crackme, прячущий код на API-функциях

крис касперски ака мыщъх, no-email

> **сейчас мы будем занимается увлекательным делом — писать crackme и пытать его на предмет взлома, а crackme будем молчать как партизан, потому что прячет защитный код на API-функциях, где до него не может дотянуться ни дизассемблер, ни дампер, ни даже отладчик. это древний прием, уходящий своими корнями в эпоху MS-DOS, но ничуть не хуже работающий и в мире Windows**

введение или руководящая идея

Будем исходить из того, что основным орудием хакера является дизассемблер и отладчик (а при анализе упакованных программ к ним еще добавляется и дампер). Существует множество хитроумных трюков, затрудняющих трассировку и отравляющих дизассемблеру жизнь, однако, они только подогревают интерес хакера и зачастую вызывают непредсказуемые конфликты, что не есть хорошо.

А давайте просто спрячем защитный код там, где никто не станет его искать? Программные комплексы наших дней содержат миллионы строк кода и потому никогда не анализируются целиком. Если хакер встречает вызов API-функции LoadLibrary, он искренне *верит*, что это действительно LoadLibrary, а не что-то еще. Теоретически, в API-функцию можно заглянуть отладчиком, однако, практически она представляет "черный ящик". Анализируя аргументы и последовательность вызов API-функций (этим занимаются API-шпионы), хакер получает общее представление о работе защитного механизма и, зачастую, прибегать к отладчику/дизассемблеру уже нет необходимости.

Мыщъх предлагает защищаться так: берем какую-нибудь ненужную API-функцию, которая заведомо не используется, и копируем поверх нее свой собственный код (далее по тексту называемый X-кодом), выполняющий что-то полезное, например, проверяющий серийный номер или ну я не знаю. Тут главное фантазию иметь! Выбирать лучше всего неброскую, ненапряженную функцию с названием не вызывающим у хакера никаких подозрением, например, GetTimeZoneInformation. Естественно, перед копированием необходимо присвоить атрибут записи, что достигается вызовом VirtualProtect с флагом PAGE_EXECUTE_READWRITE, а после копирования вернуть исходные атрибуты защиты обратно.

Модификация API-функций носит сугубо локальный характер. Механизм Copy-on-Write автоматически "расщепляет" все измененные страницы и потому изменения может увидеть только тот процесс, который их записал. Это значит, что за конфликты с другими процессами можно не волноваться. Заботиться о восстановлении оригинального содержимого API-функций так же не нужно.

Поскольку, адреса API-функций не остаются постоянными и варьируются одной системы к другой, X-код не может привязываться к своему расположению в памяти и должен быть полностью перемещаем. Чтобы не заморачиваться можно разместить X-код внутри нашей программы, а в начало API-функции внедрить jump. Конечно, это будет намного более заметно. Стоит хакеру заглянуть отладчиком в API-функцию, как он тут же поймет, что она пропатчена, а вот при копировании X-кода поверх API-функции это уже не так очевидно. Так что свой путь каждый выбирает сам.

Развивая мысль дальше, можно не затирать ненужное API, а взять общеупотребляемую функцию типа CreateFile и слегка "усовершенствовать" ее, например, незаметно расшифровывать содержимое файла или просто мухлевать с аргументами. Допустим, программа открывает файл "file_1". X-код, внедренный в CreateFile, заменяет его на "file_2", передавая управление оригинальной CreateFile – пусть она его открывает!

Фактически, мы приходим к идее создания API-перехватчика (ака шпиона), только шпионить он должен не за чужими процессорам, а за своим собственным, перехватывая нужные API-функции и скрытно выполняя некоторые дополнительные действия. Это серьезно затрудняет дизассемблирование, поскольку ключевые моменты защитного алгоритма идут мимо хакера, который ни хрена не может понять как это работает и почему (например, можно внедрить в CreateFile процедуру проверки ключевого файла, а в самой программе только имитировать его выполнение, заставляя хакера анализировать километры совершенного левого кода).

классический перехват API-функций

Прежде, чем приступать к кодированию, разберем алгоритм классического перехвата, пример готовой реализации можно найти, например, в wmfhotfix.cpp от Ильфака Гильфанова, исходный код которого можно скачать по адресу: castlecops.com/downloads-file-499-details-WMF_hotfix_source_code.html:

- перехватчик запоминает несколько первых команд API-функции в своем буфере (buf);
- дописывает к ним jump на остаток API-функции (after_thunk);
- в начало API-функции ставит jump на X-код (thunk);
- X-код выполняет все, что задумано, и прыгает на buf, отдавая управление API-функции;

Эта схема позволяет внедрять X-код перед вызовом API-функции. Внедрение-после-завершения осуществляется чуть-чуть сложнее (call вместо jump с подменой адреса возврата), но… классическая схема имеет кучу проблем и подводных граблей, которые очень сложно обойти (ведь они под водой!).

