универсальный метод обнаружения малвари

крис касперски ака мыщхъ, no-email

в Windows постоянно обнаруживаются новые дыры, через которые лезет малварь, создающая новые процессы или внедряющаяся в уже существующие. автор этой статьи предлагает универсальный метод обнаружения малвари, основанный на определении подлинного стартового адреса потока, чего другие приложения (включая могучий отладчик soft-ice) делать не в состоянии.

введение

Антивирусы, брандмауэры и прочие системы защиты (вроде упомянутого в соседней статье Buffer Zone) хорошо справляются с вирусами и червями, но в борьбе с малварью они бессильны. Чтобы не утонуть в терминологической путанице, здесь и далее по тексту, под малварью мы будем понимать программное обеспечение, скрытно проникающее на удаленный компьютер и устанавливающее там back-door или ворующее секретную информацию.

В первую очередь нас будет интересовать малварь, не способная к размножению, и зачастую написанная индивидуально для каждой конкретной атаки, а потому существующая в единственном экземпляре. При условии, что она не распознается эвристическим анализатором (а обмануть эвристический анализатор очень легко), антивирус ни за что не поймает ее, поскольку такой сигнатуры еще нет в его базе, да и откуда бы она там взялась?!

Персональный брандмауэр тоже не слишком надежная защита. Множество дыр дают злоумышленнику привилегии SYSTEM (что повыше администратора будет) с которыми можно творить все, что угодно, в том числе и принимать/отправлять пакеты в обход брандмауэра.

Тем не менее, обнаружить присутствие малвари на компьютере все-таки возможно. Автор этой статьи проанализировал множество зловредных программ и обнаружил их слабые места, выдающие факт внедрения с головой.

как малварь внедряется на компьютер

Наиболее примитивные экземпляры малвари создают новый процесс, который внимательный пользователь легко обнаружит в "диспетчере задач". Конечно, для этого необходимо знать, какие процессы присутствуют в "стерильной" системе и где располагаются их файлы. В частности, explorer.exe, расположенный не в WINNT, а в WINNT\System32, это уже никакой не explorer, а самая настоящая малварь!

Впрочем, "диспетчер задач" крайне уязвимая штука и малварь без труда скрывает свое присутствие от его взора. Тоже самое относится к FAR'y, Process Explorer'y, tlist'y и прочим системным утилитам основанным на недокументированной API-функции **`NtQuerySystemInformation()`**, экспортируемой динамической библиотекой NTDLL.DLL, и потому очень легко перехватываемую даже с прикладного уровня, без обращения к ядру и даже без администраторских привилегий.

Отладчик soft-ice – единственный известный мне инструмент не использующий `NtQuerySystemInformation()` и разбирающий структуры ядра "вручую". Спрятаться от него на порядок сложнее и в "живой природе" такая малварь пока незамечена (а лабораторные экземпляры крайне нежизнеспособны и способны обманывать только известные им версии отладчика), так что на soft-ice вполне можно положиться. Для просмотра списка процессов достаточно дать команду "`PROC`" и проанализировать результат.

Кстати, малврь, скрывающаяся от "диспетчера задача", немедленно выдает свое присутствие путем сличения "показаний" soft-ice с "диспетчером задач". Один из таких случаев продемонстрирован на рис. 1. Смотрите, soft-ice отображает процесс sysrtl, но в "диспетчере задач" он… отсутствует! Следовательно, это либо малварь, либо какой-нибудь хитроумный защитный механизм, построенный по root-kit технологии. В общем — нехорошая программа, от которой можно ждать все, что угодно и желательно избавиться как можно быстрее!

Рисунок 1 зловредный процесс sysrtl замаскировал свое присутствие от "диспетчера задач", но не смог справится с soft-ice

Для достижения наибольшей скрытности малварь должна не создавать новый процесс, а внедряться в один из уже существующих, что она с успехом и делает. Классический алгоритм внедрения реализуется так:

