энциклопедия антиотладочных приемов — кто сломал мой бряк?! выпуск #06h

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

когда в очередной раз на форуме спросили: почему установленная точка останова на срабатывает (а ведь должна!) или (что еще хуже) приводит программу к краху, мыщъх не выдержал, нервно защелкал хвостом и застучал лапами по клавиатуре, попытавшись ответить на этот вопрос раз и навсегда, собрав воедино огромное количество разрозненной инфы

вокруг INT 03h

Программная точка останова на исполнение (software breakpoint on execution) физически представляет собой однобайтовую процессорную инструкцию CCh (INT 03h), внедряемую дебаггером непосредственно в отлаживаемый код с неизбежной перезаписью оригинального содержимого. Встретившись с INT 03h, процессор генерирует исключение типа EXCEPTION_BREAKPOINT, перехватываемое отладчиком, который останавливает выполнение программы, автоматически восстанавливая содержимое байта, "испорченного" точкой останова.

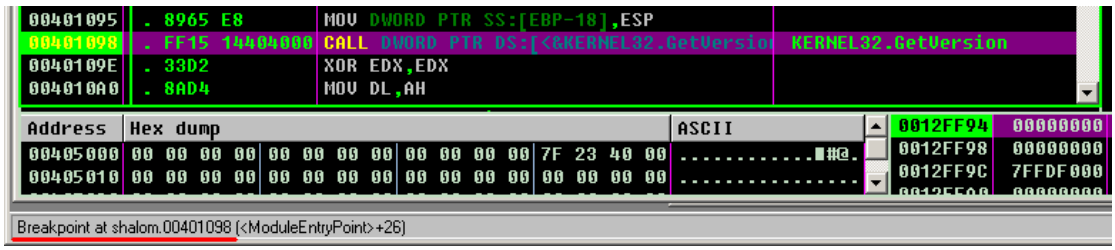

Рисунок 1 вот так срабатывают программные точки останова

Именно так и поступает Ольга при нажатии клавиши <F2> и Soft-Ice по <F7>. Достоинство программных точек останова в том, что их количество ограничено только архитектурными особенностями отладчика (т.е. практически неограниченно), в то время как аппаратных точек останова, поддерживаемых процессором на железном уровне, всего четыре.

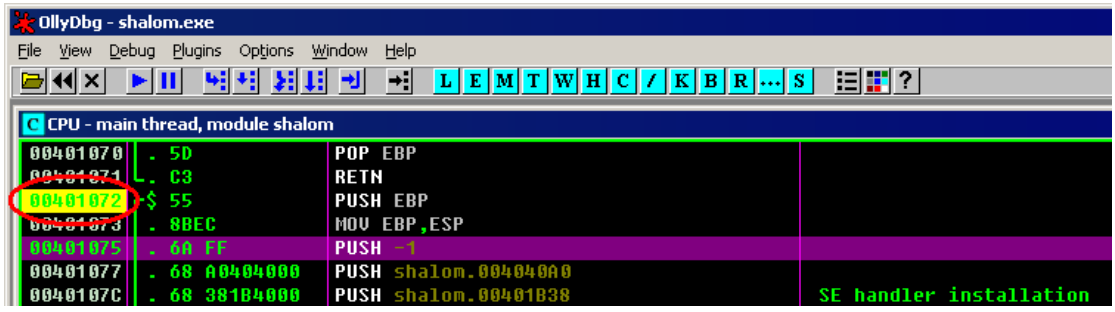

Рисунок 2 установка программной точки останова в Ольге по <F2>

Недостаток же программных точек останова в том, что они требуют модификации отлаживаемого кода, что легко обнаруживается ломаемой программой тривиальным подсчетом контрольной суммы, причем, защита может не только задетектить бряк, но и снять его, восстановив исходное значение "брякнутого" байта "вручную". Бряк, естественно, не сработает, хотя отладчик продолжит подсвечивать брякнутую строку, вводя хакера в заблуждение (хинт: отладчик хранит список точек останова внутри своего тела и не проверяет присутствие INT 03h в отлаживаемом коде).