Переменная длина машинных команд на x86 затрудняет определение их границ и мы не можем просто взять и скопировать несколько байт, ведь при этом легко разрезать последнюю машинную команду напополам и тогда вместо перехвата мы поимеем крах. Приходится либо тащить за собой примитивный дизассемблер (а это очень много ассемблерных строк), либо ориентироваться на стандартный пролог push ebp/mov ebp,esp, занимающий всего три байта (55 8B EC), в то время как jump на thunk требует как минимум пять! За прологом же может идти все, что угодно — и sub esp,xxx и push, и… В общем-то, вариантов не так уж и много, но закладываться на них ни в коем случае нельзя, иначе перехватчик получится не универсальным и не переносимым.

Проблема номер два — точки останова. Допустим, отладчик типа soft-ice или OllyDbg установил программную точку останова на перехваченную функцию, внедрив в ее начало машинную команду INT 03h (CCh) с предварительным сохранением оригинального содержимого в своем теле. Что должен делать в этом случае наш перехватчик? Даже если он распознает искаженный пролог, копировать начало API-функции в buf ни в коем случае нельзя, ведь тогда будет скопирована и точка останова. Когда она сработает и передаст управление отладчику — отладчик просмотрит свои записи, увидит, что по данному адресу лично он не устанавливал никакой точки останова и не будет ее восстанавливать! Как следствие — возникает необработанное исключение и крах. Как быть, что делать? Можно, конечно, проанализировать первый байт API-функции и если он равен CCh отказаться от перехвата, только с таким предложением лучше сразу идти в зад. Зачем нам нужен перехватчик, который ничего не перехватывает?

К тому же, механизм DEP, поддерживаемый XP SP2 и выше, запрещает выполнение кода в области данных, значит, располагать buf в стеке нельзя. То есть можно, конечно, но перед этим необходимо вызывать VirtualProtect, назначая права доступа на исполнение. Это не проблема, но все-таки напрягает. Исходный код классического перехватчика получается слишком большим и совершенно ненаглядным, так что не будем его здесь приводить, а лучше покурим хорошей травы и подумаем как нам быть.

универсальный перехватчик API-функций

А вот другой способ перехвата. Весьма критикуемый многими программистами, не то, чтобы очень элегантный, но зато предельно простой:

- сохраняем несколько байт от начала API функции, копируя их в buf (>= sizeof(jump));
- в начало API-функции ставим jump на наш thunk;
- в thunk'e: анализируем аргументы функции и вообще делаем все, что хотим;
- в thunk'e: восстанавливаем начало API-функции, копируя buf в ее начало;
- в thunk'e: вызываем восстановленную API-функцию call'ом с подменой адреса возврата;
- в thunk'e: в начало API-функции вновь устанавливаем jump на thunk и выходим;

Все это умещается буквально в 5-10 строк Сишного кода и очень быстро программируется. Это ликвидирует проблему точек останова, поскольку они исполняются на своем "законном" месте, причем отладчик всплывает на оригинальной API-функции, а код перехватчика остается незамеченным. Причем, никакая часть кода не исполняется в области данных, что очень хорошо с точки зрения DEP. Необходимость определения границ машинных

команд отпадает сама собой, поскольку перед выполнением API-функции мы возвращаем ее содержимое на место и скопированные байты автоматически "стыкуются" со своим хвостом. Короче, сплошные преимущества. Так не бывает. А недостатки где? А вот!

При перехвате интенсивно используемых функций наша программа будет слегка тормозить за счет постоянного вызова VirtualProtect(api_func, 0x1000, PAGE_READWRITE, &old)/VirtualProtect(api_func, 0x1000, old, &old_old), правда, можно присвоить началу функции атрибут PAGE_EXECUTE_READWRITE и при удалении thunk'a его не восстанавливать. Теоретически, это ослабит безопасность системы, поскольку наш процесс может пойти в разнос и что-то сюда записать, однако, эта угроза не настолько велика, чтобы принимать ее всерьез.

Многопоточность — вот главная проблема и самый страшный враг. Что произойдет, если в процессе модификации API-функции ее попытается исполнить другой поток? Переключение контекста может произойти в любой момент. Допустим, поток A выполнил инструкцию push ebp и только собирался приступить к mov ebp,esp как был прерван системным планировщиком Windows, переключившим контекст управления на поток B, устанавливающий thunk. Когда поток A возобновит свою работу, то команды mov ebp,esp там уже не окажется. Вместо нее будет торчать "хвост" от jump, попытка выполнения которого ничем хорошим не кончится. Крах будет происходить не только не многоЦПшных, но даже на одноЦПшных системах, пускай и с небольшой вероятностью. Аналогичная проблема имеется и у классических перехватчиков, но, поскольку они устанавливают thunk один-единственный раз, для них она не так актуальна. (***внимание**: имеется вполне осязаемая вероятность вызова API-функции посторонним потоком в момент, когда она будет восстановлена перехватчиком! В этом случае перехватчик упустит вызов, поэтому использовать данный алгоритм в "сторожевых" программах типа брандмауэра ни в коем случае нельзя*).

