старые антиотладочные приемы на новый лад

крис касперски ака мыщъх

новое, как говориться, это хорошо забытое старое. антиотладочные приемы времен ms-dos и debug.com возвращаются, чтобы снова работать в Windows 2000/XP/2003. статья не только вызовет у вас ритуально-ностольгический интерес, но и послужит практическим пособием по борьбе с хакерами и отладчиками.

введение

Отладчики (а отладчик, как известно, — основной инструмент хакера) прошли долгий эволюционный путь — от debug.com до soft-ice. За это время было найдено множество способов борьбы с ними, однако, не все они оказались удачны. В последнее время наблюдается острый дефицит хороших антиотладочных приемов. Так не пора ли вернутся к истокам — древним приемам, проверенных годами? Технический прогресс развивается по спирали и методы защиты, разработанные для ms-dos и утратившие актуальность к концу ее существования, оказывают убийственное воздействие на Windows-отладчики, которые просто не предусматривали такого поворота событий и капитулируют сразу, даже не пытаясь бороться.

Разумеется, непосредственный перенос антиотладочных методик из ms-dos в Windows невозможен хотя бы уже потому, что в Windows нет "прерываний" в том смысле, который в них вкладывает ms-dos. В ней нет портов ввода/вывода, физической памяти и множества других привычных для ms-dos программистов концепций и вещей. То есть, они, разумеется, есть (куда же без прерываний?!), но техника работы с ними радикально отличается. Так что, определенная адаптация все-таки необходима.

Рисунок 1 древнейший отладчик debug.com

Основной упор мы сделаем на прикладной уровень и программистов средней квалификации, которые, возможно, имеют представление о драйверах, но разрабатывать их самостоятельно не собираются. Некоторые из описанных методик практически полностью системно-независимы и работают под любой 32-разрядной операционной системой семейства Windows, частично поддерживания Windows 3.x и Windows XP 64-bit edition. Некоторые требуют только NT или производную от нее систему, причем, никаких гарантий сохранения работоспособности в последующих версиях Windows у нас нет. Даже если использовать только документированные функции, Microsoft в любой момент может их изменить, как это она уже делала неоднократно, причем, в последнее время все чаще и чаще. Лучше вообще не использовать никаких системно-зависимых функций и писать все на ANSI Си/C++ или DELPHI (фактически, DELPHI представляет собой самостоятельную "операционную систему", натянутую поверх Windows и абстрагирующую программиста от прихотей Microsoft — вы просто перекомпилируете код и все! разумеется, речь идет только о прикладных задачах и на антиотладочные приемы указанное преимущество DELHPI не распространяется). Тем не менее, бояться антиотладочных приемов ненужно. Их использование вполне допустимо, если не сказать — желательно.

Хороший сборник антиотладочных приемов глубокой древности можно найти в статье "Anti Debugging Tricks" (http://textfiles.group.lt/programming/antidbg.txt), написанной известным вирусописателем Inbar Raz (что? никаких ассоциаций? эх... молодежь!) датируемой 1993 годом, но сами описанные в ней приемы имеют намного более почетный возраст и многие из них пришли в ms-dos с еще более древних машин. Попытаемся перенести их на Windows?

программа, которая трассирует себя сама

Один из древнейших приемов основан на "поглощении" трассировочного прерывания отладчиком. Рассмотрим его поподробнее. Как известно, x86-процессоры имеют специальный trap-флаг (или, сокращенно TF). Когда он взведен, после выполнения каждой инструкции генерируется специальное отладочное прерывание, обрабатываемое отладчиком (если, конечно, он установлен и действительно отлаживает программу). В противном случае управление получает обработчик по умолчанию. В ms-dos он представляет собой "холостую" заглушку, состоящую из одного лишь IRET, и выполнение программы не прерывается, а вот Windows

возбуждает исключение, приводящее к появлению хорошо всем известного диалогового окна "программа совершила недопустимую операцию и будет закрыта".