- получив идентификатор процесса-жертвы (что можно сделать, например, через семейство процедур TOOLHELP32), малварь "скармливает" его API-функции `OpenProcess()`, возвращающей дескриптор процесса (или ошибку, если у малвари недостаточно прав);
- возращенный дескриптор процесса передается API-функции `VirtualAllocEx()`, выделяющий в адресном пространстве процесса-жертвы блок памяти требуемых размеров с атрибутами `PAGE_READWRITE` или `PAGE_READ` (но тогда все оперативные данные придется хранить в стеке);
- поверх выделенного блока копируется зловредный код (который должен быть полностью перемещаемым, т. е. сохранять свою работоспособность независимо от базового адреса загрузки), что осуществляется API-функцией `WriteProcessMemory();`
- с помощью все тех же процедур TOOLHELP32, малварь находит главный поток процесса, получает его идентификатор, который тут же преобразует в дескриптор. в Window 2000 (и ее благородных потомках) это осуществляется API-функцией `OpenThread()`, а в более ранних версиях приходилось прибегать к вызову недокументированной native-API функции `NtOpenThread()`, экспортируемой библиотекой NTDLL.DLL. под 9x задача решается "серединным" вызовом API-функции `OpenProcress()`, путем передачи управления по смещению 24h от ее начала и расшифровкой идентификатора операцией XOR со специальным "магическим" словом;
- добытый дескриптор потока передается API-функции `SuspendThread()`, останавливающий его выполнение;
- содержимое контекста остановленного потока читается API-функцией `GetThreadContext()` с флагом `CONTEXT_CONTROL`, в результате чего в структуре `CONTEXT` оказывается значение регистра `EIP`, указывающего на текущую машинную инструкцию;
- запомнив полученный `EIP`, малварь тут же корректирует его с таким расчетом, чтобы он указывал на точку входа в ранее скопированный зловредный код, и вызывает API-функцию `SetThreadContext()`, чтобы изменения EIP вступили в силу, после чего "размораживает" остановленный поток посредством `ResumeThread();`
- во избежании утечки ресурсов, дескрипторы процесса и потока закрываются — больше они малвари не понадобятся (хотя далеко не всякая малварь заботится о таким мелочах);
- получив управление, зловредный код создает новый поток вызовом `CreateThread()` и восстанавливает исходное значение регистра `EIP`;
- **примечание**: до появления процессоров, поддерживающих биты NX/XD, предотвращающих выполнение кода в стеке и куче, малварь обычно выделяла в целевом процессе регион памяти с атрибутами `PAGE_READWRITE`, а теперь — `PAGE_EXECUTE_READWRITE`, что, впрочем, слишком заметно, поэтому грамотные малваре-писатели выделяют блок с атрибутами `PAGE_EXECUTE`, что _никак_ не препятствует функции `WriteProcessMemory()` записывать туда зловредный код.

Описанный алгоритм работает на всем зоопарке операционных систем, но довольно громоздок и сложен в реализации, поэтому, малварь, ориентированная на поражение только одной NT, предпочитает создавать удаленный поток API-функций `CreateRemoteThread()`, при этом последовательность выполняемых ею действий выглядит так:

- получив идентификатор процесса жертвы, малварь "скармливает" его API-функции `OpenProcess()`, возвращающей дескриптор процесса (или ошибку, если у малвари недостаточно прав);
- возращенный дескриптор процесса передается API-функции `VirtualAllocEx()`, выделяющий внутри процесса-жертвы блок памяти требуемых размеров с атрибутами `PAGE_EXECUTE`;

- поверх выделенного блока копируется зловредный код (который, так же как и в предыдущем случае, должен быть полностью перемещаемым), что осуществляется API-функцией `WriteProcessMemory();`
- малварь вызывает API-функцию `CreateRemoteThread()`, передавая ей дескриптор процесса и указатель на стартовый адрес потока, находящийся внутри блока памяти, выделенного `VirtualAllocEx();`
- дескриптор процесса и дескриптор удаленного потока, возвращенный `CreateRemoteThread()`, закрываются, а зловредный код тем временем делает все, что ему вздумается;

Единственный недостаток, присущий последнему способу внедрения — это требование перемещаемости кода, означающее, что его придется писать на ассемблере, используя только относительную адресацию, что весьма затруднительно.

Усовершенствованный алгоритм внедрения позволяет загружать внутрь чужого процесса свою собственную динамическую библиотеку, для чего достаточно передать функции `CreateRemoreThread()` в качестве стартового адреса удаленного потока адрес API-функции `LoadLibraryA()` или `LoadLibraryW()`, а вместо указателя на аргументы — указатель на имя загружаемой библиотеки. API-функция `CreateRemoreThread()` вызовет `LoadLibraryA/LoadLibraryW` вместе с именем библиотеки, в результате чего библиотека загрузится в память, а управление получит процедура `DllMain()`. Зловредная динамическая библиотека может быть написана на любом языке – хоть на Си/Си++, хоть на DELPHI, хоть… на Visual Basic'e, что значительно расширяет круг потенциальных малваре-писателей, поскольку ассемблер знают относительно немногие.