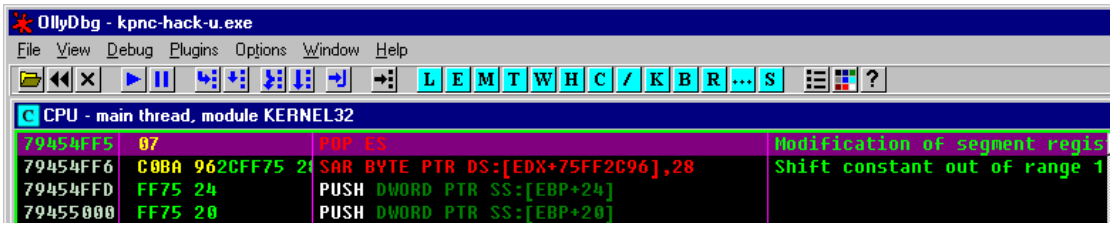

Рисунок 1 последствия неблагоприятного переключения контекста на функции CreateProcessA — машинные инструкции POP ESP/SAR byte ptr DS:[EDX+75FF2C96],28 на самом деле представляют собой хвост команды JMP 10001000, передающей управление на thunk

Семафоры и interlock'и здесь не спасают и единственное, что в наших силах — проектировать программу так, чтобы в каждый момент времени одну и ту же функцию мог вызывать только один поток. Самое простое — создать однопоточное приложение или вызывать API-функции через "переходники" с блокировками, примеры которых можно найти в любой книге, посвященной программированию многоЦПшных систем.

Так что данный способ перехвата все-таки имеет право на существование, поэтому, рассмотрим его подробнее, вплотную занявшись программированием.

создаем jump на thunk

Написать перехватчик можно и на чистом Си, но настоящие хакеры так не поступают, да и не интересно это. Мы будем писать на ассемблере! Для упрощения интеграции перехватчика с кодом защищаемой программы используем ассемблерные вставки. Microsoft Visual C++ поддерживает спецификатор naked ("голый"), запрещающий компилятору самовольничать. "Обнаженные" функции не содержат ни пролога, ни эпилога, компилятор даже не вставляет ret — все это мы должны сделать самостоятельно. Если, конечно, захотим.

Лучшего средства для создания защитных механизмов, пожалуй, и не придумаешь! Подробное описание naked-функций можно найти в SDK (см. раздел "Naked Function Calls"), нам же достаточно знать, что naked-функции объявляются как __declspec(naked) function_name(), придерживаются cdecl-соглашения (аргументы передаются справа налево и удаляются из стека материнской функцией) и не формируют стековый фрейм, то есть смещения локальных переменных и аргументов нам придется рассчитывать самостоятельно.

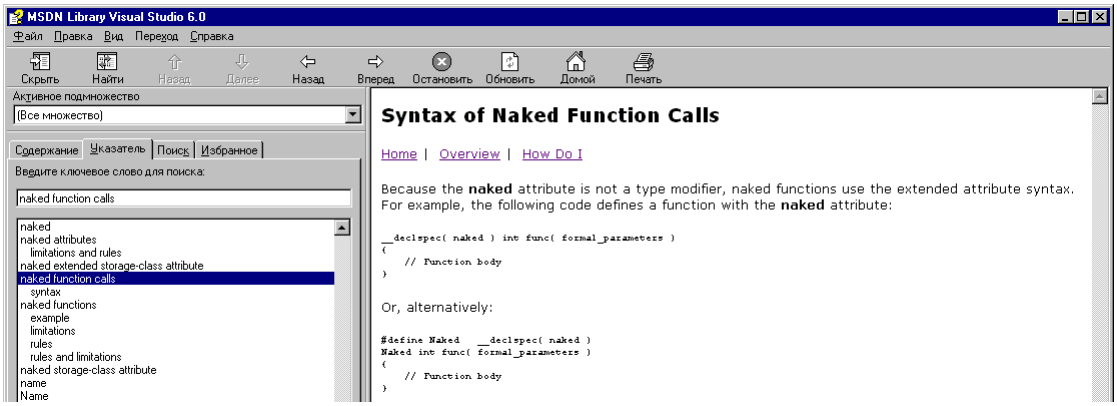

Рисунок 2 голые функции на MSDN

Попробуем в качестве разминки создать процедуру, внедряющуюся в начало некоторой API-функции и передающую управление на thunk. Проще всего это сделать с помощью команды jmp thunk, однако, при этом значение thunk придется рассчитывать вручную, поскольку в x86 процессорах команда jmp ожидает не само смещение, а разницу между смещением целевого адреса и концом команды jmp. Вычислить-то его несложно, просто отнял/прибавил — и все, но… это же нужно высаживаться на самомодифицрующийся код, что не входит в наши планы. Лучше пойти другим путем, обратившись к конструкции mov reg32, offset thunk/jmp reg32. Это на один байт длиннее и к тому же портит регистр reg32, однако, это не страшно. Все API-функции придерживаются соглашения stdcall, то есть принимают аргументы через стек, а возвращают значение через регистровую пару [edx]:eax, то есть значение eax при входе в функцию не играет никакой роли и может быть безболезненно искажено.