Формально, флаг трассировки является частью регистра флагов и потому может быть сохранен в стеке командной pushf/pushfd, однако, распознать факт трассировки таким способом скорее всего не удастся, поскольку многие отладчики маскируют свое присутствие. При пошаговом выполнении программы они осуществляют дополнительную проверку — а не является ли только что выполненная команда инструкций pushf и если это так — лезут в стек и корректируют сохраненное значение регистра флагов, сбрасывая TF в ноль. Правда, тут можно схитрить. Например, добавить несколько лишних префиксов перед командой pushf — есть шанс, что отладчик не сможет их распознать. Еще можно использовать самомодифицирующийся код, расположив команду pushf в стеке с таким расчетом, чтобы сохраненное значение регистра флагов затирало ее напрочь, правда тут есть одна маленькая проблема. Производители аппаратно-программного обеспечения наконец-то признали тот факт, что научить программистов программированию им так и не удалось, а теперь уже и не удастся. Операционные системы и приложения буквально нашпигованы ошибками переполнения буфера, через которых распространяются черви, троянские программы и другая зараза. Чтобы прикрыть лазейку пришлось пойти на довольно жесткую, спорную и радикальную меру — неисполняемый стек. И хотя распространения червей он все равно не остановит, ведь существуют и другие типы переполняющихся буферов (множество статей на эту тему лежит на ftp://nezumi.org.ru), для нас (пользователей и программистов) в практическом плане это означает, что огромное количество программ перестанет работать, поскольку они защищены упаковщиками и протекторами, использующими самомодифицирующийся код, исполняющийся в стеке. Поэтому, Microsoft предусмотрела возможность отключения защиты. При первой попытке исполнения кода на стеке, Windows XP выбросит грозное диалоговое окно (см. рис. 2) и при утвердительном ответе пользователя выполнение защищенной программы будет продолжено, так все не так уж и плохо.

Рисунок 2 диалоговое окно, появляющееся при попытке выполнения кода в стеке

Впрочем, отладчики типа Microsoft Visual C++ дурить необязательно и следующий бесхитростный код легко обнаруживает их присутствие. Попробуйте его выполнить в пошаговом режиме и посмотрите, что произойдет!

```
pushf                   ; сохраняем флаги в стеке, включая и TF
pop eax                 ; выталкиваем сохраненные флаги в eax
and eax, 100h           ; выделяем флаг трассировки
jnz under_debugger      ; если TF взведен, нас трассирую
```

Листинг 1 простейший анти-отладочный прием, распознающий трассировку под некоторыми отладчиками

Главный минус этого приема в том, что его очень легко обойти. Достаточно, например, просто подогнать курсор к команде pushf, сказать отладчику "HERE" (т. е. выполняй программу до этого места без трассировки), затем подогнать курсор к jnz и сказать "HERE" еще раз. Таким образом, защищенный фрагмент будет исполняться в обычным режиме и присутствие отладчика окажется незамеченным, поэтому, многие программисты предпочитают сохранять регистр флагов в одном месте программы, а проверять его в другом. Начинающих хакеров это сбивает с толку, но опытных так не проведешь. На конструкцию pushf/pop reg/xxx reg,100h у них уже давно выработался безусловный рефлекс, к тому же взломщику ничего не стоит заменить and eax,100h на and eax,0h и тогда программа навсегда утратит способность распознавать отладчик. Можно (и нужно), конечно, добавить проверку собственной целостности, но только навряд ли она надолго остановит хакера.

А вот слегка модифицированный вариант той же самой защиты, который распознает присутствие отладчика независимо от того исполняется ли программа в пошаговом режиме или нет. Алгоритм работы в общих чертах выглядит так: мы самостоятельно взводим флаг трассировки и выполняем следующую команду. Процессор послушно генерирует исключение, которые мы перехватываем предварительно установленным SEH-обработчиком, передающим управление нашему коду. Но при наличии отладчика, исключение "поглощается" и SEH-обработчик уже не получает управления!

```
; устанавливаем новый обработчик структурных исключений
xor     eax,eax                 ; обнуляем регистр eax
```

```
        push    offset SEH_handler     ; кладем в стек указатель на новый обработчик
        push    dword ptr fs:[eax]     ; кладем в стек указатель на старый обработчик
        mov     fs:[eax],esp           ; регистрируем новый SEH-обработчик

        ; взводим флаг трассировки
        pushf                          ; заталкиваем в стек регистр флагов
        pop eax                        ; выталкиваем его содержимое в регистр eax
        or ah, 1                       ; взводим флаг TF
        push eax                       ; кладем eax в стек
        popf                           ; выталкиваем его содержимое в регистр флагов
                                       ; теперь флаг трассировки взведен!

        jmp under_debugger             ; после выполнения этой команды генерируется
                                       ; исключение и если отладчик не установлен,
                                       ; его перехватывает SEH-обработчик, который
                                       ; корректирует EIP и эта команда не выполняется
                                       ; под отладчиком происходит переход на ветку
                                       ; under_debugger
        //
        ...                            ; основной код программы
        //

        ; SEH-обработчик. может быть расположен в любом месте
        ; (лучше расположить его подальше от защитного кода,
        ; чтобы он не так бросался в глаза)
SEH_handler:
        mov esi, [esp+0ch]     ; указатель на контекст регистров
        assume esi: PTR CONTEXT
        mov [esi].regEip, offset continue
                                       ; откуда продолжать выполнение
                                       ; в отсутствии отладчика
        xor eax, eax
        ret                            ; выход из SEH-обработчика

continue:
        ; // отсюда будет продолжено управление, если отладчик не установлен
```

