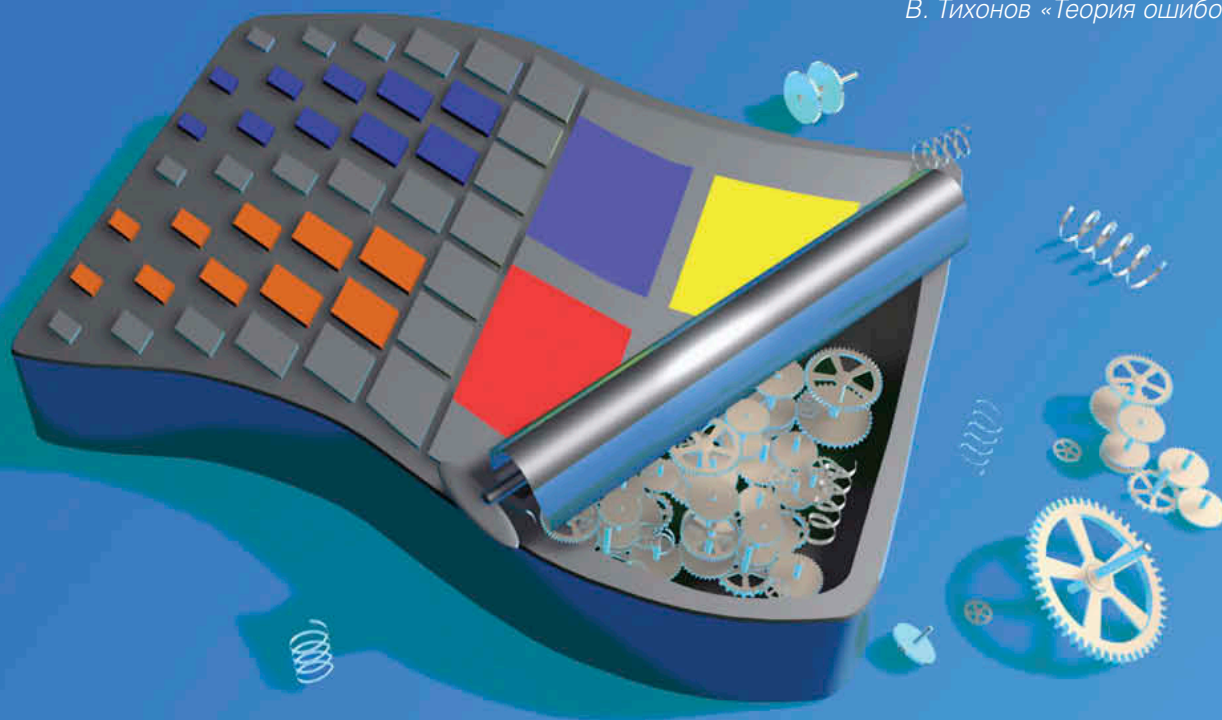


ПРАКТИЧЕСКИЕ СОВЕТЫ ПО ВОССТАНОВЛЕНИЮ СИСТЕМЫ В БОЕВЫХ УСЛОВИЯХ

1. Во время исполнения ошибки имеют наивысший приоритет. Прервать исполнение ошибки может только другая, более активная ошибка.
2. Запросы операционной системы к ошибкам ошибками могут игнорироваться.
3. Запросы ошибок к операционной системе игнорироваться не могут.
4. При работе с файлами ошибки могут пользоваться файловой системой базовой ОС и ее ошибками.
5. На ЭВМ с параллельной архитектурой может выполняться несколько ошибок одновременно.

В. Тихонов «Теория ошибок»



Практически всем администраторам приходилось сталкиваться с теми или иными сбоями ОС и ее окружения, но далеко не все могли быстро найти источник их возникновения (особенно если сбой происходит не регулярно и на чужой машине). Тем не менее существует несколько вполне универсальных стратегий поиска дефективных компонентов, разработанных и апробированных еще со времен ЕС и мейнфреймов. Вот о них-то и рассказывает настоящая статья.

КРИС КАСПЕРСКИ

Типичная реакция домашнего пользователя на нестабильность работы своей машины – полная переустановка операционной системы. Иногда это помогает, иногда – нет, но, как бы там ни было, переустановка операционной системы на сервере – достаточно грандиозное событие, самое малое на целый день выводящее локальную сеть фирмы из игры. Квалифицированный администратор отличается от неквалифицированного в первую очередь тем, что со всеми проблемами справляется на лету, до минимума сводя время простоя сети.

Вообще-то хорошо отлаженная система, базирующаяся на ОС типа FreeBSD (или подобной ей), способна без сбоев работать годами, не требуя к себе совершенно никакого внимания. Системы, построенные на базе Windows NT, этим, увы, похвастаться не могут, и для достижения сколь-нибудь стабильной работы за ними приходится постоянно ухаживать.

Аппаратное обеспечение, собираемое на коленках в ближайшем подвале, также не отличается высокой надежностью, а отличить качественную подделку от оригинала по внешним признакам достаточно трудно. На просторах России свободно продаются отбракованные чипы, левым путем добытые у производителей и выдаваемые за настоящие. Кстати, многие из именитых производителей грешат передачей своих торговых марок третьим фирмам, выпускающим довольно посредственное оборудование, но продающих его по «брендовским» ценам. Яркий тому пример – пишущий привод TEAC 552E, к которому фирма TEAC вообще не имеет никакого отношения. Про материнские платы и модули памяти вообще говорить не стоит. Их клепают все кому не лень, и многие модели вообще не работают, кое-как запускаясь на пониженных таймингах и частотах.

Словом, если сбой старушки БЭСМ-6 был настоящим ЧП, то зависание современного сервера – вполне обычное дело, воспринимаемое администраторами как неизбежное зло. Эта статья не уберет вас ни от критических ошибок приложений, ни от отказа оборудования, но, по крайней мере, научит быстро и безошибочно находить их источник. Речь пойдет преимущественно о Windows NT и производных от нее системах (Windows 2000, Windows XP), хотя поклонники UNIX также найдут здесь немало интересного.

Аппаратная часть

Вот два основных аппаратных виновника нестабильной работы системы – оперативная память и блок питания. Рассмотрим их поподробнее, отмечая особенности взаимодействия с памятью в современных чипсетах, таких как Intel 875P и подобных ему.

Тесная связь между программным и аппаратным обеспечением затрудняет деление статьи на две равные части, поскольку ряд сбоев системы (и пресловутых голубых экранов смерти – в том числе) вызван отнюдь не алгоритмическими ошибками, а неисправностью железа. Но на начальном этапе анализа «голубого экрана смерти» (далее по тексту просто голубого экрана) мы не можем надежно установить его источник и, чтобы не описывать одни и те же методики дважды, условимся относить все критические ошибки системы к программной среде. В действительности же это не вызывает никакого противоре-

чия, поскольку с аппаратными ошибками приходится бороться и программными средствами (помните известное: «как нематериальная душа возвращается в тело в результате материальных действий врача?»).

Оперативная память

Оперативная память относится к одному из наименее надежных компонентов вычислительной системы, и потому львиная доля всех сбоев приходится именно на нее. Проявления их могут быть самыми разнообразными: от критических ошибок приложений до периодических или непериодических ошибок чтения (записи) на жесткий диск или даже каскадных ошибок приема/передачи TCP/IP-пакетов (что не покажется удивительным, если вспомнить о кэширующем приводе всех драйверов, обслуживающих устройства ввода/вывода). Любой аппаратный ресурс, требующий для своей работы некоторого количества оперативной памяти, так или иначе зависит от работоспособности последней.

Существует мнение, что память «с четностью» полностью решает проблему своей надежности и сводит риск разрушения данных к разумному минимуму. На самом деле это не так. Память с четностью распознает лишь одиночные ошибки и не гарантирует обнаружение групповых. Память типа ECC (Error Check & Correction/Error Correction Code – Контроль и Исправление Ошибок), способна автоматически исправлять любые одиночные ошибки и обнаруживать любые двойные. До тех пор, пока оперативная память функционирует более или менее нормально, противостояние энтропии и помехозащитных кодов решается в пользу последних. Однако при полном или частичном выходе одного или нескольких модулей памяти из строя корректирующих способностей контролирующих кодов перестает хватать, и система начинает работать крайне нестабильно.

Концепция виртуальной памяти, реализованная в операционных системах семейства Windows и UNIX, рассматривает основную оперативную память как своеобразный кэш. А это значит, что одни и те же логические страницы адресного пространства в разное время могут отображаться на различные физические адреса. Разрушение одной-единственной физической ячейки памяти затрагивает множество виртуальных ячеек, и потому сбои памяти практически всегда проявляются «коллективными» критическими ошибками приложений, рассредоточенными в широком диапазоне адресов. Если же критические ошибки возникают лишь в некоторых процессах и располагаются по более или менее постоянным адресам – с высокой степенью вероятности можно предположить, что это программная, а не аппаратная ошибка. Исключение составляет неоткачиваемая область памяти (non-paged pool), занятая ядром системы и всегда размещающаяся по одним и тем же физическим адресам. Наличие дефективных ячеек в данной области обычно приводит к синему экрану смерти и/или полному зависанию системы, хотя в некоторых случаях ошибки драйверов передаются на прикладной уровень и роняют один или несколько процессов.

Самое интересное, что при прогоне нестабильно работающих драйверов/процессом под отладчиком ошибка волшебным образом может исчезать. В действительности ничего загадочного тут нет. За счет многократного снижения

интенсивности доступа к памяти отладчик позволяет «вытянуть» даже дефективные ячейки, затрудняя их локализацию. Некоторые руководства рекомендуют исследовать дампы, сброшенные системой при возникновении критической ошибки в ядре системы, наивно надеясь на то, что искаженные ячейки будут выглядеть как бессмысленный мусор, сразу бросающийся в глаза даже при минимальных навыках дизассемблирования. При разрушении большого количества ячеек памяти, затрагивающих исполняемый код, это действительно так. Однако искажение областей данных предложенный алгоритм выявить не в состоянии. Только чрезвычайно опытный разработчик драйверов заподозрит, что здесь что-то не так. А ведь в некоторых случаях неисправный модуль содержит всего лишь один-единственный дефективный бит информации, который при визуальном осмотре дампа вообще нереально обнаружить. К тому же не стоит забывать, что «замусоривание» памяти может быть вызвано не только аппаратными, но и программными ошибками (например, программист забыл проинициализировать буфера или направил указатели в «космос», передав управление по произвольному адресу памяти).

Худший случай – это разрушение буферов ввода/вывода, зачастую приводящее к полному краху файловой системы без какой-либо надежды на ее восстановление. По непонятной причине разработчики дисковых драйверов отказались от подсчета контрольной суммы пересылаемых через них блоков данных, что сделало файловую систему чрезвычайно уязвимой. Причем NTFS оказывается даже в худшей ситуации, чем FAT32, поскольку требует значительно меньшего объема буферной памяти для своей поддержки и к тому же значительно легче поддается «ручному» восстановлению. Автор настоящей статьи использует отказоустойчивые буфера, построенные на основе демонстрационных драйверов, входящих в состав DDK, и дополненные специальными средствами контроля. Главное ноу-хау данной технологии состоит в том, что обмен с диском ведется на «сыром» (RAW MODE) уровне, т.е. помимо области пользовательских данных в сектор входит контрольная сумма, по которой драйвер с одной и привод с другой стороны контролируют целостность данных. В жизни автора эта технология срабатывала дважды (т.е. выявляла дефективный модуль памяти, пытавшийся разрушить жесткий диск), так что усилия, затраченные на разработку драйверов, многократно окупили себя сполна!

Кстати, тестирование оперативной памяти путем прогона специальных программ (Check It, PC Diagnostic и им подобных) – не самый лучший путь для выявления ее работоспособности. В силу физической неоднородности подсистемы памяти дефективность бракованных модулей зачастую проявляется не на любой, а на строго определенной последовательности запросов и при определенном сочетании содержимого разрушенной и окрестных ячеек. Тестирующие программы перебирают ограниченное количество наиболее типичных шаблонов и потому обнаруживают лишь некоторые, наиболее дефективные дефекты. Ряд серверных чипсетов содержит в себе более или менее продвинутые средства тестирования памяти, работающие в фоновом режиме и работающие достаточно эффективно.

Ряд тестовых пакетов, таких, например, как TestMem от SERJ_M, перебирают большое количество разнотипных шаблонов и довольно лихо выявляют скрытые дефекты модулей памяти, в обычной жизни проявляющиеся лишь при стечении множества маловероятных обстоятельств. К сожалению, эволюция чипсетов в конце концов привела к тому, что и эти шаблоны перестали работать. При слишком интенсивном обмене с памятью чипсет Intel 875P и другие подобные ему чипсеты начинают вставлять холостые циклы, давая памяти время «остыть», предотвращая тем самым ее перегрев. С одной стороны, такое конструкторское решение можно только приветствовать, поскольку оно значительно повышает надежность системы, но с другой – чрезвычайно затрудняет ее тестирование. Для получения сколь-нибудь достоверных результатов тестирующая программа должна подобрать такую интенсивность прогона памяти, при которой холостые циклы еще не вставляются, но система работает уже на пределе. Насколько известно автору, подобных программ еще нет и когда они появятся на рынке – неизвестно. Так что спасение утопающих – забота самих утопающих.

