация о загруженных модулях;

информация об исключении;

информация о системе;

информация о памяти.

Всю собранную информацию операционная система предлагает послать в Microsoft для последующего анализа. Учитывая полезность этой информации и выгоду, которую предоставляет владение ей, необходимо на этапе проектирования приложения заложить возможность для сбора информации о приложении во время возникновения исключения для последующего ее анализа. Тогда разработчики будут иметь больше информации для поиска и исправления ошибок.

Каким образом Windows XP определяет, что в приложении произошло необработанное исключение? Ответить на этот вопрос можно, если разобраться с механизмом структурированной обработки исключений (SEH). Все версии Windows, начиная с версии Windows 95 и Windows NT, поддерживают этот механизм обработки исключений, позволяющий операционной системе и приложению тесно взаимодействовать в случае возникновения исключительной ситуации. И если в каком-либо приложении возникает необработанное исключение, операционная система обрабатывает его и завершает приложение.

**Структурированная обработка ошибок**

Структурированная обработка исключений (SEH) – это предоставляемый системой сервис, вокруг которого библиотеки современных языков программирования реализуют свои собственные функции для работы с исключениями.

C++-программисты наверняка знакомы с SEH в основном по использованию конструкций \_\_try ... \_\_except. Встретив в теле функции конструкцию \_\_try … \_\_except, компилятор, поддерживающий SEH, генерирует код для регистрации обработчика исключения. Затем, после возникновения исключения, операционная система ищет подходящий обработчик. Если подходящий обработчик не найден, операционная система создает dump-файл и завершает работу приложения.

Таким образом, перед нами стоит задача – сделать так, чтобы после возникновения в приложении необработанного исключения вызывался наш обработчик.

Для решения этой задачи необходимо выяснить, как операционная система ищет обработчик исключения. В поисках ответа на этот вопрос я углублялся в документацию по механизму структурированных исключений, анализировал системный ассемблерный код, смотрел, что генерирует компилятор, когда встречает конструкцию \_\_try … \_\_except, но подходящего решения не находилось. Ни в SDK, ни в DDK я не нашел ничего, что могло бы ответить на этот вопрос. Анализируя код, генерируемый компилятором для конструкции \_\_try … \_\_except, я увидел, что для каждого нового обработчика исключений в стеке создается запись, которая помещается в связанный список. Вот пример простой функции, который поможет понятнее объяснить это:

|  |
| --- |
| void foo()  {  \_\_try  {  }  \_\_except(1)  {  }  } |

Код, который был сгенерирован компилятором VC 7.0:

|  |
| --- |
| void foo()  {  00411DE0 push ebp  00411DE1 mov ebp,esp  00411DE3 push 0FFFFFFFFh  00411DE5 push 424140h  00411DEA push offset @ILT+390(\_\_except\_handler3) (41118Bh)  00411DEF mov eax,dword ptr fs:[00000000h]  00411DF5 push eax  00411DF6 mov dword ptr fs:[0],esp  00411DFD add esp,0FFFFFF38h  00411E03 push ebx  00411E04 push esi  00411E05 push edi  00411E06 lea edi,[ebp-0D8h]  00411E0C mov ecx,30h  00411E11 mov eax,0CCCCCCCCh  00411E16 rep stos dword ptr [edi]  00411E18 mov dword ptr [ebp-18h],esp  \_\_try  00411E1B mov dword ptr [ebp-4],0  00411E22 mov dword ptr [ebp-4],0FFFFFFFFh  00411E29 jmp $L19329+0Ah (411E3Bh)  {  }  \_\_except(1)  00411E2B mov eax,1  $L19330:  00411E30 ret  $L19329:  00411E31 mov esp,dword ptr [ebp-18h]  00411E34 mov dword ptr [ebp-4],0FFFFFFFFh  {  }  }  00411E3B mov ecx,dword ptr [ebp-10h]  00411E3E mov dword ptr fs:[0],ecx  00411E45 pop edi  00411E46 pop esi  00411E47 pop ebx  00411E48 mov esp,ebp  00411E4A pop ebp  00411E4B ret |