А вот с "лишним" байтом все значительно сложнее. Некоторые (впрочем, очень немногочисленные функции) состоят из одного jmp xxx, следом за которым расположен другой jmp. Естественно, это не сами функции, это просто линкер сформировал таблицу переходов, но нам-то от этого не легче! К тому же, иногда встречаются функции короче пяти байт (например, GetCurrentProcess) и внедрить в них jmp (даже без mov) уже невозможно!

```
.text:77E956D7 ; HANDLE GetCurrentProcess(void)
.text:77E956D7 public GetCurrentProcess
.text:77E956D7 GetCurrentProcess      proc    near    ; CODE XREF: UnhandledExceptionF
.text:77E956D7 83 C8 FF               or      eax, 0FFFFFFFFh
.text:77E956DA C3                     retn
.text:77E956DA GetCurrentProcess      endp
.text:77E956DA
.text:77E956DB ; Exported entry 315. GetModuleHandleA
.text:77E956DB
.text:77E956DB ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
.text:77E956DB public GetModuleHandleA
.text:77E956DB GetModuleHandleA       proc near       ; CODE XREF: .text:77E815D6↑p
.text:77E956DB
.text:77E956DB lpModuleName           = dword ptr  8
.text:77E956DB
.text:77E956DB 55                     push ebp
.text:77E956DC 8B EC                  mov ebp, esp
```

Листинг 1 API-функция GetCurrentProcess занимает всего 4 байта и внедрить в нее jump, не испортив начала следующей функции уже невозможно (на самом деле — можно, но сложно! в GetCurrrentProcess мы пишем push esp/push esp/push esp/push esp/push esp, а в GetModeleHandleA внедряем jump на sub_thunk, анализирующий что находится на вершине стека — если там четыре esp, то был вызван GetCurrentProcress, в противном случае это GetModuleHandleA)

Ладно, не будем высаживаться на измену. Все это заморочки. В общем случае, перехват работает вполне корректно и простейший перехватчик выглядит так:

```
#define JUMP_SZ        0x6            // размер jump
__declspec( naked ) jump()            // "голая" функция без пролога и эпилога
{
        __asm
        {
                mov eax, offset thunk  ; заносим в eax смещение нашего thunk'a
```

```
                jmp eax                 ; передаем на него управление
        }
}
```

Листинг 2 код, внедряющийся в начало API-функции и передающий управление на thunk

Единственная проблема — как определить его длину? Должны же мы знать сколько байт копировать? Оператор sizeof возвращает размер указателя на функцию, но отнюдь не размер самой функции. Какие еще есть пути? Можно, например, определить размер jump'a вручную (в данном случае он равен 6 байтам) или расположить за его концом фиктивную процедуру fictonic(). Разница смещений fictonic() и jump() в общем случае и будет размером jump. Почему в общем случае? Да потому, что компилятор не подписывался всегда размещать функции в порядке их объявления (хотя чаще всего все происходит именно так). Но это еще что! Если откомпилировать программу с ключом /Zi (отладочная информация), компилятор будет возвращать совсем не адрес функции jump(), а указатель на "переходник", расположенный совсем в другом месте!

```
.text:00401019 loc_401019:                             ; DATA XREF: .text:loc_40106D↓o
.text:00401019                                         ; _main+39↓o
.text:00401019                 jmp     _jump
.text:00401019 ; ---------------------------------------------------------------------------
.text:0040101E                 dd 8 dup(0CCCCCCCCh)
.text:0040103E                 align 10h
.text:00401040
.text:00401040 _thunk          proc near               ; CODE XREF: .text:loc_401005↑j
.text:00401040                 ;
.text:0040106D                 ; [мыщъх поскипал]
.text:0040106D                 ;
.text:0040106D _thunk          endp;
.text:00401081; ---------------------------------------------------------------------------
.text:00401081
.text:00401081 _jump           proc near               ; CODE XREF: .text:loc_401019↑j
.text:00401081                 mov     eax, offset loc_401005
.text:00401086                 jmp     eax
.text:00401086 _jump           endp
.text:00401086
```

Листинг 3 при компиляции с ключом /Zi компилятор MS VC вместо указателя на саму функцию возвращает указатель на переходник, что есть бэд (правда, можно написать простейший анализатор, распознающий jmp и вычисляющий эффективный адрес функции, но это же лишний код!)

Переходник и указатель разделяют целых 68h байт, а в некоторых случаях и побольше. Кошмар! Даже если копировать с "запасом" мы все равно уйдем лесом и рухнем в прорубь. Тем не менее, без ключа /Zi все работает вполне нормально, а отлаживать программу можно и в машинных кодах.

>>> врезка как это отлаживают?

Без отладочной информации отлаживать программу очень хреново, а отладочной информации у нас нет и не будет, потому что с ключом /Zi компилятор ведет себя не совсем адекватно. Чтобы не трассировать весь код целиком, в нужное место исходного кода можно внедрить __asm{int 03}, что вызовет исключение, передающее Just-In-Time отладчику бразды правления. Обычно этим отладчиком является Microsoft Visual C++ Debugger, но можно использовать и OllyDbg (Options -> Just-In-Time Debugging) или другие отладчики.