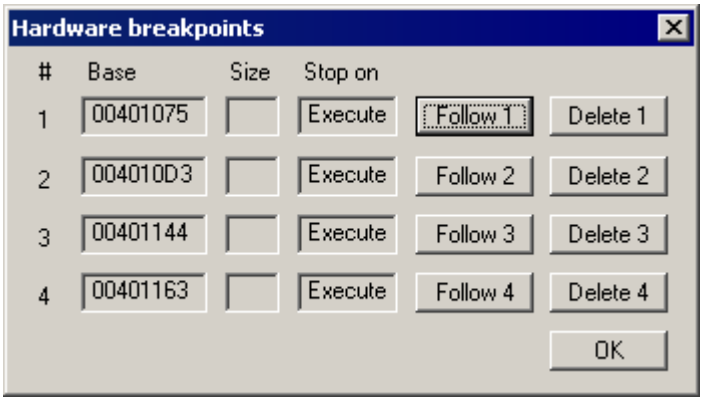

Рисунок 3 в распоряжении хакера имеется только четыре аппаратных точки останова

Многие отладчики (в том числе и Ольга) устанавливают в точку входа (Entry Point) программный бряк, который легко обнаружить из функции DllMain статически прилинкованной динамической библиотеки, возвратив принудительный ноль, что означает "ошибка инициализации" и приводит к аварийному завершению отлаживаемого приложения задолго до того как точка входа получит управление (см. листинг 1).

```
BOOL WINAPI dllmain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
        #define PE_off 0x3C    // PE magic word raw offset
        #define EP_off 0x28    // relative Entry Point filed offset
        #define SW_BP  0xCC    // software breakpoint opcode

        char    buf[_MAX_PATH];
        DWORD   pe_off, ep_off;
        BYTE*   base_x, *ep_adr;

        // obtain exe base address
        GetModuleFileName(0, buf, _MAX_PATH);

        // manual PE-header parsing to find EP value
        base_x = (BYTE*) GetModuleHandle(buf);
        pe_off = *((DWORD*)(base_x + PE_off));
        ep_off = *((DWORD*)(base_x + pe_off + EP_off));
        ep_adr = base_x + ep_off; // RVA to VA

        // check EP for software breakpoint
        // (some debuggers set software breakpoint on EP to get control)
        if (*ep_adr == SW_BP) return 0; // 0 means DLL initialization fails

        return 1;
}
```

Листинг 1 пример, демонстрирующий детекцию отладчика из статически прилинкованной DLL

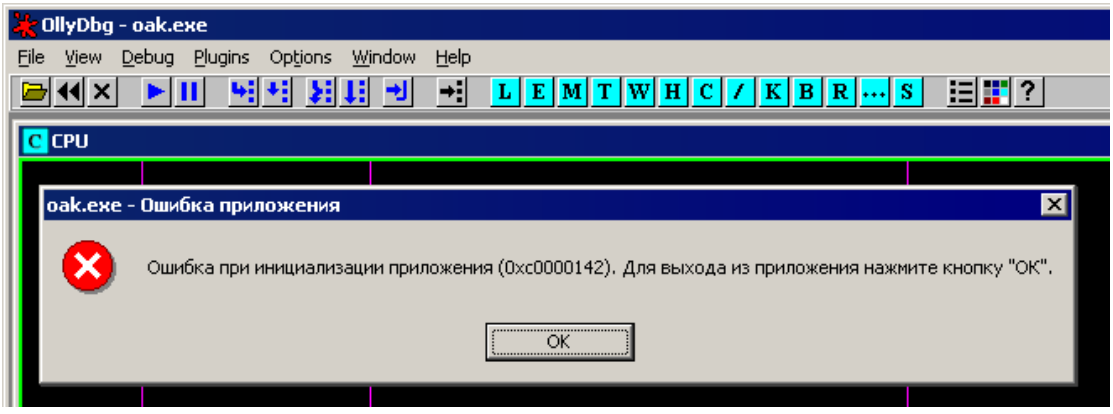

Рисунок 4 Лена обламывает Ольгу (кто такая Лена мыщъх не скажет — попробуйте догадаться сами)

К сожалению, отучить Ольгу ставить бряки в точку входа очень непросто (если вообще возможно) и приходится пускаться на хитрости. Открываем ломаемый exe в HIEW'e и внедряем в точку входа двухбайтовую команду: EBh FEh, соответствующую машинной инструкции l1: jmp short l1, приводящей к зацикливанию программы и дающий нам возможность приаттачить дебаггер к отлаживаемому процессу. Как вариант, можно впендюрить INT 03h во вторую (третью, четвертую…) команду от точки входа, запустив программу "в живую" (т.е. вне отладчика).