Сама по себе память, может быть, и не виновата. Источником ошибок вполне может быть и северный мост чипсета, содержащий контроллер памяти. Исследуя чипсет VIA KT133, автор обнаружил несколько критических ошибок планировщика очередей, приводящих к искажению передаваемых данных и визуально проявляющихся как типичные дефекты памяти.

Блок питания

Второй по распространенности источник нестабильной работы компьютера – это блок питания. Современные компьютеры предъявляют к качеству питающего напряжения достаточно жесткие требования, при нарушении которых работа компьютера становится совершенно непредсказуемой, проявляясь зависаниями, критическими ошибками и голубыми экранами смерти, выскакивающими в самых неожиданных местах. В ряде случаев отмечается замедление быстрогодействия приводов, обычно носящее характер внезапных провалов производительности (копирование файлов движется как бы рывками).

Практически все уважающие себя производители материнских плат оснащают свои детища развитой электронной системой контроля основных (опорных) напряжений, показания которых отображаются специальными утилитами. Убедитесь, что питающий потенциал соответствует норме, отклонясь от нее не более чем на 5-10%, и остается более или менее постоянным в процессе работы компьютера. Причем «недобор» напряжения намного более опасен, чем «перебор». Увеличение потенциала на 15-20% практически никогда не приводит к моментальному выходу электроники из строя, правда, вызывает ее перегрев, но при наличии качественной системы охлаждения с этим можно и смириться. Но даже незначительное уменьшение потенциала заметно снижает реакционность переходных процессов полупроводниковых элементов, и система не успевает поспевать за тактовой частотой, что приводит к зависаниям, критическим ошибкам, перезагрузкам и т. д.

На рисунке 1 приведен плохой блок питания, обнаруживающий значительную просадку на линии 12 вольт и чудовищные пульсации напряжения. Линия 3.3 вольт, обслуживающая святую святых – оперативную память, также слегка пульсирует, хотя стабилизируется отнюдь не блоком питания, но самой материнской платой, предел стабилизации которой, впрочем, тоже небезграничен, и даже качественная материнская плата бессильна выправить кривой от рождения блок питания.

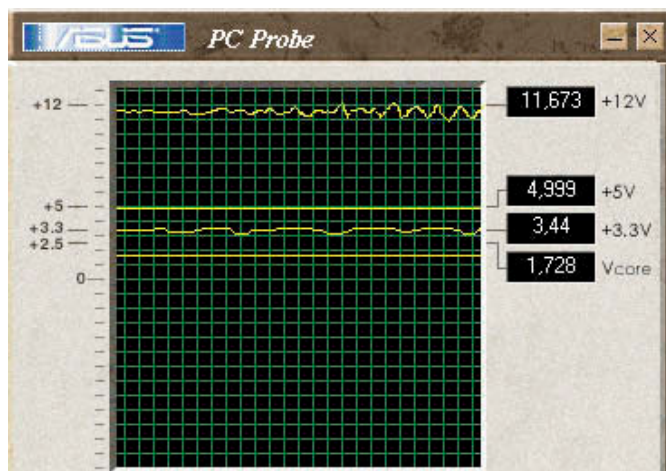


Рисунок 1. Пример плохого блока питания

К сожалению, точность интегрированных вольтметров достаточно невелика, и многие из них явно нуждаются в хорошей калибровке. Поэтому доверять таким показаниям следует с осторожностью и большой долей скептицизма, при необходимости уточняя их нормальным цифровым мультиметром.

...и все-все-все

Остальные компоненты компьютера практически никогда не вызывают серьезных проблем. Процессоры (при надлежащей системе охлаждения и не слишком большой тактовой частоте, конечно) лишь в исключительных случаях позволяют себе подвесить систему (да и то основная доля вины ложится не на сам процессор, а на интегрированный кэш). Кстати, характерная болезнь «разогнанных процессоров» – голубой экран с надписью «UNEXPECTED_KERNEL_MODE_TRAP» (подробнее об этом рассказывается во второй части статьи).

Жесткие диски, становясь все более и более интеллектуальными устройствами, достаточно неприхотливы, правда, при неправильной установке терминаторов на SCSI-устройствах Windows NT может выбрасывать голубой экран смерти, но хорошие диски терминируют себя самостоятельно.

Карты расширения от сторонних производителей, будучи расположенными на разделяемой PCI-шине, способны вызывать любые мыслимые и немыслимые конфликты, поэтому не пользуйтесь продукцией тех поставщиков, которым вы не доверяете.

Программная часть

Катастрофическая небрежность тестирования фирменного и кустарного ПО приводит к появлению многочисленных критических ошибок при его исполнении: «Программа выполнила недопустимую операцию и будет закрыта.

Если ошибка будет повторяться, обратитесь к разработчику». К несчастью, критические ошибки приложения (в терминологии Windows 2000 просто «ошибки приложения») имеют устойчивую тенденцию появляться в самые ответственные моменты времени, например, накануне сдачи финансового отчета. А разработчики... они в большинстве своем такие сообщения просто игнорируют. Иногда потому, что просто не знают, как эту информацию интерпретировать, иногда потому, что вообще не заботятся о проблемах своих пользователей.

Многие сетуют на тупость Windows и ее неспособность противостоять критическим ошибкам. Но эти обвинения совершенно безосновательны. Возникновение критической ошибки свидетельствует о том, что программа поехала крышей и пошла вразнос. Все, что только операционная система может сделать, – это пристрелить ее, в противном случае программа возвратит заведомо неверные данные, чего допускать ни в коем случае нельзя. Так что операционную систему не ругать следует, а благодарить!

Иногда сообщения о критических ошибках удается предотвратить установкой нового сервис-пака, а иногда, наоборот, – путем удаления нового. Еще можно попробовать переустановить операционную систему или только само нестабильно работающее приложение. Однако никаких гарантий, что после всех этих манипуляций сбой действительно у вас исчезнет, нет. Достаточно вспомнить нашу мемуарную историю с червем MSBLASTER, вызывающим критическую ошибку в системном сервисе svchost. Но сколько бы вы ни переустанавливали свою Windows 2000, сколько бы ни меняли железо, ситуация не улучшалась. Антивирусы, правда, сообщали о наличии вируса на компьютере (да и то не всегда), однако не объясняли, какие меры безопасности следует принять. К тому же обрушение сервиса svchost происходило отнюдь не вследствие инфицирования компьютера вирусом, а лишь при неудачной попытке его. Именно неумение хакеров сорвать стек, не уронив при этом всю систему, и демаскировало вирус, попутно организовав разрушительную DoS-атаку, до сих пор приносящую весьма ощутимые убытки.

Всякий администратор, считающий себя профессионалом, не может позволить себе роскошь действовать вслепую. Знание ассемблера и умение быстро и грамотно интерпретировать сообщения о критических ошибках, если еще не решит проблему, то, по крайней мере, придаст вам чувство уверенности и поможет локализовать истинного виновника нестабильности системы. Во всяком случае будет куда ткнуть носом зарвавшегося разработчика. Согласитесь, одно дело догадываться об ошибке и совсем другое – показывать на нее пальцем.

Приложения, недопустимые операции и все-все-все

Различные операционные системы по-разному реагируют на критические ошибки. Так, например, Windows NT резервирует два региона своего адресного пространства для выявления некорректных указателей. Один находит-

ся на самом «дне» карты памяти и предназначен для отлавливания нулевых указателей. Другой расположен между «кучей» и областью памяти, закрепленной за операционной системой. Он контролирует выход за пределы пользовательской области памяти и, вопреки расхожему мнению, никак не связан в функции WriteProcessMemory (см. техническую заметку ID: Q92764 в MSDN). Оба региона занимают по 64 Кб, и всякая попытка доступа к ним расценивается системой как критическая ошибка. В Windows 9x имеется всего лишь один 4 Кб регион, следящий за нулевыми указателями, поэтому по своим контролирующим способностям она значительно уступает NT.

В Windows NT экран критической ошибки (см. рис. 2) содержит следующую информацию:

- адрес машинной инструкции, возбудившей исключение;
- словесное описание категории исключения (или его код, если категория исключения неизвестна);
- параметры исключения (адрес недействительной ячейки памяти, род операции и т. д.).

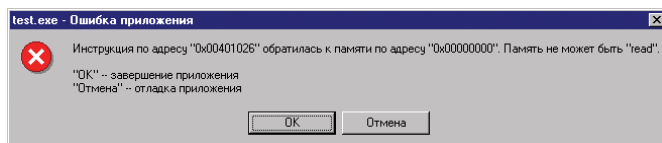


Рисунок 2. Сообщение о критической ошибке, выдаваемое операционной системой Windows 2000

Операционные системы семейства Windows 9x в этом отношении намного более информативны (см. рис. 3) и помимо категории исключения выводят содержимое регистров ЦП на момент сбоя, состояние стека и байты памяти по адресу CS:EIP (т.е. текущему адресу исполнения). Впрочем, наличие «Доктора Ватсона» (о нем – далее) стирает различие между двумя системами, и потому можно говорить лишь об удобстве и эргономике 9x, сразу предоставляющей весь минимум необходимых сведений, в то время как в NT отчет об ошибке создается отдельной утилитой.

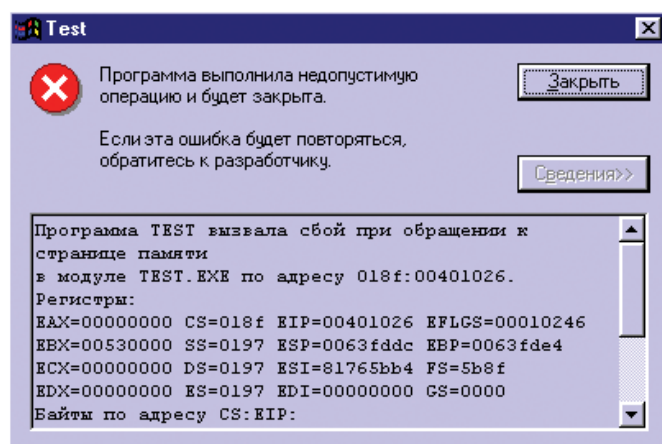


Рисунок 3. Сообщение о критической ошибке, выдаваемое операционной системой Windows 98

Если никакой из отладчиков в системе не установлен, то окно о критической ошибке имеет всего лишь одну кнопку – кнопку «ОК», нажатие которой приводит к аварийному закрытию политекорректного приложения. При желании окно критической ошибки можно оснастить кнопкой «Отмена» («Cancel»), запускающей отладчик или иную утилиту анализа ситуации. Важно понять, что «Отмена» отнюдь не отменяет автоматическое закрытие приложения, но при некоторой сноровке вы можете устранить «пробоину» вручную, продолжив нормальную работу¹.

Доктор Ватсон

«Доктор Ватсон» является штатным обработчиком критических ошибок, входящим в базовый пакет поставки всех операционных систем семейства Windows. По своей природе он представляет статическое средство сбора релевантной информации. Предоставляя исчерпывающий отчет о причинах сбоя, «Доктор Ватсон» в то же самое время лишен активных средств воздействия на некорректно работающие программы. Утихомирить разбушевавшееся приложение, заставив его продолжить свою работу с помощью одного «Доктора Ватсона», вы не сможете, и для этого вам придется прибегать к интерактивным отладчикам, одним из которых является Microsoft Visual Studio Debugger, входящий в состав одноименной среды разработки и рассматриваемый в статье далее.

Считается, что «Доктор Ватсон» предпочтительнее использовать на рабочих станциях (точнее, на автоматизированных рабочих местах), а интерактивные средства отладки – на серверах. Дескать, во всех премудростях ассемблера пользователи все равно не разбираются, а вот на сервере продвинутый отладчик будет как нельзя кстати. Отчасти это действительно так, но не стоит игнорировать то обстоятельство, что далеко не все источники ошибок обнаруживаются статическими средствами анализа, к тому же интерактивные инструменты значительно упрощают процедуру анализа. С другой стороны, «Доктор Ватсон» достается нам даром, а все остальные программные пакеты приходится приобретать за дополнительную плату. Так что предпочтительный обработчик критических ошибок вы должны выбирать сами.