Листинг 2 универсальный антиотладочный прием распознающий трассировку под большинством отладчиков

Данный прием позволяет обнаруживать soft-ice и многие другие отладчики, причем, на пассивную отладку он не реагирует (и это хорошо!). Пользователь нашей программы может запускать soft-ice и отлаживать остальные программы, но... как только он попытается загрузить в отладчик нашу она тут же подаст сигнал. Тоже самое произойдет, если хакер присоединит отладчик к уже запушенному процессу или ворвется в середину программы путем установки точек останова на API-функции. Естественно, антиотладочный код должен выполняться не до, а после загрузки отладчика. То есть, размещать его в самом начале защищаемой программы не стоит. Лучше — многократно дублировать в различных местах.

Прелесть данного приема в том, что его достаточно трудно распознать при взломе. Явные проверки отсутствуют и команда jmp under_debugger выглядит невинной овечкой. При ее выполнении без отладчика возбуждается исключение, перехватываемое обработчиком и выполнение программы идет уже совсем по другому пути (обработчик подменяет указатель команд, хотя, впрочем, он мог бы этого и не делать — в отличии от прерываний из которых в ms-dos нужно было выходить как можно скорее, чтобы не развалить систему, обработчик структурного исключения в принципе может вмещать в себя весь код программы целиком и химичить с контекстом совершенно необязательно — зачем лишний раз привлекать внимание хакера?). Под отладчиком же команда jmp under_debugger выполняется "как есть" и хакер может очень долго ковыряться в подложной ветке under_debugger не понимая, что здесь вообще происходит и откуда это взялось?! Чтобы оттянуть взлом, лучше не говорить сразу, что отладчик обнаружен, а подсунуть какой-нибудь зашифрованный код или что-то еще.

Главное, чтобы команды popf и jmp under_debugger не были разделены никакими другими инструкциями! Иначе защита не сработает! Трассировочное исключение генерируется сразу же после выполнения первой команды, расположенной после popf, и если ею окажется, например, nop, то jmp'у никаких исключений уже не достанется. Так что замаскировать защитный код, рассосредоточив его по всему оперативному периметру уже не удастся. К тому же, хакер может легко нейтрализовать защиту, просто заменив jmp under_debugger на jmp continue. Чтобы этому противостоять, необходимо взводить в SEH-обработчике специальный флаг и проверять его по ходу выполнения программы — был ли он вызван или нет. А в самом SEH-обработчике еще контролировать и тип исключения, иначе хакер просто

добавит xor eax,eax/mov [eax],eax (обращение по нулевому указателю, генерирующее исключение) и тогда SEH-обработчик будет получать управление как под отладчиком так и без него. Кстати говоря, защита данного типа была использована в программе ulink Юрия Харона, взлом которой подробно описан в моих "Фундаментальных основах хакерства — записках мыщъх'а", которую можно бесплатно скачать с моего ftp-сервера nezumi.org.ru (напоминаю, что он доступен не все время, а только когда мыщъх шевелит хвостом).