Поскольку, исключение, генерируемое INT 03h, некому обрабатывать, операционная система выплевывает сообщение о критической ошибке, предлагая добить апеуху (чтобы та не мучалась) или же запустить JIT (Just-In-Time) отладчик, в роли которого может выступить и Ольга (Options -> Just-In-Time Debugging -> Make Olly just-in-time debuuger). Кстати говоря, подобная техника носит название "Break'n'Enter" и довольно широко распространена. В частности, ее поддерживает PE-TOOLS и многие другие хакерские утилиты.

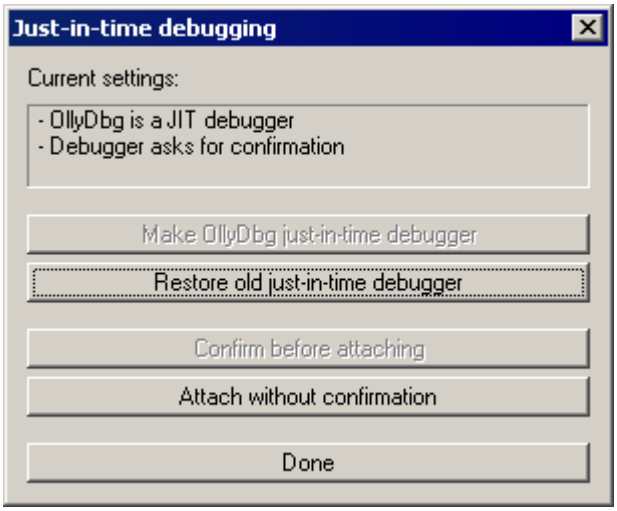

Рисунок 5 превращаем Ольгу в Just-In-Time отладчик

Но вернемся к нашим баранам. Попытка установки программной точки останова на самомодифицирующийся (упакованный или зашифрованный) код ведет к краху, причем безо всякого участия со стороны защиты. Надеюсь, не нужно объяснять почему?! Ну хорошо. Возьмем тривиальный криптор, работающий через byte XOR 66h. Допустим, мы устанавливаем бряк на команду PUSH EAX (50h). После шифровки она превращается в 36h. Представим, что поверх 36h установлена точка останова — CCh. Тогда после расшифровки (CCh XOR 66h) мы получим AAh (STOSB), что, естественно, вызовет крах, т.к. мы не только потеряли PUSH EAX, но еще и регистры ES:EDI смотрят черт знают куда, вызывая ACCESS VIOLATION.

Впрочем, тут возможны детали… Иногда программу расшифровывает не прикладной код, находящийся непосредственно в ломаемой программе, а драйвер защиты, исполняющийся совершенно в другом контексте. Кстати, о контекстах…

контексты — свои и чужие

При работе с самомодифицирующимся (зашифрованным или упакованным) кодом необходимо использовать аппаратные точки останова по исполнению (в скобках заметим, что аппаратная точка останова на чтение/доступ срабатывает только при чтении/записи ячейки, но не при исполнении). Аппаратные точки останова никак не меняют содержимое памяти отлаживаемой программы (а потому не гробят расшифровываемый код) и обнаружить их можно только чтением DRx регистров, что легко отслеживается отладчиком.

В качестве альтернативы Ольга предлагает точки останова на память (Breakpoint -> Memory, on Access/Memory, on Write), реализуемые путем сброса атрибутов соответствующей страницы в PAGE_NOACCESS или PAGE_READONLY, в результате чего при каждом обращении (записи) возбуждается исключение, отлавливаемое Ольгой, которой остается только разобраться к какой ячейке памяти происходит обращение — выполняется ли условие точки останова или нет. Точек останова на память может быть сколько угодно и они так же не гробят расшифровываемый код, а обнаруживаются они только чтением атрибутов страниц, чему легко воспрепятствовать.