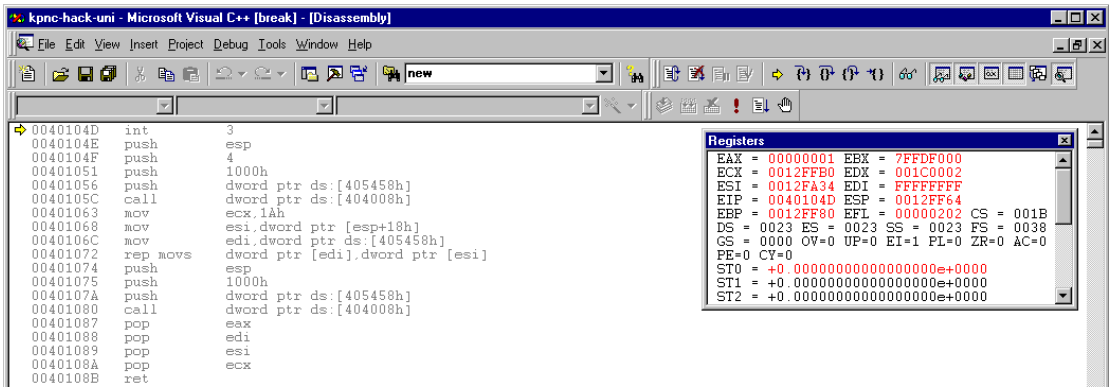

Рисунок 3 Just-In-Time отладка при помощи int 03h под Microsoft Visual C++ Debugger

перехват одиночной API-функции

Переход на thunk реализуется просто. Сам thunk запрограммировать намного сложнее. На первый взгляд никаких мин здесь нет: снимаем thunk/вызываем функцию/устанавливаем thunk. Вызываем мы, конечно, call'ом (а чем же еще!), забрасывающим на стек адрес возврата в thunk и чуть-чуть приподнимающим его вершину, но этого "чуть-чуть" оказывается вполне достаточно, чтобы API-функция не могла найти свои аргументы (см. рис. 4).

```
                                    [00]:адрес возврата в thunk
[00]:адрес возврата в программу     [04]:адрес возврата в программу
[04]:аргумент 1                     [08]:аргумент 1
[08]:аргумент 2                     [0C]:аргумент 2
[0C]:аргумент 3                     [10]:аргумент 3
```

Рисунок 4 состояние стека на момент вызова API-функции до (слева) и после (справа) перехвата с вызовом по call'у (в квадратных скобках приведено смещение аргументов относительно esp)

Отказаться от call'а нельзя — ведь наш thunk должен как-то отловить момент завершения функции, чтобы вернуть восстановленный jump на место, иначе данный перехват будет первым и последним. А что если… скопировать аргументы функции, продублировав их на вершине стека? Тогда на момент вызова API-функции картина будет выглядеть так (см. рис. 5):

```
                                    [00]:адрес возврата в thunk
                                    [04]:аргумент 1
                                    [08]:аргумент 2
                                    [10]:аргумент 3
[04]:адрес возврата в программу     [04]:адрес возврата в программу
[08]:аргумент 1                     [08]:аргумент 1
[0C]:аргумент 2                     [0C]:аргумент 2
[10]:аргумент 3                     [10]:аргумент 3
```

Рисунок 5 состояние стека на момент вызова API-функции до (слева) и после (справа) перехвата с дубляжом аргументов

За исключением потери нескольких десятков байтов стекового пространства все выглядит очень замечательно и нормально работает, вот только код перехватчика получается довольно громоздким и не универсальным. Почему? Вспомним, что API-функции придерживаются stdcall-соглашения при котором аргументы удаляет из стека сама вызываемая функция. Аргументы легко скопировать "с запасом", но откуда мы знаем сколько их удалять при выходе из thunk'а? А удалять необходимо, ведь в противном случае стек окажется несбалансированным и все рухнет.

Вообще-то, можно просто проанализировать значение регистра ESP до и после выполнения API-функции. Дельта и будет тем количеством байт, на которые мы должны увеличить ESP. Но есть и более короткие пути: если перехватываемая функция известна, достаточно просто создать функцию-двойник, имеющий тот же самый прототип. На языке Си это будет выглядеть так (в данном случае перехватывается MessageBoxA, в заголовке которой насильно прописывается строка "hacked", подтверждающая, что перехватчик исправно работает):

```
// наш thunk
//=========================================================================
// получаем управление до вызова функции MessageBoxA
// может делать что угодно с параметрами и т.д.
__stdcall thunk(HWND h, LPCTSTR lpTxt, LPCTSTR lpCaption, UINT uType)
{
        _do(_UNINSTALL_THUNK_);              // восстанавливаем оригинальную функцию

        MessageBox(h,lpTxt,"hacked",uType);  // вызываем оригинальную функцию

        _do(_INSTALL_THUNK_);                // вновь устаналиваем thunk

}
```

Листинг 4 демонстрация перехвата API-функции с известным прототипом

перехват произвольной API-функции

Язык ассемблера выгодно отличается от Си тем, что позволяет реализовать универсальный перехватчик, поддерживающие все функции и не засирающий стек дублированными аргументами. Все очень просто. Вернемся к рис. 4. Скажите, на хрена держать в стеке сразу два адреса возврата, когда можно обойтись и одним? Ведь функции отрабатывают последовательно, передавая эстафету как степени реактивной ракеты. Короче.