Для установки «Доктора Ватсона» отладчиком по умолчанию добавьте в реестр следующую запись или запустите файл Drwtsn32.exe с ключом «-i» (для выполнения обоих действий вы должны иметь права администратора):

Листинг 1. Установка «Доктора Ватсона» отладчиком по умолчанию

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\
CurrentVersion\AeDebug]
"Auto"="1"
"Debugger"="drwtsn32 -p %ld -e %ld -g"
"UserDebuggerHotKey"=dword:00000000
```

¹ Запустите «Редактор Реестра» и перейдите в раздел «HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug». Если такого раздела нет – создайте его самостоятельно. Строковой параметр «Debugger» задает путь к файлу отладчика со всеми необходимыми ключами; строковой параметр «Auto» указывает, должен ли отладчик запускаться автоматически (значение «1») или предлагать пользователю свободу выбора («0»). Наконец двойное слово параметра «UserDebuggerHotKey» специфицирует скэн-код горячей клавиши для принудительного вызова отладчика.

Теперь возникновение критических ошибок программы станет сопровождаться генерацией отчета, составленного «Доктором Ватсоном» и содержащим более или менее подробные сведения о характере ее происхождения.

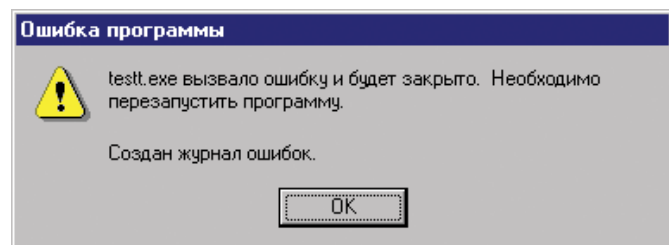


Рисунок 4. Реакция «Доктора Ватсона» на критическую ошибку

Образец дампа, созданный «Доктором Ватсоном», приведен ниже. Комментарии, добавленные автором, выделены красным цветом:

Листинг 2. Образец отчета «Доктора Ватсона»

```
Исключение в приложении:
Прил.: (pid=612)
; pid процесса, в котором произошло исключение.

Время: 14.11.2003 @ 22:51:40.674
; Время, когда произошло исключение.

Номер: c0000005 (нарушение прав доступа)
; Код категории исключения.
; Расшифровку кодов исключений можно найти в WINNT.H
; входящим в состав SDK, прилагаемом к любому
; Windows-компилятору. Подробное описание всех исключений
; содержится в документации по процессорам Intel и AMD,
; бесплатно распространяемой их производителями (внимание:
; для перевода кода исключения операционной системы в
; вектор прерывания ЦП, вы должны обнулить старшее слово),
; в данном случае это 0x5 - попытка доступа к памяти
; по запрещенному адресу.

*----> Сведения о системе <----*
Имя компьютера: KPNC
Имя пользователя: Kris Kaspersky
Число процессоров: 1
Тип процессора: x86 Family 6 Model 8 Stepping 6
Версия Windows 2000: 5.0
Текущая сборка: 2195
Пакет обновления: None
Текущий тип: Uniprocessor Free
Зарегистрированная организация:
Зарегистрированный пользователь: Kris Kaspersky
; Краткие сведения о системе.

*----> Список задач <----*
0 Idle.exe
8 System.exe
232 smss.exe
...
1244 os2srv.exe
1164 os2ss.exe
1284 windbg.exe
1180 MSDDEV.exe
1312 cmd.exe
612 test.exe
1404 drwtsn32.exe
0 _Total.exe

(00400000 - 00406000)
(77F80000 - 77FFA000)
(77E80000 - 77F37000)
; Перечень загруженных DLL.
; Согласно документации, справа от адресов должны быть
; перечислены имена соответствующих модулей, однако
; практически все они так хорошо «замаскировались», что стали
; совершенно не видны. Вытащить их имена из файла протокола
; все-таки можно, но придется немного «пошаманить» (см. ниже
; «таблицу символов»).

Копия памяти для потока 0x188
; Ниже идет копия памяти потока, вызвавшего исключение.
```

```
eax=00000064 ebx=7ffdf000 ecx=00000000 edx=00000064 J
esi=00000000 edi=00000000
eip=00401014 esp=0012ff70 ebp=0012ffc0 iopl=0 J
nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000
epl=00000202
; Содержимое регистров и флагов.

функция: <nosymbols>
; Распечатка окрестной точки сбоя.

00400ffc 0000  add  [eax],al      ds:00000064=?
; Записываем в ячейку, на которую ссылается EAX, значение AL.
; Значение адреса ячейки, вычисленной Доктором Ватсоном,
; равно 64h, что, очевидно, не соответствует
; действительности. «Доктор Ватсон» подставляет в выражение
; значение регистра EAX на момент возникновения сбоя,
; и это совсем не то значение, которое было в момент
; исполнения! К сожалению, чему был равен EAX в момент
; исполнения ни нам, ни «Доктору Ватсону» не известен.

00400ffe 0000  add  [eax],al      ds:00000064=?
; Записываем в ячейку, на которую ссылается EAX, значение AL.
; Как? Опять? Вообще-то так кодируется последовательность
; 00 00 00 00, по всей видимости являющаяся осколком
; некоторой машинной команды, неправильно
; интерпретированной дизассемблерным движком
; «Доктора Ватсона».

00401000 8b542408 mov  edx,[esp+0x8]  ss:00f8d547=????????
; Загружаем в EDX аргумент функции.
; Какой именно аргумент - сказать невозможно, т.к. мы не
; знаем адрес стекового фрейма.

00401004 33c9      xor  ecx,ecx
; Обнуляем ECX

00401006 85d2      test edx,edx
00401008 7e18      jle  00409b22
; Если EDX == 0, прыгаем на адрес 409B22h.

0040100a 8b442408 mov  eax,[esp+0x8]  ss:00f8d547=????????
; Загружаем уже упомянутый аргумент в регистр EAX.

0040100e 56        push esi
; Сохраняем ESI в стеке, перемещая тем самым указатель
; вершины стека на 4 байта вверх (в область младших
; адресов).

0040100f 8b742408 mov  esi,[esp+0x8]  ss:00f8d547=????????
; Загружаем в ESI очередной аргумент.
; Поскольку ESP был только что изменен, это совсем не тот
; аргумент, с которым мы имели дело ранее.

00401013 57        push edi
; Сохраняем регистр EDI в стеке.

СВОЙ -> 00401014 0fb3c31 movsx edi,byte ptr [ecx+esi] J
ds:00000000=?
; Вот мы и добрались до инструкции, возбудившей исключение
; доступа. Она обращается к ячейке памяти, на которую
; указывает сумма регистров ECX и ESI, а чему равно их
; значение? Прокручиваем экран немного вверх и находим, что
; ECX и ESI равны 0, о чем «Доктор Ватсон» нам и сообщает:
; «ds:000000» отметим, что этой информации можно верить,
; поскольку подстановка эффективного адреса осуществлялась
; непосредственно в момент исполнения, теперь вспомним,
; что ESI содержит копию переданного функции аргумента
; и что ECX был обнулен явно, следовательно, в выражении
; [ECX+ESI] регистр ESI - указатель, а ECX - индекс. Раз ESI
; равен нулю, то нашей функции передали указатель
; на невыделенную область памяти. Обычно это происходит
; либо вследствие алгоритмической ошибки в программе,
; либо вследствие истощения виртуальной памяти,
; к сожалению, «Доктор Ватсон» не осуществляет
; дизассемблирование материнской функции, и какой из двух
; предполагаемых вариантов правильный - нам остается лишь
; гадать... Правда, можно дизассемблировать дампы памяти
; процесса (если, конечно, он был сохранен), но это уже
; не то...

00401018 03c7      add  eax,edi
; Сложить содержимое регистра EAX с регистром EDI
; и записать результат в EAX.

0040101a 41        inc  ecx
; Увеличить ECX на единицу.
```



```
0040101b 3bca    cmp    ecx,edx
0040101d 7cf5    jl     00407014
; До тех пор пока ECX < EDX, переходить на адрес 407014
; (очевидно, мы имеем дело с циклом, управляемым счетчиком
; ECX). При интерактивной отладке мы могли бы принудительно
; выйти из функции, возвратив флаг ошибки, чтобы материнская
; функция (а с ней и вся программа целиком) могла продолжить
; свое выполнение, и в этом случае потерянной окажется лишь
; последняя операция, но все остальные данные окажутся
; неискаженными.
```

```
0040101f 5f      pop    edi
00401020 5e      pop    esi
00401021 c3      ret
; Выходим из функции.
```

```
*----> Обратная трассировка стека <----*
; Содержимое стека на момент возникновения сбоя.
; Распечатает адреса и параметры предыдущих выполняемых функций,
; при интерактивной отладке мы могли бы просто передать управление
; на одну из вышележащих функций, что эквивалентно возвращению
; в прошлое, это только в реальной жизни разбитую чашку
; восстановить нельзя, в компьютерной вселенной возможно все!
FramePtr ReturnAd Param#1 Param#2 Param#3 Param#4
Function Name
; FramePtr: Указывает на значение фрейма стека,
; выше (т.е. в более младших адресах)
; содержится аргументы функции, ниже - ее
; локальные переменные.
;
; ReturnAd: Бережно хранит адрес возврата в материнскую
; функцию. Если здесь содержится мусор и
; обратная трассировка стека начинается
; характерно шуметь, с высокой степенью
; вероятности можно предположить, что мы
; имеем дело с ошибкой «срыва стека», а
; возможно, и с попыткой атаки вашего
; компьютера.
;
; Param#: Четыре первых параметра функции - именно
; столько параметров «Доктор Ватсон» отображает
; на экране. Это достаточно жесткое ограничение -
; многие функции имеют десятки параметров и
; четыре параметра еще ни о чем не говорят;
; однако недостающие параметры легко вытащить
; из копии необработанного стека вручную -
; достаточно лишь перейти по указанному в поле
; FramePtr адресу.
;
; Func Name: Имя функции (если только его возможно
; легко вытащить и определить);
; реально отображает лишь имена функций,
; импортируемые из других DLL, поскольку
; встретить коммерческую программу,
; откомпилированную вместе с отладочной
; информацией практически нереально.
0012FFC0 77E87903 00000000 00000000 7FFDF000 C0000005 !<nosymbols>
0012FFC0 00000000 00401040 00000000 000000C8 00000100
kernel32!SetUnhandledExceptionFilter
; функции перечисляются в порядке их исполнения; самой последней
; исполнялась kernel32!SetUnhandledExceptionFilter функция,
; обрабатывающая данное исключение.
```

```
*----> Копия необработанного стека <----*
; Копия необработанного стека содержит стек таким, какой он есть.
; Очень помогает при обнаружении buffer overflow атак -
; весь shell-код, переданный злоумышленником, будет распечатан
; «Доктором Ватсоном», и вам останется всего лишь опознать его
; (подробнее об этом рассказывается в моей книге «Техника
; сетевых атак»)
0012ff70 00 00 00 00 00 00 00 00 - 39 10 40 00 00 00 00 00 .....9.@.....
0012ff80 64 00 00 00 f4 10 40 00 - 01 00 00 00 d0 0e 30 00 d.....@.....0.
...
00130090 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
001300a0 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
*----> Таблица символов <----*
; Таблица символов содержит имена всех загруженных DLL вместе
; с именами импортируемых функций. Используя эти адреса в
; качестве отправной точки, мы без труда сможем восстановить
; «перечень загруженных DLL».
```

```
ntdll.dll
77F81106 00000000 ZwAccessCheckByType
...
77FCEFB0 00000000 fltused
```

```
kernel32.dll
77E81765 0000003d IsDebuggerPresent
...
77EDBF7A 00000000 VerSetConditionMask
;
; Итак, возвращаемся к таблице загруженных DLL.
; (00400000 - 00406000) - это, очевидно, область памяти,
; занятая самой программой
; (77F80000 - 77FFA000) - это KERNEL32.DLL
; (77E80000 - 77F37000) - это NTDDLL.DLL
```

Microsoft Visual Studio Debugger

При установке среды разработки Microsoft Visual Studio она регистрирует свой отладчик основным отладчиком критических ошибок по умолчанию. Это простой в использовании, но функционально ущербный отладчик, не поддерживающий даже такой банальной операции, как поиск hex-последовательности в оперативной памяти. Единственная «вкусность», отличающая его от продвинутого во всех отношениях Microsoft Kernel Debugger – это возможность трассировки «упавших» процессов, выбросивших критическое исключение.

В опытных руках отладчик Microsoft Visual Studio Debugger способен творить настоящие чудеса, и одно из таких чудес – это возобновление работы приложений, совершивших недопустимую операцию и при нормальном течении событий, аварийно завершаемых операционной системой без сохранения данных. В любом случае интерактивный отладчик (коим Microsoft Visual Studio Debugger и является) предоставляет намного более подробную информацию о сбое и значительно упрощает процесс выявления источников его возникновения. К сожалению, тесные рамки журнальной статьи не позволяют изложить всю методику поиска неисправностей целиком, и приходится ограничиваться лишь узким кругом наиболее интересных (и наименее известных!) вопросов (см. раздел «Обитатели сумеречной зоны, или из морга в реанимацию»).

Для ручной установки Microsoft Visual Studio Debugger основным отладчиком критических ошибок добавьте в реестр следующие данные:

Листинг 3. Установка Microsoft Visual Studio Debugger основным отладчиком критических ошибок

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\
CurrentVersion\AeDebug]
"Auto"="1"
"Debugger"="\"C:\Prg Files\MS VS\Common\MSDev98\Bin\
msdev.exe\" -p %ld -e %ld"
"UserDebuggerHotKey"=dword:00000000
```

Листинг 4. Демонстрационная программа, вызывающая сообщение о критической ошибке

```
// функция возвращает сумму n символов типа char. Если ей
// передать null-pointer, она «упадет», хотя источник ошибки
// не в ней, а в аргументах, переданных материнской функцией.
test(char *buf, int n)
{
    int a, sum;
    // Здесь возбуждается исключение.
    for (a = 0; a < n; a++) sum += buf[a];
    return sum;
}

main()
{
    #define N 100
    // Инициализируем указатель на буфер.
    char *buf = 0;
```

```
// «Забываем» выделить память, здесь ошибка.
/* buf = malloc(100); */
// Передаем null-pointer некоторой функции.
test(buf, N);
}
```

Обитатели сумеречной зоны, или из морга в реанимацию

Хотите узнать, как заставить приложение продолжить нормальную работу после появления сообщения о критической ошибке? Это действительно очень актуально. Представьте, что рухнуло приложение, содержащее уникальные и еще не сохраненные данные. По минимуму их придется набивать заново, по максимуму – они потеряны для вас навсегда. На рынке имеется некоторое количество утилит, нацеленных на эту задачу (взять те же Norton Utilities), но их интеллектуальность оставляет желать лучшего и в среднем они срабатывают один раз из десяти. В то же самое время ручная реанимация программы воссоздает ее в 75-90% случаев.