Теперь давайте посмотрим, что делает этот ассемблерный код. В начале функции создается фрейм стека. В этом нет ничего необычного – такой код есть в большинстве функций, но вот несколько следующих инструкций показались мне очень интересными:

|  |
| --- |
| 00411DE3 push 0FFFFFFFFh  00411DE5 push 424140h  00411DEA push offset @ILT+390(\_\_except\_handler3) (41118Bh)  00411DEF mov eax,dword ptr fs:[00000000h]  00411DF5 push eax  00411DF6 mov dword ptr fs:[0],esp |

Вначале в стек кладется -1 (как оказалось впоследствии, просто резервируется место в стеке), а затем в стек записывается адрес статической переменной и адрес обработчика исключения. Если присмотреться к последним трем инструкциям, то можно увидеть, что из памяти по адресу fs:[0] считывается какое-то число, и кладется в стек, а на его место заносится текущий указатель стека. В принципе ничего подозрительного тут нет, но если расположить эти три инструкции последовательно несколько раз, то станет заметно, что они формируют связанный список, причем первый элемент этого списка всегда указывает на предыдущий элемент. На выходе из функции находится код, который восстанавливает предыдущее значение переменной по адресу fs:[0] :

|  |
| --- |
| 00411E3B mov ecx,dword ptr [ebp-10h]  00411E3E mov dword ptr fs:[0],ecx |

Таким образом, если функция имеет в себе конструкцию \_\_try … \_\_except, то компилятор создает в стеке запись о новом обработчике исключений и помещает информацию о ней в список обработчиков. Придя к такому выводу, я начал искать хоть какую-то информацию об обработчиках исключений и нашел публикацию, написанную Matt Pietrek-ом 7 лет назад (A Crash Course on the Depths of Win32 Structured Exception Handling). В этой статье описана структура SEH, и подтверждаются выводы, сделанные путем анализа кода приведенной выше функции. Изучив эту статью и проверив написанное в ней, я обнаружил, что с тех пор в области обработки исключений практически ничего не изменилось.

Из статьи следует, что по адресу fs:[0], находится начало связанного списка зарегистрированных обработчиков исключения, элементами которого являются структуры типа \_EXCEPTION\_REGISTRATION, расположенные в стеке.

|  |
| --- |
| struct \_EXCEPTION\_REGISTRATION  {  // указатель на следующую запись  \_EXCEPTION\_REGISTRATION \*prev;  // обработчик исключения, созданный Runtime библиотекой  SEHHandler handler;  // указатель на структуру, описывающий блок \_\_try…\_\_except  PSCOPETABLE scopetable;  // уровень вложенности текущего блока try  int trylevel;  // указатель на следующую запись  int \_ebp;  }; |

В этой структуре handler является процедурой обработки исключения. Прототип этой функции приведен ниже:

|  |
| --- |
| typedef int (\*SEHHandler)(PEXCEPTION\_RECORD, PEXCEPTION\_REGISTRATION, PCONTEXT, void\*); |

Как видите, функция обработчика исключения принимает 4 параметра. Первый параметр имеет тип PEXCEPTION\_RECORD – это указатель на структуру, содержащую информацию об исключении. Вы можете найти объявление этой структуры в заголовочном файле winnt.h:

|  |
| --- |
| typedef struct \_EXCEPTION\_RECORD {  DWORD ExceptionCode;  DWORD ExceptionFlags;  \_EXCEPTION\_RECORD \*ExceptionRecord;  PVOID ExceptionAddress;  DWORD NumberParameters;  ULONG\_PTR ExceptionInformation [EXCEPTION\_MAXIMUM\_PARAMETERS];  } EXCEPTION\_RECORD; |

Описание наиболее значимых полей этой структуры приведено ниже:

ExceptionCode – тип исключения.

ExceptionFlags – флаг исключения.