Зададимся таким вопросом: может ли отладчик трассировать программу, которая уже находится под трассировкой (например, отлаживается другим отладчиком или трассирует сама себя). Непосредственно — нет, поскольку флаг трассировки в x86 процессорах один и его вложенность не поддерживается. С другой стороны, если вышестоящий отладчик отлеживает обращение к флагу трассировки и эмулирует возбуждение трассировочного прерывания, передавая управление не на следующую команду на SEH-обработчик, такая схема может работать, правда, код отладчика существенно усложнится, а, поскольку, коммерческие отладчики ориентированы исключительно на легальных программистов, которые взломом защит не занимаются (или, во всяком случае, предпочитают это не афишировать), то создавать идеальный отладчик производителю просто не резон. Что же до некоммерческих отладчиков… При всем моем уважении к ним (я часто использую OllyDbg и люблю его), они еще не встали с горшка и для достижения совершенства им еще расти и расти. Впрочем, эмулирующие отладчики с такой защитой справляются на раз, но где вы видели эмулирующий отладчик под Windows? Можно, конечно, взять полноценный эмулятор PC с интегрированным отладчиком типа BOCHS (под который, кстати говоря, существуют и дополнительные отладчики, входящие в исходные тексты, но отсутствующие в готовой бинарной сборке), однако, отлаживать на нем Windows-приложения практически нереально, поскольку нет никакой возможности отличить код одного процесса от другого.

Программа, которая трассирует себя, под отладчиками выполняется неправильно — она трассируется отладчиком, но не сама собой. Хорошая идея — повесить на трассер распаковщик, расшифровывающий программу по мере ее выполнения. Это существенно затрудняет взлом, зачастую делая его практически невозможным. Вместо явной или косвенной проверки на отладчик, программа задействует общие с отладчиком ресурсы и под отладчиком становится просто нефункциональна.

Простейший пример такой защиты приведен ниже. Он "позаимствован" из моей книги "Техника и философия хакерских атак" (полную электронную версию которой можно бесплатно скачать с ftp://nezumi.org.ru) и работает только под ms-dos, тем не менее легко переносится в Windows, просто обработчик прерывания заменяется на обработчик структурного исключения (как именно это делается показано в листинге 2) и перед расшифровкой вызывается API-функция VirtualProtect для установки атрибута записи (по умолчанию секции .text/.code и .rodata имеют атрибут read-only и непосредственная расшифровка кода в них невозможно). Windows-вариант не сложен в реализации, но слишком громоздок и потому ненагляден.

```
        ; // устанавливаем новый обработчик трассировочного прерывания int 01h
        mov ax,2501h            ; функция 25h (установить прерывание), прерывание — 01h
        lea dx,newint01h        ; указатель на обработчик прерывания
        int 21h                 ; сервисная функция ms-dos

        ; // взводим флаг трассировки
        pushf                   ; сохраняем регистр флагов в стеке
        pop ax                  ; выталкиваем его в регистр ax
        or ah,1                 ; взводим бит TF
        push ax                 ; сохраняем измененный регистр ax в стеке
        popf                    ; выталкиваем модифицированное значение в регистр флагов

        // теперь после выполнения каждой команды процессор будет генерировать int 01,
        // передавая управление его обработчику

        // подготавливаем параметры для расшифровки
        lea si, crypted_begin
        mov cx, (offset crypted_end - crypted_begin) / 2

repeat: ; // основной цикл расшифровки
        lodsw                   ; читаем очередное слово по si в ax, увеличивая si на 2
        mov [si-2],bx           ; записываем в ячейку [esi-2] содержимое bx
        loop repeat             ; крутим цикл, пока cx не равен нулю
                                ; кажется, что это дурной цикл, работящий как memset
                                ; т.е. заполняющий область памяти содержимым bx
                                ; которое даже не было инициализировано!
                                ; однако, все не так и на каждом ходу генерируется
```

```
                                ; трассировочное прерывание, передающее управление
                                ; обработчику int 01h, который неявно модифицирует bx

        ; // сбрасываем флаг трассировки
        pushf                   ; сохраняем регистр флагов в стеке
        pop dx                  ; выталкиваем его в регистр dx
                                ; (ax используется обработчиком int 01h)
        and dh,0FEh             ; сбрасываем бит TF
        push dx                 ; сохраняем измененный регистр dx в стеке
        popf                    ; выталкиваем модифицированное значение в регистр флагов

        ; // еще одна ловушка для хакера
jmp_to_dbg:
        jmps under_debugger

        //
        ...                     ; "полезная нагрузка" (основной код программы)
        //

new_int_01h:
        xor ax, 9fadh           ; зашифровываем содержимое регистра ax
        mov bx, ax              ; помещаем его в регистр bx
        mov word ptr cs:[jmp_to_dbg],9090h
                                ; "убиваем" условный переход, ведущий к подложной ветке
                                ; (под отладчиком обработчик int 01h не получит
                                ; управления и переход не будет убит)
        iret                    ; выходим из обработчика прерывания

crypted_begin:

        //
        ...                     ; зашифрованный код/данные
        //

crypted_end:
```