Вся беда в том, что имя библиотеки должно находиться в контексте удаленного процесса, а как оно там окажется?! Существует два пути: самое простое, но не самое умное, это выделить блок памяти вызовом `VirtualAllocEx()` и скопировать туда имя через `WriteProcessMemory()`, но для этого процесс должен быть открыт с флагом "виртуальные операции" (`PROCESS_VM_OPERATION`), прав на которые у малвари может и не быть.

Выручает тот факт, что библиотеки NTDLL.DLL и KERNEL32.DLL во всех процессах проецируются по одинаковым адресам. Получив базовый адрес загрузки NTDLL.DLL или KERNEL32.DLL с помощью `LoadLibrary()`, малварь сканирует свое собственное адресное пространство на предмет наличия ASCIIZ-строки, совершенно уверенная в том, что в удаленном процессе эта строка окажется расположенной по тому же самому адресу. Остается только переименовать зловредную динамическую библиотеку в эту самую строку. Кстати, приятным побочным эффектом такого алгоритма становится автоматическая генерация псевдослучайных имен (если, конечно, малварь не будет использовать первую попавшуюся ASCIIZ-строку).

Обобщив сказанное, мы получаем следующий план:

- выбрав идентификатор процесса-жертвы, малварь "скармливает" его API-функции `OpenProcess()`, возвращающей дескриптор процесса (или ошибку, если у малвари недостаточно прав);
- определив базовый адрес загрузки NTDLL.DLL или KERNEL32.DLL, малварь ищет подходящую ASCIIZ-строку, переименовывая свою, заранее созданную, динамическую библиотеку;
- определив адрес API-функции LoadLibraryA/LoadLibraryW, малварь передает его API-функции `CreateRemoteThread()` вместе с указателем на имя библиотеки, которую необходимо загрузить внутрь целевого процесса;
- дескриптор процесса и дескриптор удаленного потока, возвращенный `CreateRemoteThread()` закрываются, а зловредный код, расположенный в `DllMain()`, делает все, что ему вздумается;

Вот три основных алгоритма внедрения в атакуемый процесс, которыми пользуется порядка 90% всей малвари.

по следам малвари или как обнаружить внедрение

Если количество процессов в системе вполне предсказуемо, то потоки многократно создаются/уничтожаются в ходе выполнения легальных программ и вопрос "сколько потоков должна иметь "стерильная" программа" лишен смысла. Достаточно открыть "диспетчер задач"

и, некоторое время понаблюдав за колонкой "потоки", прийти в полное отчаяние. Но… если присмотреться повнимательнее, можно обнаружить, что потоки, созданные малварью, значительно отличаются от всех остальных.

При внедрении малвари по двум первым сценариям зловредный код располагается в блоках памяти, выделенных `VirtualAllocEx()` и имеющих тип `MEM_PRIVATE`, в то время как нормальные исполняемые файлы и динамические библиотеки загружаются в блоки памяти типа `MEM_IMAGE`. При внедрении по третьему сценарию, зловредный код как раз и попадает в такой блок, но стартовый адрес его потока совпадает с адресом функции `LoadLibraryA()` или `LoadLibrayW()`, а указатель на аргументы содержит имя зловредной библиотеки.

Таким образом, алгоритм обнаружения вторжения сводится к определению стартовых адресов всех потоков и, если он лежит внутри `MEM_PRIVATE` или совпадает с адресом `LoadLibraryA()/LoadLibraryW()` – этот поток создан малварью или чем-то сильно на нее похожим. Вот тут-то и начинается самое интересное! Ни soft-ice, ни process explorer Марка Руссиновича определять стартовые адреса _не_ умают (хоть и пытаются). Они _очень_ часто ошибаются, особенно при работе с потоками, созданными малварью.