ExceptionAddress – адрес участка кода, где возникло исключение.

Второй параметр функции содержит в себе указатель на структуру PEXCEPTION\_REGISTRATION. Ее описание и назначение было приведено выше.

Третий параметр указывает на переменную типа PCONTEXT и несет информацию о состоянии регистров во время исключения.

Таким образом, механизм обработки исключений становится более или менее ясным, т.е. когда в приложении возникает исключение, операционная система берет текущий указатель по адресу fs:[0] и просматривает список обработчиков в поисках нужного обработчика исключения. Если обработчик найден, она передает ему управление. В противном случае операционная система выполняет свой обработчик, который вызывает функцию UnhandledExceptionFilter. Значит, для получения управления в случае возникновения необработанного исключения, нужно зарегистрировать свой обработчик и расположить его в вершине этого списка. Но мир программирования не был бы таким интересным, если бы все было так просто! Давайте пройдем дальше и посмотрим, что происходит во время старта приложения, и какую роль в обработке исключений играет runtime-библиотека.

**Старт приложения и инициализация runtime**

Когда операционная система загружает приложение, она считывает содержимое файла с диска в память, загружает все необходимые для работы приложения внешние библиотеки и инициализирует таблицу импорта адресами реальных функций. После этого загрузчик передает управление на точку входа приложения. В случае С++-приложений, написанных с использованием Microsoft Visual Studio 6.0 (7.x, 8.0), управление передается функции WinMainCRTStartup или wWinMainCRTStartup, в зависимости от версии runtime-библиотеки – UNICODE или Multi-Byte Character Set (MBCS). Эта функция подготавливает приложение к работе, инициализирует runtime, выполняет конструкторы всех статических переменных и передает управление на точку входа, определенную разработчиком. Если внимательно рассмотреть эту функцию, можно увидеть, что инициализация пользовательских статических переменных и передача управления на пользовательскую точку входа осуществляется внутри блока \_\_try … \_\_except. Углубившись в исследование фильтра исключений runtime-библиотеки, я обнаружил, что в случае возникновения необработанного исключения он вызывает функцию UnhandledExceptionFilter.

Функция UnhandledExceptionFilter находится в библиотеке kernel32.dll и присутствует в Windows, начиная с версии Windows 95/NT. Назначение этой функции, как видно из названия - обработка необработанных исключений и завершение приложения. В зависимости от того, как запущено приложение, UnhandledExceptionFilter ведет себя по-разному. Так, если приложение находится под отладкой, она передает управление отладчику, в противном случае она выводит диалоговое окно «Application Error». Значит, для обработки необработанных исключений, следует установить свой обработчик не на вершину списка обработчиков, как казалось раньше, а перед обработчиком Runtime библиотеки.

**Установка обработчика верхнего уровня**

Давайте немного отдохнем и суммируем все сказанное выше. SEH – это системный сервис, в котором унифицирован механизм обработки исключений, все обработчики текущего потока регистрируются в списке регистрации обработчиков исключений. Если в функции встречается конструкция \_\_try … \_\_except, то создается код, который регистрирует новый обработчик исключения и помещает информацию о нем в стек. Во время завершения функции (а точнее, после того, как управление вышло из секции \_\_try), функция разрегистрирует обработчик. Значит, если к текущему моменту в стеке находится три функции, каждая из которых установила свой обработчик исключения, то в списке обработчиков исключения должно находиться по крайней мере три обработчика, а в стеке должны находиться три записи об обработчиках исключений. Информация о текущем обработчике доступна по адресу fs:[0]. Runtime-библиотека регистрирует свой обработчик исключений, который (если исключение не обрабатывается приложением) вызывает функцию UnhandledExceptionFilter, после чего приложение завершается с выводом диалогового окна «Application Error».