Строго говоря, гарантированно восстановить работоспособность обрушившейся программы нельзя, равно как и невозможно выполнить откат тех действий, что предшествовали ее обрушению. В лучшем случае вам удастся сохранить свои данные на диск до того, как программа полностью потеряет нить управления и пойдет вразнос. Но и это неплохо!

Существует по меньшей мере три различных способа реанимации:

- принудительный выход из функции, возбудившей исключение;
- «раскрутка» стека с передачей управления назад;
- передача управления на функцию обработки сообщений.

Рассмотрим каждый из этих способов на примере приложения test.exe, копию которого вы можете скачать по адресу: www.samag.ru/source/test.zip.

Забегая вперед, отметим, что реанимации поддаются лишь те сбои, что вызваны алгоритмическими, а не аппаратными ошибками (т.е. сбоем оборудования). Если информация, хранящаяся в оперативной памяти, оказалась искажена в результате физического дефекта последней, то восстановить работоспособность упавшего приложения скорее всего уже не удастся, хотя если сбой не затронул жизненно важные структуры данных, некоторая надежда на благополучный исход все-таки есть.

Принудительный выход из функции

Запускаем тестовую программу, набиваем в одном или нескольких окнах какой-нибудь текст, затем в меню «Help» выбираем пункт «About TestCEdit» и в появившемся диалоговом окне щелкаем по кнопке «make error». Программа выбрасывает критическую ошибку и, если мы нажмем на «ОК», все несохраненные данные необратимо погибнут, что никак не входит в наши планы. Однако при наличии предварительно установленного отладчика мы еще можем кое-что предпринять. Пусть для определенности это будет Microsoft Visual Studio Debugger.

Нажимаем «Отмену», и отладчик немедленно дизассемблирует функцию, возбудившую исключение (см. листинг 5).

Листинг 5. Отладчик Microsoft Visual Studio Debugger дизассемблировал функцию, возбудившую исключение

```
0040135C  push  esi
0040135D  mov   esi,dword ptr [esp+8]
00401361  push  edi
00401362  movsx edi,byte ptr [ecx+esi]
00401366  add   eax,edi
00401368  inc   ecx
00401369  cmp   ecx,edx
0040136B  jl    00401362
0040136D  pop   edi
0040136E  pop   esi
0040136F  ret   8
```

Проанализировав причину возникновения исключения (функции передан указатель на невыделенную память), мы приходим к выводу, что заставить функцию продолжить свою работу невозможно, поскольку структура передаваемых данных нам не известна. Приходится прибегать к принудительному возврату в материнскую функцию, не забыв при этом установить флаг ошибки, сигнализируя программе, что текущая операция не была выполнена. К сожалению, никаких общепринятых флагов ошибок не существует, и различные функции используют различные соглашения. Чтобы выяснить, как обстоят дела в данном конкретном случае, мы должны дизассемблировать материнскую функцию и определить, какой именно код ошибки она ожидает.

Переместив курсор в окно дампа, набьем в строке адреса название регистра указателя вершины стека – «ESP» и нажмем на <Enter>. Содержимое стека тут же представит перед нашими глазами:

Листинг 6. Поиск адреса возврата из текущей функции (выделен красным шрифтом)

```
0012F488  0012FA64  0012FA64  004012FF
0012F494  00000000  00000064  00403458
0012F4A0  FFFFFFFF  0012F4C4  6C291CEA
0012F4AC  00000019  00000000  6C32FAF0
0012F4B8  0012F4C0  0012FA64  01100059
0012F4C4  006403C2  002F5788  00000000
0012F4D0  00640301  77E16383  004C1E20
```

Первые два двойных слова соответствуют машинным командам POP EDI/POP ESI и не представляют для нас совершенно никакого интереса. А вот следующее двойное слово содержит адрес выхода в материнскую процедуру (в приведенном выше листинге оно выделено красным шрифтом). Как раз оно-то нам и нужно!

Нажимаем <Ctrl-D> и затем 0x4012FF, отладчик послушно отображает следующий дизассемблерный текст:

Листинг 7. Дизассемблерный листинг материнской функции

```
004012FA  call  00401350
004012FF  cmp   eax,0FFh
00401302  je    0040132D
00401304  push  eax
00401305  lea   eax,[esp+8]
00401309  push  405054h
0040130E  push  eax
0040130F  call  dword ptr ds:[4033B4h]
00401315  add   esp,0Ch
00401318  lea   ecx,[esp+4]
0040131C  push  0
0040131E  push  0
00401320  push  ecx
00401321  mov   ecx,esi
00401323  call  00401BC4
00401328  pop   esi
00401329  add   esp,64h
0040132C  ret
```



```

0040132D push 0
0040132D ; эта ветка получает управление, если функция
; 401350h вернет FFh
0040132F push 0
00401331 push 405048h
00401336 mov ecx,esi
00401338 call 00401BC4
0040133D pop esi
0040133E add esp,64h
00401341 ret
    
```

Смотрите: если регистр EAX равен FFh, то материнская функция передает управление на ветку 40132Dh и, спустя несколько машинных команд, завершает свою работу, передавая бразды правления функции более высокого уровня. Напротив, если EAX != FFh, то его значение передается функции 4033B4h. Следовательно, мы можем предположить, что FFh – это флаг ошибки и есть. Возвращаемся в подопытную функцию, нажав <Ctrl-G> и «EIP», переходим в окно «Registers» и меняем значение EAX на FFh.

Теперь необходимо найти подходящую точку возврата из функции. Просто перейти к машинной команде «RET» нельзя, поскольку перед выходом из функции следует в обязательном порядке сбалансировать стек, или нас выбросит неизвестно куда, и программа обрушится окончательно.

В общем случае число PUSH-команд должно в точности соответствовать количеству POP (также учитывайте, что PUSH DWORD X эквивалентен SUB ESP, 4, а POP DWORD X – ADD ESP, 4). Проанализировав дизассемблерный листинг функции, мы приходим к выводу, что для достижения гармонии добра и зла мы должны стащить с вершины стека два двойных слова, соответствующие машинным командам 40135C: PUSH ESI и 401361: PUSH EDI. Это достигается передачей управления по адресу 40136Dh, где живут два добродушных POP, приводящие стек в равновесное состояние. Подводим сюда курсор и уверенным щелчком правой клавиши мыши вызываем контекстное меню, среди пунктов которого выбираем «Set Next Statement». Как вариант можно перейти в окно регистров и изменить значение EIP с 401362h на 40136Dh.

Нажатием <F5> мы заставляем процессор продолжить выполнение программы и... о чудо! Она действительно продолжает свою работу (незлобное ругательство на ошибку последней операции – не в счет!). Несохранные данные спасены!

Раскритка стека

Далеко не во всех случаях принудительный выход из функции оказывается возможным. Ряд критических сбоев затрагивает не одну, а сразу несколько вложенных функций, и тогда для реанимации программы мы должны совершить глубокий откат назад, продолжив выполнение программы с того места, где бы ее работоспособности ничто не угрожало. Точная глубина отката подбирается экспериментально и обычно составляет три-пять ступеней. Имейте в виду, что если вложенные функции модифицируют глобальные данные (например, данные «кучи»), то попытка отката может привести к полному краху отлаживаемой программы, поэтому требуемую глубину отката желательно угадать с первого раза, придерживаясь правила «лучше перебрать, чем недобрать». С другой сторо-

ны, чрезмерно глубокий откат ведет к потере всех несохраненных данных...

Процедура отката состоит из трех шагов:

- построения дерева вызовов;
- определения координат стекового фрейма для каждого из них;
- восстановления регистрового контекста материнской функции.

Хороший отладчик все это сделает за нас, и нам останется лишь записать в регистры EIP и ESP соответствующие значения. К сожалению, отладчик Microsoft Visual Studio Debugger к хорошим отладчикам не относится. Он довольно посредственно трассирует стек, пропуская FPO-функции (Frame Point Omission – функции с оптимизированным фреймом) и не сообщает координат стекового фрейма, «благодаря» чему самую трудоемкую часть работы нам приходится выполнять самостоятельно.

Впрочем, даже такой стек вызовов все же лучше, чем совсем ничего. Раскручивая его вручную, мы будем отталкиваться от того, что координаты фрейма естественным образом определяются по адресу возврата. Допустим, содержимое окна «Call Stack» выглядит так:

Листинг 8. Содержимое окна Call Stacks отладчика Microsoft Visual Studio Debugger

```

TESTCEDIT! 00401362()
MFC42! 6c2922ae()
MFC42! 6c298fc5()
MFC42! 6c292976()
MFC42! 6c291dcc()
MFC42! 6c291cea()
MFC42! 6c291c73()
MFC42! 6c291bfb()
MFC42! 6c291bba()
    
```

Попробуем найти в стеке адреса 6C2922AEh и 6C298FC5h, соответствующие двум последним ступеням исполнения. Нажимаем <ATL-6> для перехода в окно дампа и, воспользовавшись горячей комбинацией клавиш <Ctrl-G> в качестве базового адреса отображения, выбираем «ESP». Прокручивая окно дампа вниз, мы обнаруживаем оба адреса возврата (в приведенном ниже листинге они выделены рамкой):

Листинг 9. Содержимое стека после раскритки

```

0012F488 0012FA64 0012FA64 004012FF J
<-- 0040136F:ret 8 первый адрес возврата
0012F494 00000000 00000064 00403458 J
<-- 00401328:pop esi
0012F4A0 FFFFFFFF 0012F4C4 6C291CEA
0012F4AC 00000019 00000000 6C32FAF0
0012F4B8 0012F4C0 0012FA64 01100059
0012F4C4 00320774 002F5788 00000000
0012F4D0 00320701 77E16383 004C1E20
0012F4DC 00320774 002F5788 00000000
0012F4E8 000003E8 0012FA64 004F8CD8
0012F4F4 0012F4DC 002F5788 0012F560
0012F500 77E61D49 6C2923D8 00403458 J
<-- 0040132C:ret;
0012F50C 00000111 0012F540 6C2922AE J
<--6C29237E:pop ebx/pop ebp/ret 1ch
0012F518 0012FA64 000003E8 00000000
0012F518 0012FA64 000003E8 00000000
0012F524 004012F0 00000000 00000000
0012F530 00000000 00000000 0012FA64
0012F53C 000003E8 0012F564 6C298FC5
0012F548 000003E8 00000000 00000000
0012F554 00000000 000003E8 0012FA64
    
```

Ячейки памяти, лежащие выше адресов возврата, представляют собой значения регистров, сохраненные в стеке при входе в функцию и восстанавливаемые при ее завершении. Ячейки памяти, лежащие ниже адресов возврата, оккупированы аргументами функции (если, конечно, у функции есть аргументы), или же принадлежат локальным переменным материнской функции, если дочерняя функция не принимает никаких аргументов.