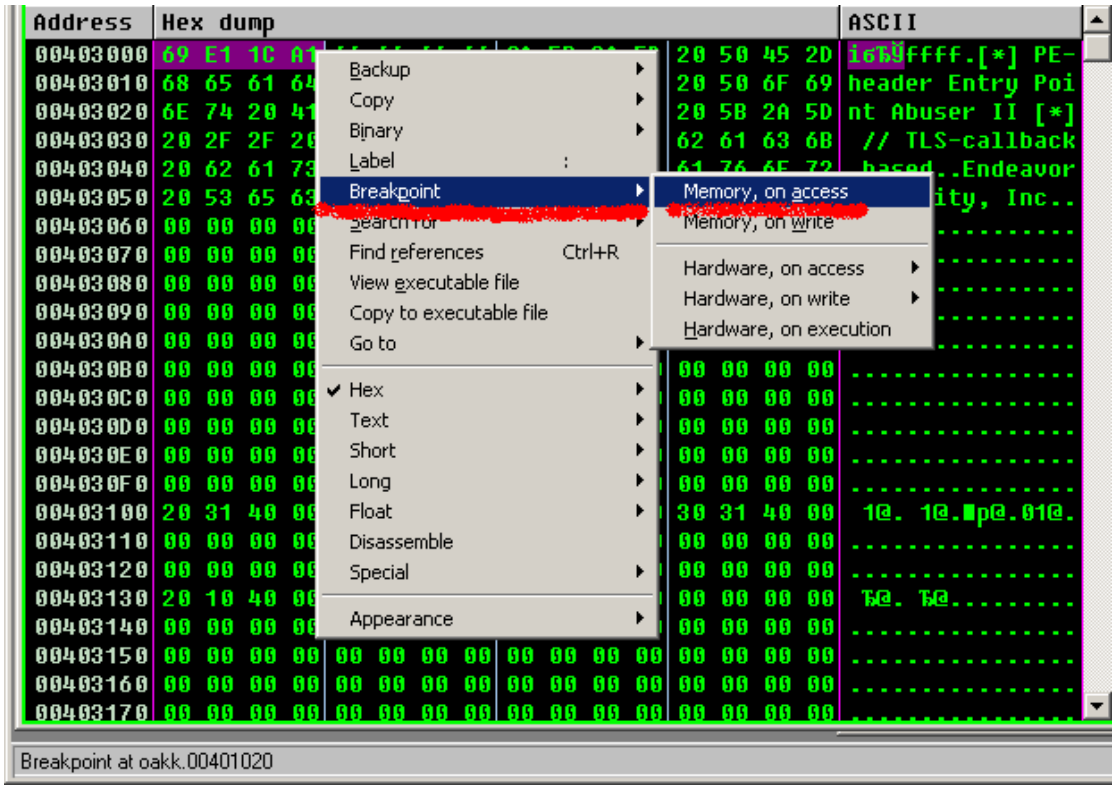

Рисунок 6 точки доступа на память, поддерживаемые Ольгой

Все это, конечно, очень хорошо, вот только при определенных обстоятельствах аппаратные точки (равно как и точки останова на память) не срабатывают или вызывают неожиданный крах приложения и хорошо еще если не обрушивают систему в голубой экран смерти. Как такое может происходить?!

Вернемся к ситуации с драйвером. Как он отреагирует на установленную аппаратную точку останова? А никак не отреагирует. Отладочные регистры хранятся в регистровом контексте процесса и при переходе на ядерный уровень этот контекст неизбежно изменяется, отладочные регистры перезагружаются, и установленных точек останова в них не оказывается. Правда, Soft-Ice поддерживает глобальные точки останова, но… увы… они работают только в W2K и бета-версии Server 2003.