Теперь настало время написать код, который бы использовал сказанное выше и подтвердил правильность наших суждений. Давайте напишем простую функцию, которая бы пробегала по всем зарегистрированным обработчикам и выводила информацию о них на экран. Код функции приведен ниже:

|  |
| --- |
| void zWalkThroughSEH()  {  \_EXCEPTION\_REGISTRATION \* pVCExcRec;  \_\_asm mov eax, FS:[0]  \_\_asm mov [pVCExcRec], EAX  // Перебираем блоки в связанном списке. 0xFFFFFFFF означает конец списка.  printf("Exception Registration chain:\n");  while (0xFFFFFFFF != (unsigned)(UINT\_PTR)pVCExcRec)  {  printf("\tCurrent SEH record: 0x%X\n\tPrev SEH Record: 0x%X\n\tHandler: 0x%X\n\n",  pVCExcRec,  pVCExcRec->prev,  pVCExcRec->hander);  pVCExcRec = (\_EXCEPTION\_REGISTRATION \*)(pVCExcRec->prev);  }  } |

Вызов эту функции после начала выполнения функции main покажет, что к моменту выполнения функции main в списке обработчиков уже зарегистрированы два обработчика. Текущий обработчик, как было показано раньше, установлен библиотекой Runtime. А вот последний установлен системой.

|  |
| --- |
| Exception Registration chain:  Current SEH record: 0x12FFB0  Prev SEH Record: 0x12FFE0  Handler: 0x41123A  Current SEH record: 0x12FFE0  Prev SEH Record: 0xFFFFFFFF  Handler: 0x77E94809 |

Как вы, наверное, уже догадались, создавать свой обработчик и располагать его за системным нет никакого смысла, поскольку исключение будет обработано в runtime-библиотеке, и приложение будет завершено. Тогда я попытался создать свой обработчик и расположил информацию о нем перед обработчиком runtime-библиотеки, выделив для него место в динамической памяти, но увы, мое приложение просто было выгружено из памяти после возникновения исключения, а вставленный обработчик не был выполнен. Как оказалось, так делать нельзя потому, что все записи списка обработчиков исключений должны лежать в стеке, причем каждая следующая запись должна быть расположена выше предыдущей.

Итак, нельзя расположить информацию об обработчике перед информацией о runtime-обработчике. Но никто не мешает переписать значение поля hander обработчика runtime-библиотеки, установив его так, чтобы он указывал на нашу функцию. Код, который реализует это, приведен ниже.

|  |
| --- |
| void zHookUpSEHChain(SEHHandler handler)  {  \_EXCEPTION\_REGISTRATION \* pVCExcRec;  \_\_asm mov eax, FS:[0]  \_\_asm mov [pVCExcRec], EAX  // Перебираем блоки в связанном списке. 0xFFFFFFFF означает конец списка.  while (0xFFFFFFFF != (unsigned)(UINT\_PTR)pVCExcRec)  {  if ( (unsigned)(UINT\_PTR)pVCExcRec->prev->prev == 0xFFFFFFFF)  {  defHandler = pVCExcRec->hander;  pVCExcRec->hander = handler;  break;  }  pVCExcRec = (\_EXCEPTION\_REGISTRATION \*)(pVCExcRec->prev);  }  } |

где

defHandler – статическая переменная, в которой сохраняется адрес предыдущего обработчика.

handler – наш обработчик исключения.

Разумеется, внимательный читатель уже заметил некоторую нелогичность в этих суждениях. Зачем пытаться зарегистрировать свой обработчик таким изощренным методом, если достаточно поместить свой блок \_\_try \_\_except в функции main? Дело в том, что при использовании MFC, ATL или какой-то иной библиотеки не имеется доступа к пользовательской точке входа, и, стало быть, нельзя установить свой обработчик.