Возвращаясь к листингу 5, отметим, что два двойных слова, лежащие на верхушке стека, соответствуют машинным командам POP EDI и POP ESI, а следующий за ними адрес – 4012FFh – это тот самый адрес, управление которому передается командой 40136Fh:RET 8. Для продолжения раскрутки стека мы должны дизассемблировать код по этому адресу:

Листинг 10. Дизассемблерный листинг праматеринской функции («бабушки»)

```
004012FA call 00401350
004012FF cmp eax, 0FFh
00401302 je 0040132D
00401304 push eax
00401305 lea eax, [esp+8]
00401309 push 405054h
0040130E push eax
0040130F call dword ptr ds:[4033B4h]
00401315 add esp, 0Ch
00401318 lea ecx, [esp+4]
0040131C push 0
0040131E push 0
00401320 push ecx
00401321 mov ecx, esi
00401323 call 00401BC4
00401328 pop esi
00401329 add esp, 64h
0040132C ret ; SS:[ESP] = 6C2923D8
```

Прокручивая экран вниз, мы замечаем инструкцию ADD ESP, 64, закрывающую текущий кадр стека. Еще восемь байт снимает инструкция 40136Fh:RET 8 и четыре байта оттягивает на себя 401328:POP ESI. Таким образом, позиция адреса возврата в стеке равна: current_ESP + 64h + 8 + 4 == 70h. Спускаемся на 70h байт ниже и видим:

Листинг 11. Адрес возврата из праматеринской функции

```
0012F500 77E61D49 6C2923D8 00403458 <-- 00401328:POP ESI/ret;
```

Первое двойное слово – это значение регистра ESI, который нам предстоит вручную восстановить; второе – адрес возврата из функции. Нажатием <Ctrl-G>, «0x6C2923D8» мы продолжаем раскручивать стек:

Листинг 12. Дизассемблерный листинг прапраматеринской функции

```
6C2923D8 jmp 6C29237B
...
6C29237B mov eax, ebx
6C29237D pop esi
6C29237E pop ebx
6C29237F pop ebp
6C292380 ret 1Ch
```

Вот мы и добрались до восстановления регистров! Сместившись на одно двойное слово вправо (оно только что было вытолкнуто из стека командой RET), переходим в окно «Registers» и восстанавливаем регистры ESI, EBX, EBP, извлекая сохраненные значения из стека:

Листинг 13. Содержимое регистров, ранее сохраненных в стеке вместе с адресом возврата

```
0012F500 77E61D49 6C2923D8 00403458 <-- 6C29237D:pop esi
0012F50C 00000111 0012F540 6C2922AE <-- 6C29237E:pop ebx
/ pop ebp/ret 1Ch
```

Как вариант можно переместить регистр EIP на адрес 6C29237Dh, а регистр ESP на адрес 12F508h, после чего нажать на <F5> для продолжения выполнения программы. И этот прием действительно срабатывает! Причем реанимированная программа уже не ругается на ошибку последней операции (как это было при восстановлении путем принудительного выхода из функции), а просто ее не выполняет. Красота!

Передача управления на функцию обработки сообщений

Двум предыдущим способам «реанимации» приложений присущи серьезные ограничения и недостатки. При тяжелых разрушениях стека, вызванных атаками типа buffer overflow или же просто алгоритмическими ошибками, содержимое важнейших регистров процессора окажется искажено, и мы уже не сможем ни совершить откат (стек утерян), ни выйти из текущей функции (EIP смотрит в космос). В консольных приложениях в такой ситуации действительно очень мало что можно сделать... Вот GUI – другое дело! Концепция событийно ориентированной архитектуры наделяет всякое оконное приложение определенными серверными функциями. Даже если текущий контекст выполнения необратимо утерян, мы можем передать управление на цикл извлечения и диспетчеризации сообщений, заставляя программу продолжить обработку действий пользователя.

Классический цикл обработки сообщений выглядит так:

Листинг 14. Классический цикл обработки сообщений

```
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

Все, что нам нужно, – это передать управление на цикл while, даже не заботясь о настройке кадра стека, поскольку оптимизированные программы (а таковых большинство) адресуют свои локальные переменные не через EBP, а непосредственно через сам ESP. Конечно, при обращении к переменной msg, функция угробит содержимое стека, лежащее ниже его вершины, но это уже не важно.

Правда, при выходе из приложения оно упадет окончательно (ведь вместо адреса возврата из функции обработки сообщений машинная команда RET обнаружит на вершине стека неизвестно что), но это произойдет после сохранения всех данных, и потому никакой угрозы не несет. Исключение составляют приложения, «забывающие» закрыть все открытые файлы и переключившие эту работу на плечи функции ExitProcess. Что ж! Можно так подправить адрес возврата, чтобы он указывал на ExitProcess!

Давайте создадим простейшее Windows-приложение и поэкспериментируем с ним. Запустив Microsoft Visual

Studio выберем «New → Project → Win32 Application» и там – «Typical Hello, World application». Добавим новый пункт меню, а в нем: char *p; *p = 0; и откомпилируем этот проект с отладочной информацией.

Роняем приложение на пол и, запустив отладчик, подгоняем мышь к первой строке цикла обработки сообщений и в появившемся контекстном меню находим пункт «Set Next Statement». Нажимаем <F5> для возобновления работы программы и... она действительно возобновляет свою работу!

А теперь откомпилируем наш проект в чистовом варианте (т.е. без отладочной информации) и попробуем реанимировать приложение в голом машинном коде. Пользуясь тем обстоятельством, что Windows – это действительно многозадачная среда, в которой крушение одного процесса не мешает работе всех остальных, запустим свой любимый дизассемблер (например IDA PRO) и проанализируем таблицу импорта отлаживаемой программы (вообще-то это может сделать и бесплатно распространяемый dumpbin, но его отчет не так нагляден).

Целью нашего поиска будут функции TranslateMessage/DispatchMessage и перекрестные ссылки, ведущие к циклу выборки сообщений.

Листинг 15. Поиск функций TranslateMessage/DispatchMessage в таблице импорта

```
.idata:004040E0 ; BOOL __stdcall ↓
        TranslateMessage(const MSG *lpMsg)
.idata:004040E0 extrn TranslateMessage:dword ; DATA ↓
        XREF: WinMain@16+71?r
.idata:004040E0 ; WinMain@16+8D?r
.idata:004040E4 ; LONG __stdcall ↓
        DispatchMessageA(const MSG *lpMsg)
.idata:004040E4 extrn DispatchMessageA:dword ; DATA ↓
        XREF: WinMain@16+94?r
.idata:004040E8
```

С функцией DispatchMessage связана всего лишь одна перекрестная ссылка, со всей очевидностью ведущая к искомому циклу обработки сообщений, дизассемблерный код которого выглядит так:

Листинг 16. Дизассемблерный листинг функции обработки сообщений

```
.text:00401050      mov     edi, ds:GetMessageA
.text:00401050      ; первый вызов GetMessageA (это еще
                    ; не цикл, это только его преддверье)
.text:00401050
.text:00401056      push    0          ; wParamFilterMax
.text:00401058      push    0          ; wParamFilterMin
.text:0040105A      lea     ecx, [esp+2Ch+Msg]
.text:0040105A      ; ECX указывает на область памяти, через
                    ; которую GetMessageA станет возвращать
                    ; сообщение. Текущее значение ESP может быть
                    ; любым, главное, чтобы оно указывало на
                    ; действительную область памяти (см. карту
                    ; памяти, если значение ESP оказалось искажено
                    ; настолько, что вывело его в «космос»)
.text:0040105A
.text:0040105E      push    0          ; hWnd
.text:00401060      push    ecx         ; lpMsg
.text:00401061      mov     esi, eax
.text:00401063      call   edi         ; GetMessageA
.text:00401066      ; вызываем GetMessageA
.text:00401063
.text:00401065      test   eax, eax
.text:00401067      jz     short loc_4010AD
.text:00401067      ; проверка на наличие необработанных
                    ; сообщений в очереди
.text:00401067
...
.text:00401077 loc_401077: ;CODE XREF: _WinMain@16+A9?j
```

```
.text:00401077      ; начало цикла обработки сообщений
.text:00401077
.text:00401077      mov     eax, [esp+2Ch+Msg.hwnd]
.text:0040107B      lea     edx, [esp+2Ch+Msg]
.text:0040107B      ; EDX указывает на область памяти,
                    ; используемую для передачи сообщений
.text:0040107B
.text:0040107F      push    edx         ; lpMsg
.text:00401080      push    esi         ; hAccTable
.text:00401081      push    eax         ; hWnd
.text:00401082      call   ebx         ; TranslateAcceleratorA
.text:00401082      ; вызываем функцию TranslateAcceleratorA
.text:00401082
.text:00401084      test   eax, eax
.text:00401086      jnz     short loc_40109A
.text:00401086      ; проверка на наличие в очереди
                    ; необработанных сообщений
.text:00401086
.text:00401088      lea     ecx, [esp+2Ch+Msg]
.text:0040108C      push    ecx         ; lpMsg
.text:0040108D      call   ebp         ; TranslateMessage
.text:0040108D      ; вызываем функцию TranslateMessage, если
                    ; есть что транслировать
.text:0040108D
.text:0040108F      lea     edx, [esp+2Ch+Msg]
.text:00401093      push    edx         ; lpMsg
.text:00401094      call   ds:DispatchMessageA
.text:00401094      ; диспетчеризуем сообщение
.text:00401094
.text:0040109A loc_40109A: ; CODE XREF: WinMain@16+86?j
.text:0040109A      push    0          ; wParamFilterMax
.text:0040109C      push    0          ; wParamFilterMin
.text:0040109E      lea     eax, [esp+34h+Msg]
.text:004010A2      push    0          ; hWnd
.text:004010A4      push    eax         ; lpMsg
.text:004010A5      call   edi         ; GetMessageA
.text:004010A5      ; читаем очередное сообщение из очереди
.text:004010A5
.text:004010A7      test   eax, eax
.text:004010A9      jnz     short loc_401077
.text:004010A9      ; вращаем цикл обработки сообщений
.text:004010A9
.text:004010AB      pop     ebp
.text:004010AC      pop     ebx
.text:004010AD loc_4010AD: ; CODE XREF: WinMain@16+67?j
.text:004010AD      mov     eax, [esp+24h+Msg.wParam]
.text:004010B1      pop     edi
.text:004010B2      pop     esi
.text:004010B3      add     esp, 1Ch
.text:004010B6      retn     10h
.text:004010B6 _WinMain@16      endp
```

Мы видим, что цикл обработки сообщений начинается с адреса 401050h и именно на этот адрес следует передать управление, чтобы возобновить работу упавшей программы. Пробуем сделать это и... программа работает!

Разумеется, настоящее приложение оживить намного сложнее, поскольку цикл обработки сообщений в нем рассредоточен по большому количеству функций, отождествить которые при беглом дизассемблировании невозможно. Тем не менее, приложения, построенные на основе общедоступных библиотек (например MFC, OVL) обладают вполне предсказуемой архитектурой, и реанимировать их вполне возможно.

Рассмотрим, как устроен цикл обработки сообщений в MFC. Большую часть своего времени исполнения MFC-приложения проводят внутри функции CWinThread::Run(void), которая периодически опрашивает очередь на предмет поступления свежих сообщений и рассылает их соответствующим обработчикам. Если один из обработчиков споткнулся и довел систему до критической ошибки, выполнение программы может быть продолжено в функции Run. В этом-то и заключается ее главная прелесть!

Функция не имеет явных аргументов, но принимает скрытый аргумент this, указывающей на экземпляр клас-

са CWinThread или производный от него класс, без которого функция просто не сможет работать. К счастью, таблицы виртуальных методов класса CWinThread содержат достаточно количество «родимых пятен», чтобы указатель this можно было воссоздать вручную.

Загрузим функцию Run в дизассемблер и отметим все обращения к таблице виртуальных методов, адресуемой через регистр ECX.

Листинг 17. Дизассемблерный листинг функции Run (фрагмент)

```
.text:6C29919D n2k_Trasnlate_main:
; CODE XREF: MFC42_5715+1F?j
; MFC42_5715+67?j ...
.text:6C29919D      mov     eax, [esi]
.text:6C29919F      mov     ecx, esi
.text:6C2991A1      call    dword ptr [eax+64h]
; CWinThread::PumpMessage(void)
.text:6C2991A4      test    eax, eax
.text:6C2991A6      jz      short loc_6C2991DA
.text:6C2991A8      mov     eax, [esi]
.text:6C2991AA      lea     ebp, [esi+34h]
.text:6C2991AD      push    ebp
.text:6C2991AE      mov     ecx, esi
.text:6C2991B0      call    dword ptr [eax+6Ch]
; CWinThread::IsIdleMessage(MSG*)
.text:6C2991B3      test    eax, eax
.text:6C2991B5      jz      short loc_6C2991BE
.text:6C2991B7      push    1
.text:6C2991B9      mov     [esp+14h], ebx
.text:6C2991BD      pop     edi
.text:6C2991BE loc_6C2991BE:
; CODE XREF: MFC42_5715+51?j
.text:6C2991BE      push    ebx ; wRemoveMsg
.text:6C2991BF      push    ebx ; wParamFilterMax
.text:6C2991C0      push    ebx ; wParamFilterMin
.text:6C2991C1      push    ebx ; hWnd
.text:6C2991C2      push    ebp ; lParam
.text:6C2991C3      call    ds:PeekMessageA
.text:6C2991C9      test    eax, eax
.text:6C2991CB      jnz     short n2k_Trasnlate_main
.text:6C2991CD
```

Таким образом, функция Run ожидает получить указатель на двойное слово, указывающее на таблицу виртуальных методов, 0x19 и 0x1B элементы которой представляют собой функции PumpMessage и IsIdleMessage соответственно (или переходники к ним). Адреса импортируемых функций, если только динамическая библиотека не была перемещена, можно узнать в том же дизассемблере; в противном случае, следует отталкиваться от базового адреса модуля, отображаемого отладчиком по команде «Modules». При условии, что эти две функции не были перекрыты программистом, поиск нужной нам виртуальной таблицы не составит никакого труда.

По непонятным причинам библиотека MFC42.DLL не экспортирует символьных имен функций, и эту информацию нам приходится добывать самостоятельно. Обработав библиотеку MFC42.LIB утилитой dumpbin, запущенной с ключом «/ARCH», мы определим ординалы обеих функций (ординал PumpMessage – 5307, а IsIdleMessage – 4079). Остается найти эти значения в экспорте библиотеки MFC42.DLL (dumpbin /EXPORTS mfc42.dll > mfc42.txt), из чего мы узнаем что адрес функции PumpMessage: 6C291194h, а IsIdleMessage – 6C292583h.

Теперь мы должны найти указатели на функции PumpMessage/IsIdleMessage в памяти, а точнее – в секции данных, базовый адрес которой содержится в заголовке PE-файла, только помните, что в x86-процессорах

наименее значимый байт располагается по меньшему адресу, т.е. все числа записываются задом наперед (к сожалению, отладчик Microsoft Visual Studio Debugger не поддерживает операцию поиска в памяти, и нам придется действовать обходным путем – копировать содержимое дампа в буфер обмена, вставлять его в текстовый файл и, нажав <F7>, искать адреса уже там).

Долго ли, коротко ли, но интересующие нас указатели обнаруживаются по адресам 403044h/40304Ch (естественно, у вас эти адреса могут быть и другими). Причем обратите внимание: расстояние между указателями в точности равно расстоянию между указателями на [EAX + 64h] и [EAX + 6Ch], а очередность их размещения в памяти обратна порядку объявления виртуальных методов. Это хороший признак, и мы, скорее всего, находимся на правильном пути:

Листинг 18. Адреса функций IsIdleMessage/PumpMessage, найденные в секции данных