Ситуация с точками останова на память намного более интересная. Существует два (на самом деле три, но это уже детали) способа передачи памяти драйверу. В первом случае операционная система копирует блок памяти в промежуточный регион, а во втором — дает драйверу прямой доступ к адресному пространству прикладного процесса. Точки останова на память никак не воздействуют на отладочные регистры, но изменяют страничные атрибуты, заставляя процессор генерировать исключение при обращении к ним, причем, для корректной работы приложения, отлавливать исключение должен именно отладчик, а не кто-то еще.

Будучи отладчиком прикладного уровня, Ольга просто не в состоянии отлавливать исключения в ядре, в котором они, собственно говоря, и возбуждаются. Для драйвера такой поворот событий оказывается полной неожиданностью. В лучшем случае он возвращает ошибку, в худшем же необработанное ядерное исключение валит систему в BSOD. Так что пользоваться точками останова на память следует с большой осторожность.

Хорошо, а если у нас нет драйвера — тогда что? Достаточно часто встречается ситуация, когда расшифровщик вынесен в отдельный поток. И хотя этот поток выполняется на прикладном уровне, у него имеется свой собственный регистровый контекст, в которой аппаратные точки, естественно, не попадают, а потому, точки останова, установленные в Ольге, не срабатывают, в чем легко убедиться на простом примере (см. листинг 2).

Кстати говоря, если мы попросим Ольгу "всплывать" при загрузке динамических библиотек (что очень полезно для перехвата функций DllMain, выполняющихся еще до передачи управления на точку входа в EXE файл), то "всплывать" Ольга будет в _системном_ контексте, а потому и аппаратные бряки уйдут лесом. В смысле не сработают, т.к. у базового потока совершенно другой контекст.

```
int to_break;                  // DWORD куда мы будем ставить точку останова на доступ

DWORD WINAPI ThreadProc( LPVOID lpParameter)
{
        // аппаратная точка останова в Ольге здесь не срабатывает (в айсе срабатывает!)
        // зато в Ольге срабатывает точка доступа на память, реализуемая через NOACCESS
        to_break = 0x666;
        return (to_break+1);
}

main()
{
        DWORD ThreadId;

        // здесь мы устанавливаем аппаратную точку останова на доступ. она срабатывает!
        int a = to_break;

        // создаем новый поток и обращаемся к переменной to_break оттуда
        CreateThread(0, 0, ThreadProc, 0, 0, &ThreadId);

        Sleep(100);            // даем потоку немного времени, чтобы поработать
        return a;
}
```

Листинг 2 демонстрация "ослепления" аппаратных точек останова путем порождения вспомогательного потока

Точки останова на память, меняющие атрибуты страниц, работают вне контекста потока, в котором они были установлены, а потому если в предыдущем примере на to_break установить точку останова на память, Ольга превосходно это дело засечет. Тоже самое относится и к динамическим библиотекам. Красота!!!

Однако, точки останова на память далеко не всесильны. И они легко обламываются API-функциями ReadProcessMemory/WriteProcessMemory. Тоже самое, впрочем, относится и к аппаратным точкам останова. Почему? Да потому, что ReadProcessMemory/WriteProcessMemory выполняются в ядерном контексте, причем, система игнорирует атрибуты PAGE_NOACCESS и PAGE_READONLY, что и демонстрирует следующий пример (см. листинг 3):

```
main()
{
        int a;

        static to_break;       // DWORD куда мы будем ставить точку останова на доступ
        static to_store;
        static NumberOfBytesRead;

        // здесь мы устанавливаем аппаратную точку останова на доступ. она срабатывает!
        a = to_break;

        // читаем to_break через ReadProcessMemory
        // ни аппаратная точка останова, ни точка останова на память не срабатывают!
        ReadProcessMemory(GetCurrentProcess(),
                &to_break, &to_store, sizeof(to_break), &NumberOfBytesRead);
        return a;
}
```

Листинг 3 ослепление аппаратных точек останова и точек останова на память API-функцией ReadProcessMemory