Сейчас пришло время собрать воедино все сказанное выше и написать небольшую программу, иллюстрирующую способ установки обработчика. К статье прилагается файл ehSimple.cpp, в котором вы найдете код установки обработчика. Первый обработчик реализован в виде класса CatUnhandledExceptionFilter, объявленного следующим образом:

|  |
| --- |
| class CatUnhandledExceptionFilter  {  private:  // SEHHandler oldHandler – переменная, в которую будет записан адрес  //предыдущего обработчика исключения. Объявление типа SEHHandler  // было приведено выше.  static SEHHandler oldHandler;  static void zHookUpSEHChain(SEHHandler handler);  static int myHandler(PEXCEPTION\_RECORD pEhRecors, PEXCEPTION\_REGISTRATION pEhRegRecord, PCONTEXT pContext, void\* pp);  public:  CatUnhandledExceptionFilter();  ~CatUnhandledExceptionFilter();  }; |

static void zHookUpSEHChain(SEHHandler handler); – это функция для подмены обработчика исключений runtime-библиотеки. Код ее почти не отличается от предложенного ранее. Единственным изменением является переменная, в которой сохраняется адрес предыдущего обработчика.

static int myHandler(PEXCEPTION\_RECORD pEhRecors, PEXCEPTION\_REGISTRATION pEhRegRecord, PCONTEXT pContext, PEXCEPTION\_RECORD pp); - это наш обработчик, который будет вызван в случае возникновения необработанного исключения.

|  |
| --- |
| int CatUnhandledExceptionFilter::myHandler(PEXCEPTION\_RECORD pEhRecors,  PEXCEPTION\_REGISTRATION pEhRegRecord, PCONTEXT pContext, void\* pp)  {  printf("\*\*\* In My Handler \*\*\*\n");  printf("Exception address: 0x%X\n", pEhRecors->ExceptionAddress);  printf("Exception code: 0x%X\n", pEhRecors->ExceptionCode);  return CatUnhandledExceptionFilter::oldHandler(pEhRecors, pEhRegRecord, pContext, pp);  } |

В программе создается статический объект типа CatUnhandledExceptionFilter. Во время создания этого объекта в конструкторе вызывается функция подмены самого верхнего обработчика исключений. После того, как статические объекты приложения созданы, runtime передает управление функции main, в которой генерируется исключение по доступу к памяти, в результате чего управление переходит нашему обработчику исключения, который сейчас не делает ничего, а просто выводит информацию об исключении на экран и передает управление подмененному обработчику.

После возникновения необработанного исключения операционная система вызывает наш обработчик для обработки исключения и он, выполнив то, что ему нужно, передет управление дальше.

Недостатком приведенного выше кода является то, что он работоспособен только в однопоточных приложениях. Все это происходит потому, что указатель на вершину цепочки EXCEPTION\_REGISTRATION находится в структуре TIB, которая хранит в себе информацию о текущем потоке, а значит, при вставке обработчика исключения с использованием значение FS:[0] мы установим обработчик только для одного потока.

Но что же делать в случае многопоточного приложения? Для этого, видимо, нужно проделать описанные действия для каждого потока. К счастью, этого делать не придется. В следующем разделе вы увидите, как обработать необработанные исключения во всех потоках.

**Фильтр необработанных исключений приложения**

Как уже упоминалось раньше, если ОС не нашла подходящего обработчика исключений, она вызывает функцию UnhandledExceptionFilter. Эта функция находится в kernel32.dll и имеется во всех версиях Windows. В заголовочных файлах SDK она объявлена следующим образом:

|  |
| --- |
| LONG UnhandledExceptionFilter(\_EXCEPTION\_POINTERS \*ExceptionInfo); |

где ExceptionInfo – это указатель на структуру, которая описывает исключение и содержимое регистров во время возникновения исключения.

Назначение этой функции – показать диалоговое окно, сообщающее, что произошло необработанное исключение, и, в зависимости от того, находится приложение под отладкой или нет, передать управление отладчику или завершить поток. Еще одной функцией этого фильтра является вызов пользовательского обработчика необработанных исключений, который может быть установлен функцией SetUnhandledExceptionFilter.