Листинг 3 пример простейшей самотрассирующейся программы под ms-dos

Можно ли это взломать? Если честно, то данный пример ломается без особого напряжения и с минимумом телодвижений. Просто устанавливаем аппаратную точку останова на crypted_begin и дожидаемся завершения распаковки, после чего снимаем дамп, превращаем его в exe-файл, который уже можно отлаживать обычным путем. Чтобы не дать себя обмануть защита должна использовать множество вложенных расшифровщиков или бороться с отладчиком иным путем (например, убивать его через доступ к физической памяти, о чем мы поговорим чуть ниже).

Главное достоинство описанного приема в том, что он хорошо чувствует себя под всей линейкой Windows (как 9x, так и NT) и маловероятно, чтобы в последующих версиях что-нибудь изменилось.

доступ к физической памяти

В ms-dos код отладчика и системные данные (например, таблица векторов прерываний) находились в одном адресном пространстве с отлаживаемой программой, что открывало большой простор для методов борьбы. Можно было проскандировать память и убить отладчик или просто задействовать отладочные вектора (int 01h и int 03h) под нужны защитного механизма, положив туда что-то полезное (скажем, ключ расшифровки), а через некоторое время считать его обратно. Если никакого отладчика нет или он неактивен, искажение отладочных векторов не нарушает работоспособности операционной системы и мы читаем то, что поклали. А вот под отладчиком все будет иначе. Скорее всего произойдет тотальный крах, поскольку при генерации трассировочного прерывания или достижении точки останова, управление будет передано в "космос" (вектор ведь искажен!). Если же отладчик принудительно восстановит вектора, тогда вместо сохраненных данных, защищенная программа прочитает уже не ключ расшифровки, а нечто совершенно иное! Теоретически, на 386+ процессорах отладчик может контролировать доступ к отладочным векторам и эмулировать операции чтения/записи, не производя их на самом деле. Но это потребует двух аппаратных точек останова, которых в x86 процессорах всего четыре, да и тех постоянно не хватает, так что разбрасывается ими не резон.

Операционная система Windows использует раздельные адресные пространства и виртуальную память, а это значит, что отлаживаемое приложение не может "дотянуться" ни до

отладчика, ни до векторов прерываний. Можно, конечно, написать драйвер (как известно, драйвер может все или практически все), но о сопутствующих проблемах мы уже говорили. Написание драйверов требует высокой квалификации, к тому же в силу их крошечного размера, драйвера очень просто ломать, а написать сложный драйвер практически нереально — его "пуско-наладка" займет всю оставшуюся жизнь.

В операционных системах семейства NT имеется специальное псевдоустройство "PhysicalMemory", предоставляющее доступ к физической памяти на чтение/запись. Это действительно физическая память, причем еще до виртуальной трансляции и в ней есть все то, что находится в памяти компьютера в данный момент. Страниц, выгруженных на диск в файл подкачки, там нет, но нам их и не надо. Нам нужен код и данные операционной системы.

При наличии прав администратора мы можем не только читать "PhysicalMemory", но и писать. Да-да! Вы не ослышались и это не опечатка! С прикладного уровня можно проникнуть в святая святых операционной системы, свободно модифицируя код, исполняющийся на привилегированном уровне нулевого кольца (RING0), а это, значит, что мы фактически имеем RING0 на прикладном кольце (RING3)! Любой отладчик может быть уничтожен без проблем, даже если это отладчик-эмулятор. Исключение составляют лишь виртуальные машины типа BOCHS, но, как мы уже говорили, пытаться отладить Windows приложение на них несерьезно. Хакер утонет в посторонних потоках и системных вызовах!