В момент передачи управления на thunk на вершине стека находится адрес возврата в прикладную программу. Запоминаем его в глобальной переменной saved, и пишем сюда адрес возврата в thunk, на который будет передано управление после выхода из API-функции. Скопировав на стек сохраненный адрес мы сможем вернуться в прикладную программу по ret и при этом не придется химичить с аргументами!

Красота! А вот ее готовая программная реализация: (### а вот как эта красота реализуется):

```
// наш thunk
//=========================================================================
// получаем управление до вызова перехватываемой API-функции
// здесь можем делать что угодно с параметрами и т.д.
 __declspec( naked ) thunk()
{
__asm{
        push offset _UNINSTALL_THUNK_        ; снимаем thunk с функции
        call _do_asm
        add esp,4

        pop [saved]                          ; подменяем ret_addr на post_exit
        push offset post_exit

        ; "боевая нагрузка" до вызова функции
        ; ===================================================================
        ; подменяем строку заголовка и меняем тип диалогового окна
pre_run:
        mov eax, offset my_string
        ; mov [esp+12], offset my_string     ; ms vc не поддерживает такую команду :(
        mov [esp+12], eax
        mov [esp+10h], 1

        mov eax,[p]                          ; вызываем перехваченную функцию as is
        jmp eax

        ; "боевая нагрузка" после вызова функции
        ;====================================================================
        ; ничего не делаем (добавьте сюда собственный код, если хотите)
post_exit:

        push offset _INSTALL_THUNK_          ; снова устанавливаем thunk на функцию
        call _do_asm
        add esp,4

        push [saved]                         ; возвращаемся туда, откуда нас вызывали
        retn
        }
}
```

Листинг 5 ассемблерный код универсального перехватчика, работающего через подмену адреса возврата и способный нести на своем борту "боевую начинку", выполняемую до или после вызова API-функции

>>> врезка тупой, тупой компилятор MS VC

Встроенный ассемблер компилятора Visual C++ 6.0 не вполне поддерживает синтаксис Intel и отказывается транслировать инструкцию "mov [esp+12], offset my_string", выдавая убийственно сообщение "fatal error C1001: INTERNAL COMPILER ERROR":

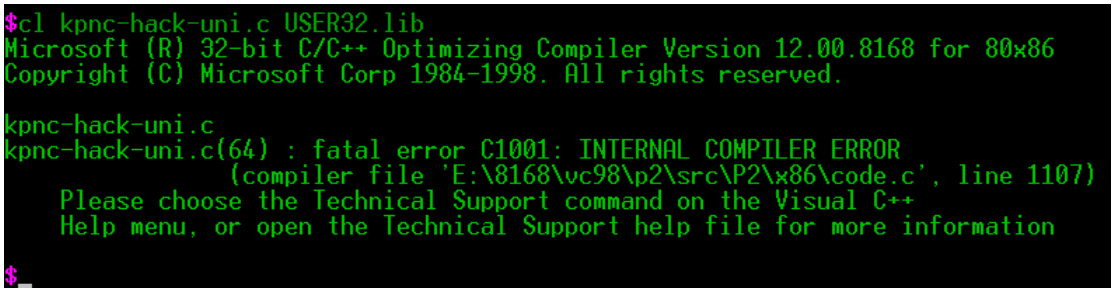

Рисунок 6 реакция MS VC 6 на mov [esp+12], offset my_string

Гребанный Microsoft! Приходится перепихивать offset через регистр, что длиннее и вообще выглядит некрасиво (mov eax, offset my_string/mov [esp+12], eax), но писать непосредственно в машинных кодах через _emit мыщъх'у как-то не улыбается.

устанавливаем и удаляем thunk

Остается заточить несложную процедуру, устанавливающую jump на thunk и восстанавливающую содержимое API-функции перед ее вызовом. Эта функция называется memcpy. Ну… почти memcpy. Чтобы установка jump'a завершилась успехом необходимо вызвать VirtualProtect с флагом PAGE_READWRITE, что слегка усложняет реализацию, однако, не столь радикально, чтобы впадать в депрессию. Это можно запрограммировать как на ассемблере, так и на Си. На Ассамблее — круче, на Си — быстрее.

Ниже приводится ассемблерный листинг с несколькими интересными хаками:

```
// устанавливает/удаляет thunk
//============================================================================
  __declspec( naked )  _do_asm(char *src)
{
__asm
{
        ; сохраняем регистры, которые будут изменены
        push ecx
        push esi
        push edi

        ; резервируем место под локальные переменные
        push esp                                ; резервируем место под old-old (hack!!!)
        push eax                                ; резервируем место под old

        ; вызываем VirtualProtect(p,0x1000,PAGE_READWRITE, &old);
        ; присваивая себе атрибут записи
        push esp                                ; &old
        push PAGE_READWRITE                     ; нельзя PAGE_EXECUTE_READWRITE!
        push 0x1000                             ; size
        push [p]                                ; указатель на регион
        call ds:VirtualProtect

        ; копируем память из src в p двойными словами
        mov ecx, JUMP_SZ/4                      ; size в дв. словах
        mov esi, [esp+18h]                      ; src !!!следить за смещением!!!
        mov edi, [p]                            ; dst
        rep movsd                               ; копируем!

        ; вызываем VirtualProtect(p,0x1000,old,&old-old)
        ; восстанавливая прежние атрибуты зашиты
        push esp                                ; old (hack!!!)
```

```
        push 1000h                              ; size
        push [p]                                ; указатель на регион
        call ds:VirtualProtect

        pop eax                                 ; выталкиваем old
        ;pop eax                                ; old-old уже вытолкнут v_prot

        ; восстанавливаем измененные регистры
        pop edi
        pop esi
        pop ecx

        ; выходим
        retn
        }
}
```