Я сознательно не упоминал о функции SetUnhandledExceptionFilter, потому что ее использование имеет ряд важных недостатков, перекрывающих преимущества, предоставляемых ее использованием. Чтобы внести ясность, давайте рассмотрим ее код:

|  |
| --- |
| SetUnhandledExceptionFilter:  77E7E5A1 mov ecx,dword ptr [esp+4]  77E7E5A5 mov eax,dword ptr ds:[77ED73B4h]  77E7E5AA mov dword ptr ds:[77ED73B4h],ecx  77E7E5B0 ret 4 |

Как видно из кода, функция берет адрес, переданный ей в качестве параметра, и помещает его в системную область памяти, возвращая предыдущее значение. Т.е. она замещает установленный ранее обработчик новым. Такое поведение говорит, что использование этой функции может привести к тому, что обработчик может быть переустановлен кем угодно.

Вторым важным недостатком использования этой функции является невозможность отладки обработчика, поскольку функция UnhandledExceptionFilter передает управление отладчику до вызова пользовательского обработчика, мешая нормальной отладке.

Если вы согласны с перечисленными недостатками использования функции SetUnhandledExceptionFilter, то давайте подумаем, как получить управление во время возникновения необработанного исключения. Если вы не согласны, то можете пропустить следующий раздел и перейти непосредственно к разделу «Сбор и сохранение информации о состоянии процесса».

**Получение управления**

До сих пор речь шла о механизме обработки исключений, используемом в ОС Windows, и о том, как он используется компиляторами. Реализованный выше простой алгоритм подменяет установленный компилятором обработчик, что позволяет получить управление при возникновении в приложении необработанного исключения. Но алгоритм был рассчитан на однопоточное приложение, а это значительно сужает возможности его использования. Кроме всего прочего, если в приложении возникает необработанное исключение, то установленный Runtime-библиотекой обработчик вызывает функцию UnhandledExceptionFilter, которая также может вызваться операционной системой. Таким образом, функция UnhandledExceptionFilter является ключевой системной функцией, которая обрабатывает все необработанные исключения. Подмена этой функции позволила бы выполнить поставленную задачу.

Поскольку функция UnhandledExceptionFilter может быть вызвана как из обработчика Runtime, так и системой, то изменение ее адреса в таблице импортируемых функций может не решить всей проблемы, поэтому для большей надежности нужно изменить код функции UnhandledExceptionFilter так, что бы сразу после ее вызова управление попадало к нам. К статье прилагается проект ehfilter, в котором реализован класс CatUnhandledExceptionFilter, устанавливающий обработчик необработанных исключений. Объявление класса CatUnhandledExceptionFilter приведено ниже:

|  |
| --- |
| class CatUnhandledExceptionFilter  {  private:  friend LONG myUnhandledExceptionFilter( \_EXCEPTION\_POINTERS\* ExceptionInfo );  UINT\_PTR m\_oldSystemUnhandledFilter;  LONG UnhandledExceptionFilter(\_EXCEPTION\_POINTERS\* ExceptionInfo);  public:  CatUnhandledExceptionFilter();  ~CatUnhandledExceptionFilter();  ool HookUpUnhandledFilter();  }; |

где

m\_oldSystemUnhandledFilter – адрес оригинальной функции UnhandledExceptionFilter.

myUnhandledExceptionFilter – дружественная классу функция-переходник, ее назначение и код будут рассмотрены ниже.

UnhandledExceptionFilter – наш фильтр необработанных исключений

HookUpUnhandledFilter – функция установки нашего фильтра исключений.

Проект ehfilter является обычной DLL, которая должна быть загружена в адресное пространство приложения. Во время загрузки библиотеки в файле main.cpp создается глобальная переменная gFeedBackFilter типа CatUnhandledExceptionFilter. Во время создания этой переменной в конструкторе определяется адрес функции UnhandledExceptionFilter и запоминается в переменной m\_oldSystemUnhandledFilter. Когда в библиотеку приходит сообщение DLL\_PROCESS\_ATTACH, вызывается функция HookUpUnhandledFilter, которая устанавливает наш фильтр необработанных исключений.