Некоторые считают PhysicalMemory ужасной дырой в безопасности. Дескать, как это так — с прикладного уровня и сразу в дамки! На самом деле здесь нет ничего ненормального. Если у хакера есть права администратора, он может беспрепятственно загружать драйвера, из которых можно делать все, что угодно, причем, в NT загрузка драйвера не требует перезагрузки, а это, значит, что наличие псевдоустройства "PhysicalMemory" никак, я повторяю НИКАК, не отражается на безопасности, а всего лишь делает доступ к физической памяти более комфортным и удобным. В UNIX (модель безопасности которой вырабатывалась годами) издавна существуют псевдоустройства /dev/mem и /dev/kmem предоставляющие доступ к физической памяти до и после трансляции, но никто не собирается их критиковать. Эта функция нужна достаточно многим системным программам и самой операционной системе, если ее изъять программисты начнут писать свои драйвера, предоставляющие доступ к физической памяти, и тогда начнется полный разброд и распад, поскольку в них не исключены всякого рода ошибки. Разработчик драйвера может забыть о проверке уровня привилегий и давать доступ не только администраторам, а всем пользователям системы, что будет нехорошо. Тем не менее, Microsoft все-таки пошла на довольно-таки спорный шаг и в Windows 2003 Server с установленным Service Pack 1 доступа к PhysicalMemory уже не имеет ни администратор, ни даже System. (см. статью "Changes to Functionality in Microsoft Windows Server 2003 Service Pack 1 Device\PhysicalMemory Object" на сайте Microsoft: www.microsoft.com/technet/prodtechnol/windowsserver2003/library/BookofSP1/e0f862a3-cf16-4a48-bea5-f2004d12ce35.mspx).

Тем не менее, на всех остальных системах данный прием работает вполне нормально, поэтому не стоит списывать его со счетов, а с Windows 2003 Sever SP1 мы еще разберемся! (Не сейчас, в смысле еще не в этой статье, но как-нибудь потом).

Давайте возьмем утилиту objdir из пакета NT DDK и запустим ее с параметром "\Device" (просмотр устройств и псевдоустройств, установленных в системе). Вот что она скажет:

```
$objdir \Devic
…
ParTechInc0             Device
ParTechInc1             Device
ParTechInc2             Device
PfModNT                 Device
PhysicalMemory          Section
PointerClass0           Device
Processor               Device
RasAcd                  Device
RawCdRom                Device
```

Листинг 4 просмотр списка установленных (псевдо)устройств утилитой objdir

Как видно, PhysicalMemory это не совсем устройство, точнее совсем не устройство (Device), а секция (Section), поэтому для работы с ней следует использовать функцию NtOpenSection. Пример простейшей реализации такого вызова приведен ниже:

```
// разные переменные
NTSTATUS ntS; HANDLE Section; OBJECT_ATTRIBUTES ObAttributes;
INIT_UNICODE(ObString,L"\\Device\\PhysicalMemory");

// инициализация атрибутов
InitializeObjectAttributes(&ObAttributes, &ObString,
                           OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

// открываем секцию PhysicalMemory
ntS = NtOpenSection(&Section, SECTION_MAP_READ|SECTION_MAP_WRITE, &ObAttributes);
```

Листинг 5 открытие псевдоустройства PhysicalMemory

Возможно, мы захотим предоставить доступ к псевдоустройству PhysicalMemory не только администратору, но и всем другим пользователям. Для этого необходимо изменить права. Естественно, делать это может только администратор, зато потом физическую память будут трогать все кому не лень. Многие вирусы и root-kit'ы именно так и поступают. Они прописывают себя в автозагрузке (или любым другим путем закрепляются в системе) и ждут пока пользователь хотя бы разочек не зайдет в систему под администратором. После этого они меняют права на PhysicalMemory и дальше работают уже из-под пользователя. Легальные программы, в частности защитные механизмы, зачастую поступают так же. Требуя администратора лишь на стадии установки они, тем не менее, успевают за это время существенно ослабить безопасность системы. Ну что тут можно сказать? Никогда не устанавливайте на компьютер программ, которым не вполне доверяете. То есть, не устанавливайте никаких программ вообще. Шутка. Ладно, не будем отвлекаться, а лучше рассмотрим какие шаги необходимо предпринять для изменения атрибутов доступа.