Листинг 6 ассемблерный код процедуры, устанавливающей и снимающий thunk с API

Хак номер один. Резервирование места под локальные переменные командой push. Ну это и не хак вовсе. Так даже компиляторы поступают! Инструкция push eax забрасывает на верхушку стека содержимое регистра eax, а команда push esp заталкивает указатель на eax, передавая его как аргумент функции VirtualProtect, которая записывает сюда текущие атрибуты, выталкивая указатель из стека по завершении. А это значит, что на вершине стека вновь оказывается локальная переменная, с прежними атриумами. Вот только передать ее функции VirtualProtect через push esp уже не получится, поскольку она ожидается во втором слева аргументе. Компилятор (даже самый оптимизирующий) наверняка влепил бы сюда команды типа push eax/push esp/push [esp+8], что слишком длинно и вообще маст дай.

Вот если бы в стеке уже содержался указатель на фиктивную ячейку памяти, которую было можно передать VirtualProtect, но, к сожалению, его там нет… Но! Ведь его можно очень легко сделать! Для этого достаточно лишь передать аргумент до первого вызова VirtualProtect, что и делаем команда push esp с комментарием "hack!!!". Да! Аргумент для второго вызова VirtualProtect заносится в стек в первую очередь, экономя целых три машинных команды. Почему три? Да потому, что одну из двух локальных переменных выталкивает сама функция VirtualProtect и это второй хак!

Вот оно — отличие между ассемблером и языками высокого уровня. На ассемблере мы можем писать намного более эффективно и компактно, пускай даже ценою потерянного времени, но зато как интересно оптимизировать программы, выкидывая из них все ненужное!

>>> врезка защита от Microsoft

Ходят неясные слухи, что последующие версии Windows запретят прикладному коду присваивать все три атрибута PAGE_EXECUTE_READWRITE одновременно, поскольку реально это нужно только зловредному коду типа червей. Это сможет делать только система или на худой конец администратор. Поэтому, перед копированием необходимо присвоить атрибуты PAGE_READWRITE, а поле — PAGE_EXECUTE.

демонстрация перехватчика или попытки взлома

Законченная модель перехватчика содержится в файле kpnc-hack-uni.c, который по сути представляет самый настоящий crackme, выводящий диалоговое окно с заголовком "not-hacked yet" на экран. Во всяком случае, дизассемблер утверждает именно так (см. листинг 7), но в действительности там красуется строка "hacked", свидетельствующая о том, что дизассемблерам верить нельзя.

```
.text:004010CB _demo          proc near               ; CODE XREF: j__demo↑j
.text:004010CB                push    ebp
.text:004010CC                mov     ebp, esp
.text:004010CE                push    0
.text:004010D0                push    offset aNotHackedYet   ; "not hacked yet"
.text:004010D5                push    offset aDemo           ; "==[ demo ]=="
.text:004010DA                push    0
.text:004010DC                call    ds:__imp__MessageBoxA@16
.text:004010E2                pop     ebp
.text:004010E3                retn
.text:004010E3 _demo          endp
```

Листинг 7 дизассемблерный листинг функции demo, создающей диалоговое окно с заголовком "not hacked yet", которое перехватывается нашим перехватчиком и "исправляется" на "hacked"

Первый раз MessageBox вызывается без перехватчика, затем с перехватчиком и в конце еще один раз снова без перехватчика (чтобы подтвердить, что перехватчик снимается правильно и без побочных последствий).

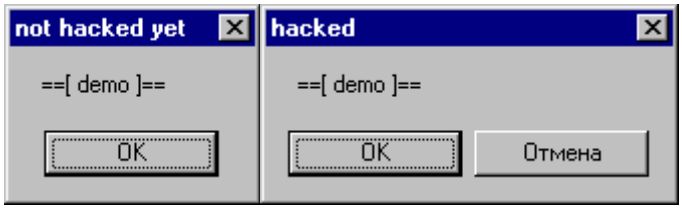

Рисунок 7 диалоговое окно, вызываемое функцией demo до (слева) и после (справа) перехвата; функция — одна, диалоговые окна — разные!

Конечно, данный crackme очень легко проанализировать и взломать, поскольку он невелик и защитный код сразу же бросается в глаза. Но в полновесной программе со всеми ее мегабайтами просто так захачить ничего не получится потому что, ну короче. Отладчик (потенциально) способен обнаружить перехват — для этого достаточно просто немного потрассировать API-функцию, но! Легко сказать "немного потрассировать". Это же очень даже до фига трассировать придется, особенно в свете того, что точка останова, установленная до перехвата, срабатывает не в перехватчике, а в оригинальной API-функции.