Давайте напишем "макетную" программку, создающую поток тем же самым методом, что и малварь и попробуем обнаружить факт "вторжения" при помощи подручных утилит. Предельно упрощенный исходный текст "макетника" выглядит так:

```
// код потока, который ничего не делает, а только мотает цикл
thread(){while(1);}

main()
{
        void *p;        // переменная многоцелевого использования

        // создаем "честный" поток
        CreateThread(0,0,(void*)&thread,0x999,0,&p);

        // создаем "нечестный" поток так, как это делает malware
        // выделяем блок памяти из кучи, копируем туда код потока
        // и вызываем CreateThread
        p = VirtualAlloc(0, 0x1000, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        memcpy(p,thread,0x1000);CreateThread(0,0,p,0x666,0,&p);

        // ждем нажатия на любую клавишу
        getchar();
}
```

Листинг 1 "макетная" программа va_thread.c, создающая поток тем же самым методом, что и малварь

Компилируем с настройками по умолчанию (в случае MS VC++ командная строка выглядит так: "`cl.exe va_thread.c`") и запускаем. Загрузка процессора (даже на двухпроцессорной машине!) сразу подпрыгивает до 100%, но так и должно быть, поскольку мы создаем два потока, мотающих бесконечный цикл, один из которых "честный", а другой "зловредный" (имитирующий малварь). Плюс главный поток приложения, ожидающий нажатия на клавишу по которой происходит завершение программы. Итого — три потока.

Загружаем soft-ice и нажимаем <CTRL-D>, дожидаясь его вызова, после чего даем команду "`THREAD -x va_thread`" для отображения детальной информации о потоках и смотрим на полученный результат (см. рис. 2). Да! Тут есть на что посмотреть! Стартовый адрес первого потока (Start EIP) определен как `KERNEL32!SetUnhandledExceptionFilter+001A` (77E878C1h), а двух остальных — `KERNEL32!CreateFileA+00C3` (77E92C50h), что вообще ни в какие ворота не лезет, если не сказать, что это откровенная деза.

Рисунок 2 отладчик soft-ice, пытающийся определить стартовые адреса потоков, но возвращающий вместо этого нечто необъяснимое

Отбросив бесполезный soft-ice в сторону, обратимся к process explorer'у. Щелкнув правой клавишей мыши по процессу "va_thread" (или нажав SHIFT-F10, если мыши под рукой нет), лезем в "properties" и открываем вкладку "threads". Что мы видим? process explorer корректно определил адреса двух потоков (см. рис. 3): `va_thread+0x1405` (основной системный поток — если заглянуть дизассемблером по этому адресу, мы обнаружим точку

входа в файл va_thread.exe) и `va_thread+0x1000` ("честный" поток, созданный вызовом `CreateThread(0,0,(void*)&thread,0x999,0,&p)` — это следует из того, что по адресу `va_thread+0x1000` распложена процедура thread). Но вот вместо стартового адреса третьего, "нечестного" потока, process explorer выдал какую-то ерунду, засунув его внутрь KERNEL32.DLL, а точнее — `KERNEL32.DLL + B700h`, где его _заведомо_ не может быть.

Если бы process explorer ошибался только на "нечестных" потоках, он вполне бы сгодился для определения малвари, но, увы, он ошибается слишком часто, в том числе и на легальных потоках, созданных операционной системой или ее компонентами.

Рисунок 3 process explorer успешно определил стартовые адреса двух "честных" потоков, но споткнулся о "нечестный" поток

Исследования, проведенные автором этой статьи, показали, что истинный стартовый адрес потока лежит на дне пользовательского стека во втором или третьем двойном слове (считая от единицы), а следом на ним идет указатель на аргументы, в чем легко удостоверится с помощью отладчика OllyDbg.

Запустив отладчик, в меню "file" выбираем "attach" и подключаемся к процессу "va_thread.exe", после чего открываем окно "threads" (в меню "view") и видим не три, а целых четыре потока! Все правильно — четвертый поток создан отладчиком для своих нужд. Это единственный поток, чье поле entry (точка входа) не равно нулю. Стартовые адреса трех остальных потоков OllyDbg определить не смог, предоставив нам возможность сделать это самостоятельно.

Дважды щелкнув мышью по любому из потоков мы попадаем внутрь его "закромов". Отладчик обновляет содержимое регистров, окно CPU, дамп памяти и окно стека, которое нас интересует больше всего. Прокручиваем мышью ползунок до самого конца и обнаруживаем на дне нечто очень интересное (см. рис. 4), а именно — два двойных слова: 666h и 520000h. Первое из них до боли напоминает аргумент, переданный "нечестному" потоку (см. листинг 1), а по второму расположена функция, мотающая бесконечный цикл, весьма напоминающая нашу функцию `thread()`. Обратившись к карте памяти (view → memory), мы убедимся, что этот адрес принадлежит региону `MEM_PRIVATE`, выделенному `VirtualAlloc()`. Аналогичным образом определяются стартовые адреса и двух других потоков.