- открываем \Device\PhysicalMemory, используя NtOpenSection, получаем дескриптор
- извлекаем из него дескриптор безопасности (security descriptor) через GetSecurityInfo
- добавляем права на чтение/запись в текущий ACL с помощью функции SetEntriesInAcl
- обновляем дескриптор безопасности посредством вызова SetSecurityInfo
- закрываем дескриптор, возвращенный NtOpenSection

Теперь физическую память можно читать и писать. Для этого секцию PhysicalMemory нужно спроецировать на виртуальное адресное пространство, вызвав NativeAPI-функцию NtMapViewOfSection. Ниже показано как это сделать:

```
// переменные-аргументы
HANDLE Section = xxx; // ← входной параметр дескриптор секции
PVOID vAddress = xxx; // ← входной параметр виртуальный адрес куда проецировать
DWORD Size     = xxx; // ← кол-во байт для проецирования от начала секции

// прочие переменные, инициализированные программой
PHYSICAL_ADDRESS pAddress; NTSTATUS ntS; DWORD MappedSize; PVOID MappedAddress=NULL;

// ВНИМАНИЕ! ФУНКЦИИ __GetPhysicalAddress НЕ СУЩЕСТУЕТ В ПРИРОДЕ!
// ОНА ДАНА ЧИСТО УСЛОВНО. НИЖЕ ПО ТЕКСТУ БУДЕТ ОБЪЯСНЕНО ПОЧЕМУ
pAddress = __GetPhysicalAddress((PVOID) vAddress);

// проецируем секцию PhysicalMemory на виртуальное адресное пространство
ntS = NtMapViewOfSection(Section, (HANDLE) -1, &MappedAddress, 0L, Size,
                                    &pAddress, &MappedSize, 1, 0, PAGE_READONLY);
```

Листинг 6 проецирование физической памяти на виртуальное адресное пространство

Единственная проблема — это трансляция адресов (перевод физических адресов в виртуальные и наоборот). NT предоставляет доступ только к физической памяти до трансляции, а, поскольку, физическая память используется как кэш, то одни и те же физические страницы в разное время могут отображаться на различные виртуальные адреса многих адресных пространств. "Официально" трансляцией занимается функция MmGetPhysicalAddress, доступная только на уровне ядра, что нас, — прикладников, — естественно, не устраивает. Тем не менее, ситуация вовсе не так безнадежна, как это кажется на первый взгляд.

Для большинства задач искать соответствие между физическими и виртуальными адресами вообще не нужно! Ведь существуют же сигнатуры! Достаточно создать банк сигнатур всех популярных отладчиков после чего их обнаружение не станет проблемой, а большинство служебных структур типа таблицы дескрипторов прерываний (IDT) вообще работают с физическими адресами. Найти их путем сканирования PhysicalMemory — не проблема. После

чего останется только поиздеваться над прерываниями int 01h и int 03h (или над их обработчиками). Прием, конечно, грубый и убивающий любые отладчики независимо от того отлаживают ли они нашу программу или нет. Правильные программисты так не поступают! Защитные механизмы не должны, просто не имеют морального (и отчасти даже юридического) права мешать отладке посторонних программ, но... Зачастую, они все-таки мешают. Так что сказанное надо воспринимать не как руководство к действию, а как пособие по ремонту чужих защит. О взломе в данном случае никто и не говорит. Почему я, — легальный пользователь защищенной программы, заплативший за нее деньги — не могу держать ко компьютере soft-ice?! Почему я должен каждый раз перезагружать компьютер для работы с программой?!

Впрочем, все это лирика. В некоторых случаях (например, для борьбы с отладчиками прикладного уровня) выполнять трансляцию все-таки необходимо. Это не так-то просто сделать! Статья "Playing with Windows /dev/(k)mem" из PHRACK'a (http://www.phrack.org/phrack/59/p59-0x10.txt) дает некоторые зацепки, частично решающие проблемы, но до полной победы еще далеко.