А вот еще идея — пусть в процессе распаковки программы, распаковщик копирует ключевой код поверх API-функций и удаляет его из самой программы. Тогда сдампленная программа окажется неработоспособной, ведь содержимое API-функций по умолчанию не дампиться! Конечно, хакер может сделать это вручную, если, конечно, разберется в ситуации, что будет совсем непросто!

>>> врезка маскировка VirtualProtect

Вызов VirtualProtect – это Ахиллесова пята API-перехватчика, демаскирующая его присутствие. Если хакер установит точку останова, он не только обнаружит факт модификации API-функций, но еще и определит их адреса, переданные VirtualProtect в первом слева аргументе, поэтому, необходимо прибегнуть к маскировке.

Самое простое (но не самое надежное) — вместо VirualProtect вызывать VirtualProtectEx, передавая вместо описателя процесса значение -1 (FFFFFFFFh). Вдруг хакер не догадается установить на нее точку останова? Так ведь нет, догадается же…

На NT-подобных системах мы можем использовать NtProtectVirtualMemory, вызывая ее не с первого байта. Подробное описание этого трюка можно найти в "Записках мыщъх'а" или статье "точки останова на win32 API и противодействие им", опубликованной в "Системном Администраторе". На 9x никакой NtProtectVirtualMemory нет, и там нужно вызывать VirualProtectEx, опять-таки, не с первого байта, тогда все точки останова не получат управления и наша шпионская деятельность останется незамеченной.

Некоторые предлагают заменить VirtualProtect на VirtualAlloc с флагом MEM_COMMIT | MEM_RESET, чтобы изменить атрибуты страницы памяти, однако, по отношению к системным библиотекам этот трюк не работает и атрибуты не изменяются, так что у нас остается только VirtualProtectEx.

>>> глобализация перехвата

Универсальный перехватчик легко доработать, научив его внедряться в чужие процессы (тогда он превратиться в самый настоящий API-монитор, позволяющий не только следить за системными вызовами но и, например, блокировать нежелательные функции или подменять их своими, эмулируя открытие ключевого файла, регистрацию по сети и т. д.).

Достаточно многие API-шпионы запускают подопытное приложение как отладочный процесс (CreateProcrss с флагом DEBUG_PROCESS), что дает им полный доступ к адресному пространству, но такой API-шпион легко обнаружить вызовом IsDebuggerPresent и чтобы

оставаться незамеченными мы должны перехватывать эту функцию, всегда возвращая нулевое значение (отладчик не установлен).

Другие предпочитают модифицировать память другого процесса функциями VirualProtectEx/WriteProcrssMemory, предварительно выделив блок памяти вызовом VirtualAllocEx. В этом случае факт шпиона обнаружить намного сложнее и большинство защитных механизмов пропускают его мимо ушей, но мы можем шпионить только за известными процессами.

А если мы хотим мониторить все процессы сразу? Например, хотим ограничить доступ в сеть, заблокировать удаление файлов или просто интересуемся какая су… нехорошая программа постоянно дрыгает диском? Это действительно возможно! Все, что нам нужно — оформить перехватчик в виде dll и прописать ее в следующей ветке системного реестра: HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs, тогда она будет проецироваться на адресное пространство всех процессов, запускаемых в системе. Инициализация перехватчика осуществляется в функции DllMain, вызываемой при подключении динамической библиотеке к очередному процессу, а так же в некоторых других случаях (создание/удаление новых потоков и т. д.). Пример готовой реализации приведен в уже упомянутом hotfix'e от Ильфака.

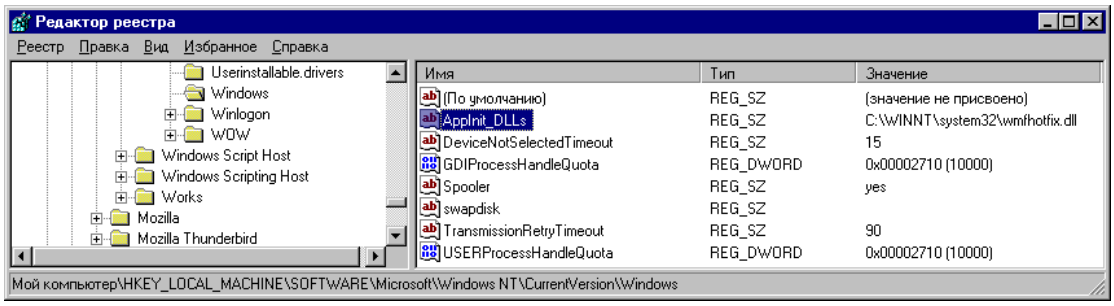

Рисунок 8 ветка реестра, ответственная за проецирование динамических библиотек на все процессы

заключение

Большинство антиотладочных трюков утрачивают свою силу, когда становятся известными. Но только не этот! Обнаружить факт перехвата, даже зная о его возможности, очень сложно, так что мы имеем довольно могучий защитный прием с широким спектром действия и убойным радиусом поражения, а если его еще и усовершенствовать… получится термоядерное оружие, которое можно применять как для взлома чужих приложений, так и для защиты своих собственных.