```
; IsIdleMessage/PumpMessage
00403044 6C2911D4 6C292583 6C291194
00403050 6C2913D0 6C299144 6C297129
0040305C 6C297129 6C297129 6C291A47
```

Указатели, указывающие на адреса 403048h/40304Ch, очевидно, и будут кандидатами в члены искомой таблицы виртуальных методов класса CWinThread. Расширив сферу поиска всем адресным пространством отлаживаемого процесса, мы обнаруживаем два следующих переходника:

Листинг 19. Переходники к функциям IsIdleMessage/PumpMessage, найденные там же

```
00401A20 jmp     dword ptr ds:[403044h] ; IsIdleMessage
00401A26 jmp     dword ptr ds:[403048h] ;
00401A2C jmp     dword ptr ds:[40304Ch] ; PumpMessage
```

Ага, уже теплее! Мы нашли не сами виртуальные функции, но переходники к ним. Раскручивая этот запутанный клубок, попробуем отыскать ссылки на 401A26h/401A2Ch, которые передают управление на приведенный выше код:

Листинг 20. Виртуальная таблица класса CWinThread

00403490	00401A9E 00401040 004015F0	
	<-- 0x0, 0x1, 0x2	элементы
0040349C	00401390 004015F0 00401A98	
	<-- 0x3, 0x4, 0x5	элементы
004034A8	00401A92 00401A8C 00401A86	
	<-- 0x6, 0x7, 0x8	элементы
004034B4	00401A80 00401A7A 00401A74	
	<-- 0x9, 0xA, 0xB	элементы
004034C0	00401010 00401A6E 00401A68	
	<-- 0xC, 0xD, 0xE	элементы
004034CC	00401A62 00401A5C 00401A56	
	<-- 0xF, 0x10, 0x11	элементы
004034D8	00401A50 00401A4A 00401A44	
	<-- 0x12, 0x13, 0x14	элементы
004034E4	00401A3E 004010B0 00401A38	
	<-- 0x15, 0x16, 0x17	элементы
004034F0	00401A32 00401A2C 00401A26	
	<-- 0x18, 0x19, 0x1A элементы (PumpMessage)	
004034FC	00401A20 00401A1A 00401A14	
	<-- 0x1B, 0x1C, 0x1D элементы (IsIdleMessage)	

Даже неопытный исследователь программ распознает в этой структуре данных таблицу виртуальных функций.

Указатели на переходники к PumpMessage/IsIdleMessage разделяются ровно одним элементом, как того и требуют условия задачи. Предположим, что это виртуальная таблица, которая нам и нужна. Для проверки этого предположения отсчитаем 0x19 элементов вверх от 4034F4h и попытаемся найти указатель, ссылающийся на ее начало. Если повезет и он окажется экземпляром класса CWinThread, тогда программа сможет корректно продолжить свою работу:

Листинг 21. Экземпляр класса CWinThread, вручную найденный нами в памяти

```
004050B8 00403490 00000001 00000000
004050C4 00000000 00000000 00000001
```

Действительно, в памяти обнаруживается нечто похожее. Записываем в регистр ECX значение 4050B8h, находим в памяти функцию Run (как уже говорилось, если только она не была перекрыта, ее адрес – 6C299164h – известен). Нажимаем <Ctrl-G>, затем «0x6C299164», и в контекстном меню, вызванном правой клавишей мыши, выбираем «Set Next Statement». Программа, отделившись легким испугом, продолжает свое исполнение, ну а мы на радостях идем пить пиво (кофе, квас, чай – по вкусу).

Аналогичным путем можно вернуть к жизни и зависшие приложения, потерявшие нить управления и не реагирующие ни на мышь, ни на клавиатуру.

Как подключить дампы памяти

...в отделе программ весь пол был усеян дырочками от перфокарт и какие-то мужики ползали по раскатанной по полу 20-метровой распечатке аварийного дампа памяти с целью обнаружения ошибки в распределителе памяти ОС-360. К президенту подошел начальник отдела и сообщил, что есть надежда сделать это еще к обеду.
Ю. Антонов «Юность Гейтса»

Дамп памяти (memory dump, также называемый корой [от английского core – сердцевина], crash- или аварийным дампом), сброшенный системой при возникновении критической ошибки – не самое убедительное средство для выявления причин катастрофы, но ничего другого в руках администратора зачастую просто не бывает. Последний вздох операционной системы, похожий на дурно пахнущую навозную кучу, из которой высовывается чей-то наполовину разложившийся труп, мгновенным снимком запечатленный в момент неустрашимого сбоя, – вот что такое дампы памяти! Копание в нем вряд ли доставит вам удовольствие. Не исключено, что истинного виновника краха системы вообще не удастся найти. Допустим, некий некорректно работающий драйвер вторгся в область памяти, принадлежащую другому драйверу, и наглым образом затер критические структуры данных, сделав из чисел винегрет. К тому моменту, когда драйвер-жертва пойдет вразнос, драйвер-хищник может быть вообще выгружен из системы, и определить его причастность к крушению системы по одному лишь дампу практически нереально.

Тем не менее, полностью игнорировать факт существования дампа, право же, не стоит. В конце концов, до возникновения интерактивных отладчиков ошибки в программах приходилось искать именно так. Избалованность современных программистов визуальными средствами ана-

лиза, увы, не добавляет им уверенности в тех ситуациях, когда неумолимая энтропия оставляет их со своими проблемами один на один. Но довольно лирики. Переходим к делу, расписывая каждое действие по шагам.

Первым делом необходимо войти в конфигурацию системы («Панель управления» → «Система»/«Control Panel» → «System») и убедиться, что настройки дампа соответствуют предъявляемым к ним требованиям («Дополнительно» → «Загрузка и восстановление» → «Отказ системы»/«Startup/Shutdown» → «Recovery» в Windows 2000 RUS и Windows NT 4.0 ENG соответственно). Операционная система Windows 2000 поддерживает три разновидности дампов памяти: малый дамп памяти (small memory dump), дамп памяти ядра (kernel memory dump) и полный дамп памяти (complete dump memory). Для изменения настроек дампа вы должны иметь права администратора.

Малый дамп памяти занимает всего лишь 64 Кб (а отнюдь не 2 Мб, как утверждает контекстная помощь) и включает в себя:

- копию голубого экрана смерти;
- перечень загруженных драйверов;
- контекст обвалившегося процесса со всеми его потоками;
- первые 16 Кб содержимого ядерного стека обвалившегося потока.

Разочаровывающие малоинформативные сведения! Непосредственный анализ дампа дает нам лишь адрес возникновения ошибки и имя драйвера, к которому этот адрес принадлежит. При условии, что конфигурация системы не была изменена после возникновения сбоя, мы можем загрузить отладчик и дизассемблировать подозреваемый драйвер, но это мало что даст. Ведь содержимое сегмента данных на момент возникновения сбоя нам неизвестно, более того, мы не можем утверждать, что видим те же самые машинные команды, что вызвали сбой. Поэтому малый дамп памяти полезен лишь тем администраторам, которым достаточно одного лишь имени нестабильного драйвера. Как показывает практика, в подавляющем большинстве случаев этой информации оказывается вполне достаточно. Разработчикам драйвера отсылается гневный бан-рапорт (вместе с дампом!), а сам драйвер тем временем заменяется другим – более новым и надежным. По умолчанию малый дамп памяти записывается в директорию %SystemRoot%\Minidump, где ему присваивается имя «Mini», дата записи дампа и порядковый номер сбоя на данный день. Например «Mini110701-69.dmp» – 69 дамп системы от 07 ноября 2001 года (не пугайтесь! это просто я отлаживал драйвера).

Дамп памяти ядра содержит намного более полную информацию о сбое и включает в себя всю память, выделенную ядром и его компонентами (драйверами, уровнем абстракции от оборудования и т. д.), а также копию экрана смерти. Размер дампа памяти ядра зависит от количества установленных драйверов и варьируется от системы к системе. Контекстная помощь утверждает, что эта величина составляет от 50 до 800 Мб. Ну, на счет 800 Мб авторы явно загнули, и объем в 50-100 Мб выглядит бо-

более вероятным (техническая документация на систему сообщает, что ориентировочный размер дампа ядра составляет треть объема физической оперативной памяти, установленной на системе). Это наилучший компромисс между накладными расходами на дисковое пространство, скоростью сброса дампа и информативностью последнего. Весь джентльменский минимум информации в вашем распоряжении. Практически все типовые ошибки драйверов и прочих ядерных компонентов могут быть локализованы с точностью до байта, включая и те, что вызваны физическим сбоем аппаратуры (правда, для этого вы должны иметь некоторый патологоанатомический опыт исследования трупных дампов системы). По умолчанию дмп памяти ядра записывается в файл %SystemRoot%\Memory.dmp, затирая или не затирая (в зависимости от текущих настроек «Системы») предыдущий дмп.

Полный дмп памяти включает в себя все содержимое физической памяти компьютера, занятое как прикладными, так и ядерными компонентами. Полный дмп памяти оказывается особенно полезным при отладке ASPI/SPTI приложений, которые в силу своей специфики могут уронить ядро даже с прикладного уровня. Несмотря на довольно большой размер, равный размеру оперативной памяти, полный дмп остается наиболее любимым дампом всех системных программистов (системные же администраторы в своей массе предпочитают малый дмп). Это не покажется удивительным, если вспомнить, что объемы жестких дисков давно перевалили за отметку 100 Гб, а оплата труда системных программистов за последние несколько лет даже несколько возросла. Лучше иметь невостребованный полный дмп под рукой, чем кусать локти при его отсутствии. По умолчанию полный дмп памяти записывается в файл %SystemRoot%\Memory.dmp, затирая или не затирая (в зависимости от текущих настроек «Системы») предыдущий дмп.

Выбрав предпочтительный тип дампа, давайте совершим учебный урон системы, отрабатывая методику его анализа в полевых условиях. Для этого нам понадобится:

- Комплект разработчика драйверов (Driver Development Kit или сокращенно DDK), бесплатно распространяемый фирмой Microsoft и содержащий в себе подробную техническую документацию по ядру системы; несколько компиляторов Си/Си++ и ассемблера, а также достаточно продвинутые средства анализа дампа памяти.
- Драйвер W2K_KILL.SYS или любой другой драйвер-убийца операционной системы, например BDOS.EXE от Марка Русиновича, позволяющий получить дмп в любое удобное для нас время, не дожидаясь возникновения критической ошибки (бесплатную копию программы можно скачать с адреса <http://www.sysinternals.com>).
- Файлы символьных идентификаторов (symbol files), необходимые отладчикам ядра для его нормального функционирования и делающие дизассемблерный код более наглядным. Файлы символьных идентификаторов входят в состав «зеленого» набора MSDN, но, в принципе, без них можно и обойтись, однако переменная окружения _NT_SYMBOL_PATH по любому должна быть определена, иначе отладчик i386kd.exe работать не будет.

- Одна или несколько книжек, описывающих архитектуру ядра системы. Очень хороша в этом смысле «Внутреннее устройство Windows 2000» Марка Русиновича и Дэвида Соломона, интересная как системным программистам, так и администраторам.

Итак, установив DDK на свой компьютер и завершив все приложения, запускаем драйвер-убийцу и... под скрипящий звук записывающегося дампа, система немедленно выбрасывает голубой экран смерти, кратко информирующий нас о причинах сбоя (см. рис. 5).