прыжки в середину команды

Рассмотрим, простой, но невероятно эффективный анти-отладочный примем, высаживающий хакеров на реальную измену (особенно начинающих). Попробуем совершить прыжок в середину команды, но так, чтобы это не сильно бросалось в глаза. Например, так (см. листинг 4).

```
.00401072: 3EFF10              call    d,ds:[eax]      ; // CALL с префиксом DS
…
.004010A8: E8C6FFFFFF          call    .000401073      ; // прыжок в середину команды
…
.004010C9: E8A4FFFFFF          call    .000401072      ; // прыжок в начало команды
```

Листинг 4 борьба с точками останова путем прыжка в середину команды

Допустим, мы установили точку останова на команду "CALL d, DS:[EAX]", которой соответствует опкод 3Eh FFh 10h. Как видно, первым идет префикс DS (3Eh) без которого CALL будет работать так же как и с ним, даже чуть-чуть быстрее. Именно поверх префикса Ольга и записывает CCh при нажатии <F2>, а Soft-Ice делает тоже самое по <F7>.

Команда "CALL 00401073h", расположенная совсем в другом месте программы, пропускает префикс, начиная выполнение непосредственно с FFh 10h. Точка останова при этом, естественно, не срабатывает, однако, чтобы усыпить бдительность хакера защита делает "холостой" вызов "CALL 00401072h", приводящий к "всплытию" отладчика, однако, поскольку предыдущий CALL пропущен, ломать такую защиту можно очень долго.

Самое интересное, что Ольга всячески сопротивляется установке программной точке останова на середину команды и в этом есть свой резон, поскольку, в общем случае (подчеркиваю — в _общем_ случае), программная точка останова, установленная в середину команды, ведет к непредсказуемому поведению процессора, зависящему от структуры опкода конкретной команды.

Тут сильно выручают точки останова на память, которые справляются с подобными ситуациями влет.

когда несколько условий выполняются одновременно

Вопрос на засыпку: что произойдет если установить на команду сразу две точки останова: на доступ и исполнение и оба этих условия сработают одновременно? Конкретный пример показан ниже (см. листинг 5):

```
L1: MOV EAX, d, DS:[L2]
```

Листинг 5 устанавливаем аппаратную точку останова по исполнению на L1 и аппаратную точку останова на L2. вопрос — сколько точек останова сработает?!

Согласно спецификации от Intel, если два или более условий выполняются одновременно, то генерируется одно отладочное исключение, но в статусном регистре DR6 обозначены все сработавшие бряки. Однако, в данном случае, условия точек останова выполняются не одновременно, а последовательно.

Первой срабатывает точка останова по исполнению команды "MOV EAX, d, DS:[L2]", при этом регистр EIP указывает на L1. Другими словами, отладочное прерывание генерируется _до_ выполнения команды когда к метке L2 никто и не думал обращаться. Логично, что бряк, установленный на L2, должен сработать сразу же после первого.

Должен. Но… он не срабатывает. Никогда. Почему?! А потому что разработчики отладчиков думают не головой, а… Ладно, оставим наезды и засядем за чтение технической литературы. В смысле мануалов, откуда мы быстро узнаем, что процессоры поддерживают специальный Resume Flag (он же #RF), хранящийся в регистре флагов EFLAGS и подавляющий генерацию отладочного исключения на время выполнения следующей машинной команды.

Для чего он нужен? А вот для чего! После срабатывания точки останова по исполнению, регистр EIP указывает на L1 и возобновлении выполнения данной команды, мы снова словим отладочное исключение и регистр EIP как и прежде будет указывать на L1. Чтобы разорвать этот заколдованный круг, отладчик взводит #RF-флаг, подавляя отладочные исключения, генерируемые текущей исполняемой командой.

В процессе выполнения инструкции "MOV EAX, d, DS:[L2]" срабатывает точка останова на доступ к L2, но… отладочное исключение не генерируется, отладчик не всплывает и хакер остается в глухом подвале.

Как это можно использовать на практике?! Да элементарно! Если L2 указывает на пароль/серийный номер, достаточно вынудить хакера установить _аппаратную_ точку останова по исполнению на L1, тогда аппаратная же точка останова на L2 не сработает и ее проворонят. А чтобы не проворонить необходимо комбинировать аппаратные точки останова с программными или с точками доступа на память.