Проще всего транслируются адреса из диапазона 80000000h:A0000000h. Перевод виртуального адреса в физический осуществляется путем наложения маски 1FFFF000h, однако, начиная с адреса 877EF000h это правило срабатывает не всегда. Адреса < 80000000h и > A0000000h уже не могут быть транслированы таким путем, хотя с некоторой вероятностью маска FFFF000h все-таки срабатывает и простейший вариант реализации функции __GetPhysicalAddress выглядит так:

```
PHYSICAL_MEMORY MyGetPhysicalAddress(void *BaseAddress)
{
        if (BaseAddress < 0x80000000 || BaseAddress >= 0xA0000000)
        {
                return(BaseAddress & 0xFFFF000);
        }

        return(BaseAddress & 0x1FFFF000);
}
```

Листинг 7 простейший (но ненадежный) алгоритм трансляции адресов

Его можно встретить в некоторых вирусах и root-kit'ов (когда вы столкнетесь с ним при дизассемблировании вы будете знать, что это такое), однако, в легальных программах (особенно коммерческих!) его использование категорически недопустимо и потому необходимо либо все-таки писать свой драйвер, вызывающий MmGetPhysicalAddress из режима ядра, либо воспользовавшись тем фактом, что адреса из диапазона 80000000h: 877EF000h транслируются однозначно, внедрить в операционную систему специальный "жучок". То есть, фактически создать в ней свой собственный call-gate, позволяющий вызывать ядерные функции с прикладного уровня. Один из вариантов его реализации приведен в вышеупомянутой статье, однако, он не свободен от ошибок и на многопроцессорных машинах (которыми, в частности, являются все машины с материнской платой и процессором Pentium-4 с технологией Hyper-Threading, не говоря уже о многоядерных AMD), возможны "синие экраны смерти", которые опять-таки недопустимы...

Означает ли это, что данный антиотладочный прием полностью бесполезен? Вовсе нет! Трудности создания устойчивой и надежно работающей программы на его основе носят технический характер и вполне преодолимы. Если не трогать трансляцию, то никаких проблем вообще не возникает!

>>> врезка как работает Win2K/XP SDT Restore

Хорошим примером использования доступа к физической памяти "честными" программами является утилита SDT Restore, копию которой можно бесплатно скачать с http://www.security.org.sg/code/sdtrestore.html. Как и следует из ее названия, она занимается тем, что восстанавливает SDT – Service Description Table, содержащую указатели на системные вызовы, которые могут быть перехвачены злоумышленником для сокрытия своего присутствия в системе, то есть стелсирования. Многие root-kit'ы так и делают. Они подменяют оригинальные вызовы на свои, и система теряет способность обнаруживать создаваемые ими файлы, процессы, сетевые соединения и т. д.

В ms-dos для борьбы со Stealth-вирусами обычно прибегали с системной дискете, но те времена уже давно прошли и хотя Windows в принципе можно загрузить и с лазерного диска (например, Windows PE) или с дополнительно винчестера, такой подход не слишком-то удобен,

хотя бы уже потому, что требует перезагрузки, а сервера лучше не перезагружать. Проще (хотя и не столь надежно) восставить SDT через PhysicalMemory. И хотя root-kit может легко отследить обращение к ней (для этого ему достаточно перехватить NtOpenSection), все известные мне зловредные программы этим еще не занимаются и потому они могут быть легко обезврежены. Во всяком случае пока.

Подробнее о методах маскировки и борьбы можно прочитать в статье "Hide'n'Seek — Anatomy of Stealth Malware" (http://www.blackhat.com/presentations/bh-europe-04/bh-eu-04-erdelyi/bh-eu-04-erdelyi-paper.pdf).

заключение

Мы "воскресили" только два типа анти-отладочных приемов, а всего их... Подробное описание займет целую книгу, если не больше. Но дело даже не в этом. Основная сила защитных механизмов в засекреченности (или малоизвестности) алгоритмов их работы. В отличии от криптографии, где стойкость шифра в основном определяется стойкостью ключа, защитить программный код таким методом невозможно. Железо не позволяет! Тут действует такое правило: коль скоро программу можно запустить, можно ее и взломать.

Антиотладочные приемы, описанные в доступной литературе, с который может ознакомиться всякий хакер, лучше не применять. Да, они по-прежнему затрудняют взлом и знание факта, что программа занимается самотрассировкой еще не позволяет эту трассировку обойти. Скорее всего взломщику потребуется написать собственный отладчик, а на это требуется время. Тем не менее, если даже самый стойкий антиотладочный прием получает широкое распространение и встречается во множестве программ, разработка взламывающего инструментария становится психологически и экономически оправданной. Хакеры садятся за клавиатуры и начинают его писать.

Согласен, информация должна быть открыта. В засекречивании антиотладочных приемов нет никакой необходимости, тем не менее, заниматься их поиском каждый должен самостоятельно, тем более, что в современных процессорах и операционных системах есть что искать!