```
*** STOP: 0x0000001E (0xC0000005, 0xBE80B000, 0x00000000, 0x00000000)
KMODE_EXCEPTION_NOT_HALTED

*** Address 0xBE80B000 base at 0xBE80A000, Date Stamp 389db915 - w2k_kill.sys

Beginning dump of physical memory
Dumping physical memory to disk: 69
```

Рисунок 5. Голубой экран смерти (BSOD – Blue Screen Of Death), свидетельствующий о возникновении неустрашимого сбоя системы с краткой информацией о нем

Для большинства администраторов голубой экран смерти означает лишь одно – системе плохо. Настолько, что она предпочла смерть позору неустойчивого функционирования. Что же до таинственных писем – они остаются сплошной загадкой. Но только не для настоящих профессионалов!

Мы начнем с левого верхнего угла экрана и, зигзагами спускаясь вниз, трассируем все надписи по порядку.

- «*** STOP:» буквально означает «останов [системы]» и не несет в себе никакой дополнительной информации;
- «0x0000001E» представляет собой Bug Check-код, содержащий категорию сбоя. Расшифровку Bug Check-кодов можно найти в DDK. В данном случае это 0x1E – KMODE_EXCEPTION_NOT_HALTED, о чем и свидетельствует символьное имя, расположенное строкой ниже. Краткое объяснение некоторых, наиболее популярных Bug Check-кодов приведено в таблице 1. Полноту фирменной документации она, разумеется, не заменяет, но некоторое представление о целесообразности скачивания 70 Мб DDK все-таки дает;
- арабская вязь в круглых скобках – это четыре Bug Check-параметра, физический смысл которых зависит от конкретного Bug Check-кода и вне его контекста теряет всякий смысл; применительно к KMODE_EXCEPTION_NOT_HALTED – первый Bug Check-параметр содержит номер возбужденного исключения. Судя по таблице 1, это – STATUS_ACCESS_VIOLATION – доступ к запрещенному адресу памяти – и четвертый Bug Check-параметр указывает, какой именно. В данном случае он равен нулю, следовательно, некоторая машинная инструкция попыталась совершить обращение по null-pointer соответствующему инициализированному указателю, ссылающемуся на невыделенный регион памяти. Ее адрес содержится во втором Bug Check-параметре. Третий Bug Check-параметр в данном конкретном случае не определен;
- «*** Address 0xBE80B000» – это и есть тот адрес, по которому произошел сбой. В данном случае он идентифицирован второму Bug Check-параметру, однако так бывает далеко не всегда (Bug Check-коды собственно и не подражались хранить чьи-либо адреса);

■ «base at 0xBE80A00» – содержит базовый адрес загрузки модуля нарушителя системного порядка, по которому легко установить паспортные данные самого этого модуля (внимание: далеко не во всех случаях правильное определение базового адреса вообще возможно). Воспользовавшись любым подходящим отладчиком (например, soft-ice от Нумега или i386kd от Microsoft), введем команду, распечатывающую перечень загруженных драйверов с их краткими характеристиками (в i386kd это осуществляется командой «!drivers»). Как одним из вариантов можно воспользоваться утилитой drivers.exe, входящей в NTDDK. Но какой бы вы путь ни избрали, результат будет приблизительно следующим:

```

kd> !drivers!drivers
Loaded System Driver Summary
Base      Code Size  Data Size  Driver Name      Creation Time
80400000 142dc0 (1291 kb) 4d680 (309 kb) ntoskrnl.exe    Wed Dec 08 02:41:11 1999
80062000 cc20 ( 51 kb) 32c0 ( 12 kb) hal.dll        Wed Nov 03 04:14:22 1999
f4010000 1760 ( 5 kb) 1000 ( 4 kb) BOOTVID.DLL    Thu Nov 04 04:24:33 1999
bff80000 21ee0 ( 135 kb) 59a0 ( 22 kb) ACPI.sys       Thu Nov 11 04:06:04 1999
be190000 16f60 ( 91 kb) cccc0 ( 51 kb) kmixer.sys     Wed Nov 10 09:15:20 1999
bada0000 33ee0 ( 213 kb) 10ae0 ( 66 kb) ATMPD.DLL     Fri Nov 12 06:48:40 1999
be80a000 200 ( 0 kb) a00 ( 2 kb) w2k_kill.sys  Mon Aug 28 02:40:12 2000
TOTAL: 835ca0 (8407 kb) 326180 (3224 kb) ( 0 kb 0 kb)

```

Обратите внимание на выделенную жирным цветом строку с именем «w2k_kill.sys», найденную по ее базовому адресу 0xBE80A00. Это и есть тот самый драйвер, который нам нужен! А впрочем, этого можно и не делать, поскольку имя «неправильного» драйвера и без того присутствует на голубом экране;

■ две нижние строки отражают прогресс сброса дампа на диск, развлекая администратора чередой быстро меняющихся циферок на это время.

Таблица 1.

hex-код	Категория Символьное имя	Описание
0xA	IRQL_NOT_LESS_OR_EQUAL	Драйвер попытался обратиться к странице памяти на уровне DISPATCH_LEVEL или более высоким, что и привело к краху, поскольку менеджер виртуальной памяти работает на более низком уровне. Источником сбоя может быть и BIOS, и драйвер, и системный сервис (особенно этим грешат вирусные сканеры и FIM-поперы). Как вариант – проверить кабельные терминаторы на SCSI-накопителях и Master/Slave на IDE, отключите кширование памяти в BIOS. Если и это не поможет, обратитесь к четырем параметрам Bug Check-кода, содержащим ссылку на память, к которой осуществлялся доступ, уровень IRQL, тип доступа (чтение/запись) и адрес машинной инструкции драйвера. Компонент ядра возбудил исключение и “забыл” его обработать, номер исключения содержится в первом Bug Check-параметре. Обычно он принимает одно из следующих значений: 0x80000003 (STATUS_BREAKPOINT): встретилась программная точка останова – отладочный рудимент, по небрежности не удаленный разработчиком драйвера; 0x80000005 (STATUS_ACCESS_VIOLATION): доступ к запрещенному адресу (четвертый Bug Check-параметр уточняет к какому) – ошибка разработчика; 0xC000021A (STATUS_SYSTEM_PROCESS_TERMINATED): сбой процессов CSRSS и/или Winlogon, источником которого могут быть как компоненты ядра, так и пользовательские приложения; обычно это происходит при заражении машины вирусом или нарушении целостности системных файлов; 0xC0000221 (STATUS_IMAGE_CHECKSUM_MISMATCH): целостность одного из системных файлов оказалась нарушена. Второй Bug Check-параметр содержит адрес машинной команды, возбудившей исключение.
0x24	NTFS_FILE_SYSTEM	Проблема с драйвером NTFS.SYS, обычно возникающая вследствие физического разрушения диска, реже – при остром недостатке физической оперативной памяти.
0x2E	DATA_BUS_ERROR	Драйвер обратился к несуществующему физическому адресу; если только это не ошибка драйвера, оперативная память и/или кэш-память процессора (видеопамять) неисправны или же работают на запредельных тактовых частотах.
0x35	NO_MORE_IRP_STACK_LOCATIONS	Драйвер более высокого уровня обратился к драйверу более низкого уровня через IoCallDriver-интерфейс, свободного пространства в IRP-стеке не оказалось и передать весь IRP-пакет целиком не удалось. Это гибельная ситуация, не имеющая прямых решений; попытайтесь удалить один или несколько наименее нужных драйверов, быть может тогда система заработает.
0x3F	NO_MORE_SYSTEM_PTES	Результат сильной фрагментации таблицы PTE, приводящей к невозможности выделения требуемого драйвером блока памяти; обычно это характерно для аудио/видео драйверов, манипулирующих огромными блоками памяти и к тому же не всегда их вовремя освобождающих; для решения проблемы попробуйте увеличить количество PTE (до 50,000 максимум) в следующей ветке реестра: HLLM\SYSTEM\CurrentControlSet\Control\Session-Manager\Memory Management\SystemPages
0x50	PAGE_FAULT_IN_NONPAGED_AREA	Обращение к несуществующей странице памяти, вызванное либо неисправностью оборудования (как правило, оперативной, видео или кэш-памяти), либо некорректно спроектированным сервисом (этим грешат многие антивирусы, в том числе Антивирус Касперского и Doctor Web), либо разрушениями NTFS-тома (запустите chkdsk с ключами /f и /r), также попробуйте запретить кширование памяти в BIOS.
0x58	FTDISK_INTERNAL_ERROR	Сбой RAID-массива – при попытке загрузки с основного диска система обнаружила, что он поврежден, тогда она обратилась к его зеркалу, но таблицы разделов не оказалось и там.

hex-код	Категория Символьное имя	Описание
0x76	PROCESS_HAS_LOCKED_PAGES	Драйвер не смог освободить залочные страницы после завершения операции ввода-вывода; для определения имени дефективного драйвера следует обратиться к ветке HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management, и установить параметр TrackLockedPages типа DWORD в значение 1, потом перезагрузить систему, после чего та будет сохранять трассируемый стек и, если нехороший драйвер вновь начнет чудить, возникнет BSOD с Bug Check-кодом 0xCB, позволяющим определить виновника.
0x77	KERNEL_STACK_INPAGE_ERROR	Страница данных памяти ядра по техническим причинам недоступна, если первый Bug Check-код не равен нулю, то он может принимать одно из следующих значений: 0xC000009A (STATUS_INSUFFICIENT_RESOURCES – недостаток системных ресурсов; 0xC000009C (STATUS_DEVICE_DATA_ERROR – ошибка чтения с диска (может bad-sector?); 0xC000009D (STATUS_DEVICE_NOT_CONNECTED – система не видит привод (неисправность контроллера, плохой контакт шлейфа); 0xC000016A (STATUS_DISK_OPERATION_FAILED – ошибка чтения диска (bad-сектор или неисправный контроллер); 0xC0000185 (STATUS_IO_DEVICE_ERROR – неправильное терминирование SCSI-привода или конфликт IRQ IDE-приводов. Нулевое же значение первого Bug Check-кода указывает на неизвестную аппаратную проблему. Такое сообщение может появиться и при заражении системы вирусами, и при разрушении диска старыми докторами, и при отказе RAM – войдите в консоль восстановления и запустите ChkDsk с ключом /r.
0x7A	KERNEL_DATA_INPAGE_ERROR	Страница данных памяти ядра по техническим причинам недоступна, второй Bug Check-параметр содержит статус обмена, четвертый – виртуальный страничный адрес, загрузить который не удалось. Возможные причины сбоя – те же дефектные сектора, поппание в pagelste.sys, сбой дискового контроллера, ну и вирусы, наконец.
0x7B	INACCESSIBLE_BOOT_DEVICE	Загрузка устройства недоступна – таблица разделов повреждена или не соответствует файлу boot.ini. Также такое сообщение появляется при замене материнской платы с интегрированным IDE-контроллером (или замене SCSI-контроллера), поскольку всякий контроллер требует “своих” драйверов и при подключении жесткого диска с установленной NT на компьютер, оснащенный несовместимым оборудованием, операционная система просто откажется грузиться и ее необходимо переустановить (оплаты администраторы могут переустановить непосредственно сами дисковые драйвера, загрузившись с консоли восстановления). Также не помешает проверить общую исправность оборудования и наличие вирусов на диске.
0x7F	UNEXPECTED_KERNEL_MODE_TRAP	Исключение процессора, несовместимое операционной системой. Обычно возникает вследствие неисправности оборудования (как правило – разгона CPU), его несовместимости с установленными драйверами или алгоритмическими ошибками в самих драйверах. Проверьте исправность оборудования и удалите все посторонние драйвера. Первый Bug Check-параметр содержит номер исключения и может принимать следующие значения: 0x00 – попытка деления на ноль; 0x01 – исключение системного отладчика; 0x03 – исключение точки останова; 0x04 – переполнение; 0x05 – генерируется инструкцией BOUND; 0x06 – неверный опкод; 0x07 – двойной опкод (Double Fault). Описание остальных исключений содержится в документации на процессоры Intel и AMD.
0xC2	BAD_POOL_CALLER	Текущий поток вызвал некорректный pool-request, что обычно происходит по причине алгоритмической ошибки, допущенной разработчиком драйвера. Ошибка, судя по всему, и сама система не остается без ошибок, поскольку для устранения этого голубого экрана Microsoft рекомендует установить SP2.
0xCB	DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS	После завершения процедуры ввода/вывода, драйвер не может освободить заблокированные страницы (см. PROCESS_HAS_LOCKED_PAGES). Первый Bug Check-параметр содержит вызываемый, а второй Bug Check-параметр вызывающий адрес. Последний, четвертый параметр указывает на UNICODE-строку с именем драйвера.
0xD1	DRIVER_IRQL_NOT_LESS_OR_EQUAL	То же самое, что и IRQL_NOT_LESS_OR_EQUAL.
0xE2	MANUALLY_INITIATED_CRAS	Сбой системы, спровоцированный вручную, путем нажатия горячей комбинации клавиш «Ctrl+Scroll Lock», при условии, что параметр CrashOnCtrlScroll реестра: M\System\CurrentControlSet\Services\lsass\Parameters содержит ненулевое значение.

Восстановление системы после критического сбоя

Неестественное, почти половое влечение к кнопке F8 появилось в Кролике совершенно не внезапно.

Щербаков Андрей

«14400 бод и 19200 юзеров, и те же самые все-все-все...»

Операционные системы семейства NT достаточно безболезненно переносят критические сбои, даже если те произошли в самый неудобный момент времени (например, в период дефрагментации диска). Отказоустойчивый драйвер файловой системы все сделает сам (хотя запустить ChkDsk все же не помешает).

Если был выбран «полный дамп памяти» или «дамп памяти ядра», то при следующей успешной загрузке системы жесткий диск будет долго молотить головкой, даже если к нему и не происходит никаких обращений. Не пугайтесь! Просто Windows перемещает дамп из виртуальной памяти на место его постоянного проживания. Запустив «Диспетчер Задач», вы увидите новый процесс в списке – SaveDump.exe, – вот он этим и занимается. Необходи-

димось в подобной двухтактной схеме сброса дампа объясняется тем, что в момент возникновения критической ошибки работоспособность драйверов файловой системы уже не гарантируется, и операционная система не может позволить себе их использовать, ограничиваясь временным размещением дампа в виртуальной памяти. Кстати, если имеющегося объема виртуальной памяти окажется недостаточно (Дополнительно → Параметры быстрогодействия → Виртуальная память), сброс дампа окажется невозможным.

Если же система от загрузки отказывается, упорно забрасывая вас голубыми экранами смерти, вспомните о существовании клавиши <F8> и выберите пункт «Загрузка последней удачной конфигурации» («Last Known Good Configuration»). Более радикальной мерой является запуск системы в безопасном (safe) режиме с минимумом загружаемых служб и драйверов. Переустановка системы – это крайняя мера и без особой нужды к ней лучше не прибегать. Лучше войдите в «консоль восстановления» и перетащите файл дампа на другую машину для его исследования.

Подключение дампа памяти

Для подключения дампа памяти к отладчику Windows Debugger (windbg.exe) в меню «File» выберите пункт «Crash Dump» или воспользуйтесь горячей комбинацией <Ctrl-D>. В отладчике i386kd.exe для той же цели служит ключ «-z» командной строки, за которым следует полный путь к файлу дампа, отделенный от ключа одним или несколькими пробелами, при этом переменная окружения _NT_SYMBOL_PATH должна быть определена и содержать полный путь к файлам символьных идентификаторов, в противном случае отладчик аварийно завершит свою работу. Как один из вариантов можно указать к командой строке ключ «-у», и тогда экран консоли будет выглядеть так: «i386kd -z C:\WINNT\memory.dmp -у C:\WINNT\Symbols», причем отладчик следует вызывать из Checked Build Environment/Free Build Environment консоли, находящейся в папке Windows 2000 DDK, иначе у вас ничего не получится.

Хорошая идея – ассоциировать dmp-файлы с отладчиком i386kd, запуская их одним ударом «Enter» из FAR. Впрочем, выбор средства анализа – дело вкуса. Кому-то нравится KAnalyze, а кому-то достаточно и простенького DumpChk. Выбор аналитических инструментов чрезвычайно велик (один лишь DDK содержит четыре из них!), и чтобы хоть как-то определиться с выбором, мы остановимся на i386kd.exe, также называемом Kernel Debugger.

Как только консоль отладчика появится на экране (а Kernel Debugger – это консольное приложение, горячо любимое всеми, кто провел свою молодость за текстовыми терминалами), курсор наскоро дизассемблирует текущую машинную инструкцию и своим тревожным мерцанием затягивает нас в пучину машинного кода. «Ну что, глазки строить будем или все-таки дизассемблировать?» – незлобно ворчим мы, выбивая на клавиатуре команду «и», заставляющую отладчик продолжить дизассемблирование.

Судя по символьным идентификаторам PspUnhandledExceptionInSystemThread и KeBugCheckEx, мы находимся глубоко в ядре, а точнее – в окрестностях того кода, что выводит BSOD на экран:

Листинг 22. Результат дизассемблирования подключенного дампа памяти с текущего адреса