Код функции HookUpUnhandledFilter приведен ниже:

|  |
| --- |
| bool CatUnhandledExceptionFilter::HookUpUnhandledFilter()  {  if ( m\_oldSystemUnhandledFilter == 0 )  return false;  DWORD addr = m\_oldSystemUnhandledFilter;  DWORD old = 0;  if ( TRUE == VirtualProtect((LPVOID)addr, 5, PAGE\_READWRITE, &old) )  {  unsigned char \*p = (unsigned char\*)addr;  \*p = 0xE9;  UINT\_PTR ehFilter = (UINT\_PTR)myUnhandledExceptionFilter;  addr += 5;  ehFilter = ehFilter - addr;  p++;  DWORD \*pp = (DWORD\*)p;  \*pp = ehFilter;  m\_oldSystemUnhandledFilter += 5;  VirtualProtect((LPVOID)addr, 5, old, &old);  return true;  }  return false;  } |
| ПРИМЕЧАНИЕ  Этот код будет работать только в семействе ОС Windows NT. Дело в том, что 2 верхних гигабайта, где размещены системные библиотеки, в Windows 9х недоступны на запись из пользовательского (user) режима. – прим.ред. | | |

Сначала функция проверяет, был ли найден системный обработчик UnhandledExceptionFilter. Если он не найден, функция возвращает false и завершает свою работу. Затем, поскольку необходимо писать в системную область памяти, изменяются атрибуты доступа к ней, что делает ее доступной для чтений/записи, и записывается инструкция безусловного перехода на функцию-переходник myUnhandledExceptionFilter. Функция-переходник имеет две цели – это вызвать фильтр и вернуть управление системной функции.

Приведенные в примере реализации функций HookUpUnhandledFilter и myUnhandledExceptionFilter являются сильно упрощенными и, конечно, неприменимы в реальной жизни. Мало того, попытка их применения может привести к печальным последствиям. Однако они достаточны для иллюстрации механизма подмены вызова системных функций. В реальности же необходимо дизассемблировать код функции UnhandledExceptionFilter, запоминать его и затем использовать при возврате управления. Но это в значительной мере усложнило бы код и могло скрыть основной его смысл, поэтому я решил оставить эту реализацию для демонстрации самого факта возможности подобных действий.

**Сбор и сохранение информации о состоянии процесса**

Как было упомянуто выше, функция UnhandledExceptionFilter принимает один аргумент, являющийся указателем на \_EXCEPTION\_POINTERS. В этой структуре сохранена информация о состоянии приложения после возникновения исключения, состояние регистров, информация об исключении. Но этого может быть мало для восстановления реальной картины произошедшего. Поэтому в своем обработчике я попытаюсь собрать ту информацию, которая поможет понять причину сбоя. Объем этой информации может зависеть от типа приложения и нельзя привести весь список необходимых параметров, способных помочь при анализе причины падения приложения. Программист должен сам решить, какой объем и тип информации ему требуется. Здесь я хочу лишь сказать, что информации, которую собирает о приложении библиотека dbghelp.dll, поставляемая с Windows, достаточно для большинства приложений. dbghelp.dll сохраняет следующее:

- информация о потоках;

- информация о загруженных модулях;

- информация об исключении;

- информация о системе;

- информация о памяти.

После сбора информации ее можно сохранить в файле на диске для последующего анализа и разбора.

**Заключение**

Операционная система Windows предоставляет каждому разработчику уникальную возможность обрабатывать исключения, возникающие в его программах, с помощью механизма Структурированной Обработки Исключений (SEH). Но приложения не полностью используют возможности, предоставляемые этим системным сервисом, и оставляют часть ошибок необработанными. В статье я привел пример того, как получить управление в случае фатальной ошибки приложения, и дал приложению получить управление вместо того, чтобы быть просто выгруженным. Этот метод может дать приложению последний шанс:

сохранить данные;

собрать информацию о состоянии на момент ошибки;

хотя бы извиниться....