```
8045249c 6a01          push    0x1
kd>u
PspUnhandledExceptionInSystemThread@4:
80452484 8B442404 mov     eax, dword ptr [esp+4]
80452488 8B00          mov     eax, dword ptr [eax]
8045248A FF7018      push    dword ptr [eax+18h]
8045248D FF7014      push    dword ptr [eax+14h]
80452490 FF700C      push    dword ptr [eax+0Ch]
80452493 FF30          push    dword ptr [eax]
80452495 6A1E          push    1Eh
80452497 E8789AFDFF call    KeBugCheckEx@20
8045249C 6A01          push    1
8045249E 58           pop     eax
8045249F C20400      ret     4
```

В стеке ничего интересного также не содержится, судите сами (просмотр содержимого стека осуществляется командой «kb»):

Листинг 23. Содержимое стека не дает никаких намеков на природу истинного виновника

```
kd> kb
ChildEBP RetAddr  Args to Child
f403f71c 8045251c f403f744 8045cc77 f403f74c  J
ntoskrnl!PspUnhandledExceptionInSystemThread+0x18
f403fddc 80465b62 80418ada 00000001 00000000  J
ntoskrnl!PspSystemThreadStartup+0x5e
00000000 00000000 00000000 00000000 00000000  J
ntoskrnl!KiThreadStartup+0x16
```

Такой поворот событий ставит нас в тупик. Сколько бы мы ни дизассемблировали ядро, это ни на йоту не приблизит нас к источнику критической ошибки. Что ж, все вполне логично. Текущий адрес (8045249Ch) лежит далеко за пределами драйвера-убийцы (0BE80A00h). Хорошо, давайте развернемся и пойдем другим путем. Помните тот адрес, что высвечивал голубой экран смерти? Не помните – не беда! Если это только не запрещено настройками, копии всех голубых экранов сохраняются в Журнале системы. Откроем его («Панель управления») → «Администрирование» → «Просмотр событий»):

Листинг 24. Копия голубого экрана смерти, сохраненная в системном журнале

```
Компьютер был перезагружен после критической ошибки:
0x0000001e (0xc0000005, 0xbe80b000, 0x00000000, 0x00000000).
Microsoft Windows 2000 [v15.2195]
Копия памяти сохранена: C:\WINNT\MEMORY.DMP.
```

Отталкиваясь от категории критической ошибки (0x1E), мы без труда сможем определить адрес инструкции-убийцы – 0xBE80B000 (в приведенном выше листинге он выделен красным шрифтом). Даем команду «и 0xBE80B000» для просмотра его содержимого и видим:

Листинг 25. Результат дизассемблирования дампа памяти по адресу, сообщенному голубым экраном смерти

```
kd>u 0xBE80B000
be80b000 a100000000 mov     eax, [00000000]
be80b005 c20800      ret     0x8
be80b008 90           nop
```

```
be80b009 90      пор
be80b00a 90      пор
be80b00b 90      пор
be80b00c 90      пор
be80b00d 90      пор
```

Ага! Вот это уже больше похоже на истину! Инструкция, на которую указывает курсор (в тексте она выделена красным цветом), обращается к ячейке с нулевым адресом, возбуждая тем самым губительное для системы исключение. Теперь мы точно знаем, какая ветка программы вызвала сбой.

Хорошо, а как быть, если копии экрана смерти в нашем распоряжении нет? На самом деле, синий экран всегда с нами, надо только знать, где искать! Попробуйте открыть файл дампа в любом hex-редакторе, и вы обнаружите следующие строки:

Листинг 26. Копия голубого экрана в заголовке дампа программы

```
00000000: 50 41 47 45 44 55 4D 50 | 0F 00 00 00 93 08 00 00 | PAGEDUMP: Y
00000010: 00 00 03 00 00 80 8B 81 | C0 A4 46 80 80 A1 46 80 | AJB LFAAGFA
00000020: 4C 01 00 00 01 00 00 00 | 1E 00 00 00 05 00 00 C0 | LE O A L
00000030: 00 B0 80 BE 00 00 00 00 | 00 00 00 00 00 41 47 45 | AGE
```

С первого же взгляда удастся опознать все основные Bug Check-параметры: 1E 00 00 00 – это код категории сбоя 0x1E (на x86 процессорах наименее значимый байт располагается по меньшему адресу, то есть все числа записываются задом наперед); 05 00 00 C0 – код исключения ACCESS VIOLATION; а 00 B0 80 BE – и

есть адрес машинной команды, породившей это исключение. В комбинации же 0F 00 00 00 93 08 легко узнается номер билда системы, стоит только записать его в десятичной нотации.

Для просмотра Bug Check-параметров в более удобочитаемом виде можно воспользоваться следующей командой отладчика: «dd KiBugCheckData»:

Листинг 27. Bug Check параметры, отображаемые в удобочитаемом виде

```
kd> dd KiBugCheckData
dd KiBugCheckData
8047e6c0 0000001e c0000005 be80b000 00000000
8047e6d0 00000000 00000000 00000001 00000000
8047e6e0 00000000 00000000 00000000 00000000
8047e6f0 00000000 00000000 00000000 00000000
8047e700 00000000 00000000 00000000 00000000
8047e710 00000000 00000000 00000000 00000000
8047e720 00000000 00000000 00000000 00000000
8047e730 00000000 e0ffffff edffffff 00020000
```

Другие полезные команды: «!drivers» – выводящая список драйверов, загруженных на момент сбоя, «!arbitr» – показывающая всех арбитров вместе с диапазонами арбитража, «!filecache» – отображающая информацию о кэше файловой системы и PT, «!vm» – отчитывающая об использовании виртуальной памяти и т. д. – всех не перечислишь! (полный перечень команд вы найдете в руководстве по своему любимому отладчику).



3R
react realize recruit



react • realize • recruit

"We work to guarantee your success"

3R company is a recruiting agency with wide experience in the search for top and middle-level personnel both for international and Russian companies operating in various spheres of business. 3R also offers HR consulting services and professional psychological testing. The Agency staff experience lays in different industries and services: FMCG, IT, PR & Marketing, Oil & Gas, Finance.

t r i . r

21b, Kuusinena street, 125252, "ICSTI's representative", Moscow
phone: 258 0909 (multichannel), fax: 258 0908, e-mail: info@3r.ru

Конечно, в реальной жизни определить истинного виновника краха системы намного сложнее, поскольку всякий нормальный драйвер состоит из множества сложно взаимодействующих функций, образующих запутанные иерархические комплексы, местами пересеченные туннелями глобальных переменных, превращающих драйвер в самый настоящий лабиринт. Приведем только один пример. Конструкция вида «mov eax, [ebx]», где ebx == 0, работает вполне нормально, послушно возбуждая исключение, и пытаться поговорить с ней по-мужски – бессмысленно! Нужно найти тот код, который записывает в EBX нулевое значение, и сделать это непросто. Можно, конечно, просто прокрутить экран вверх, надеясь, что на данном участке программный код выполнялся линейно, но никаких гарантий, что это действительно так, у нас нет, равно как нет и возможности обратной трассировки (back trace). Грубо говоря, адрес предшествующей машинной инструкции нам неизвестен, и закладываться на прокрутку экрана нельзя!

Загрузив подопытный драйвер в любой интеллектуальный дизассемблер, автоматически восстанавливающий перекрестные ссылки (например IDA PRO), мы получим более или менее полное представление о топологии управляющих ветвей программы. Конечно, ди-

зассемблирование в силу своей статической природы, не гарантирует, что управление не перекинулось откуда-то еще, но по крайней мере сужает круг поиска. Вообще же, о дизассемблировании написано множество хороших книг (и «Фундаментальные основы хакерства» Криса Касперски в том числе), поэтому не будем останавливаться на этом вопросе, а просто пожелаем всем читателям удачи.

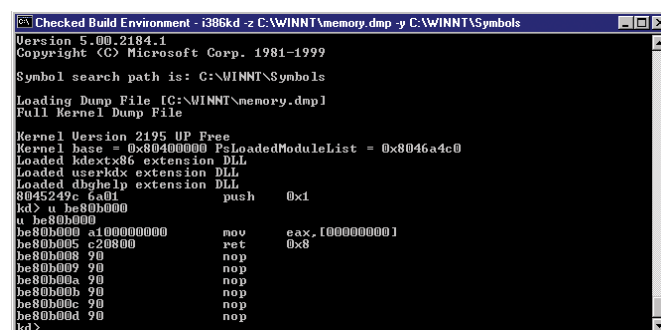


Рисунок 6. Отладчик i386kd за работой. Несмотря на свою отталкивающую внешность, это чрезвычайно мощный и удобный в работе инструмент, позволяющий проворачивать умопомрачительные пассажи нажатием всего пары-тройки клавиш (одна из которых вызывает ваш собственный скрипт)