Рисунок 4 определение стартового адреса потока с помощью отладчика OllyDbg

Свершилось! Мы научились определять подлинные стартовые адреса "честных" и "нечестных" потоков вместе с переданным им указателем на аргументы. Однако, использовать для этих целей OllyDbg не слишком удобно. Потоков в системе много и пока их все в ручную переберешь… рабочий день давно закончится и солнце зайдет за горизонт. Вообще-то, можно написать простой скрипт (OllyDbg поддерживает скрипты), но при этом отладчик придется всюду таскать за собой, что напрягает. Лучше (и правильнее!) написать свой собственный сканер, тем более, что он легко укладывается в сотню строк и на его разработку уйдет совсем немного времени.

Полный исходный текст содержится в файле ProcList.c, прилагаемом к статье, а здесь, для экономии места, приводятся лишь ключевые фрагменты. Но, прежде чем углубляться в теоретическую дискуссию, проверим сканер в работе. Наберем в командой строке "`proclist.exe > out`" и загрузим образовавшийся файл out в любой текстовой редактор (например, встроенный в FAR). Нажмем <F7> (search) и введем имя интересующего нас процесса ("va_thead.exe"). Запомним его идентификатор (в данном случае равный 578h) и, снова нажав <F7> введем его для поиска принадлежащих ему потоков. Вот они, все три, перечисленные в листинге 2, лежат рядышком.

```
LoadLibraryA at : 79450221h
LoadLibraryW at : 794502D2h
-----------------------------------------------------
szExeFile           : va_thread.exe
cntUsage            : 0h
th32ProcessID       : 578h
…
--thr-----------------------------------------------
th32ThreadID       : 5E0h
th32OwnerProcessID : 578h
…
```

```
handle              : 3D8h
ESP                 : 0012FD30h
start address       : 00401595h
point to args       : 00000000h
type                : MEM_IMAGE
[0012FFF0h: 00000000 00000000 00401595 00000000]
--thr-------------------------------------------------
th32ThreadID        : 608h
th32OwnerProcessID  : 578h
…
handle              : 3D8h
ESP                 : 0051FFB4h
start address       : 00401000h
point to args       : 00000999h
type                : MEM_IMAGE
[0051FFF0h: 00000000 00401000 00000999 00000000]
--thr-------------------------------------------------
th32ThreadID        : 5C8h
th32OwnerProcessID  : 578h
…
handle              : 3D8h
ESP                 : 0062FFB4h
start address       : 00520000h
point to args       : 00000666h
type                : MEM_PRIVATE
[0062FFF0h: 00000000 00520000 00000666 00000000]
```

Листинг 2 фрагмент отчета сканера proclist.exe, определяющего стартовые адреса всех потоков вместе с типами блоков памяти

Первые два потока находятся внутри блоков `MEM_IMAGE` и ни у одного из них стартовые адреса не совпадают с адресами функций `LoadLibraryA()/LoadLibraryW()`, следовательно, это "честные" потоки, созданные легальным путем. А вот третий поток лежит внутри региона `MEM_PRIVATE`, выделенного API-функцией `VirtualAlloc()`. Значит, это "нечестный" поток и мы не бьем тревогу только потому, что сами же его и создали.

Теперь, как было обещано, обсудим технические детали. Прежде всего нам потребуется получить список потоков, имеющихся в системе. Это можно сделать как документированными средствами через TOOLHELP32, так и недокументированной native-API функций `NtQuerySystemInformation()`, на которой TOOLHELP32, собственно говоря, и основан. Конечно, если она перехвачены малварью, мы никогда не увидим зловредных потоков, но техника обнаружения/снятия перехвата — это тема отдельной статьи, пока же придется ограничиться тем, что есть (на всякий случай, сравните показания TOOLHELP32 с командой "THREAD" отладчика soft-ice, вдруг обнаружатся какие-то различия).

Короче, список потоков в простейшем случае получается так:

```
#include <stdio.h>
#include <windows.h>
#include <tlhelp32.h>

print_thr(THREADENTRY32 thr)
{
        printf("cntUsage           : %Xh\n",thr.cntUsage);
        printf("th32ThreadID       : %Xh\n",thr.th32ThreadID);
        printf("th32OwnerProcessID : %Xh\n",thr.th32OwnerProcessID);
        printf("tpBasePri          : %Xh\n",thr.tpBasePri);
        printf("tpDeltaPri         : %Xh\n",thr.tpDeltaPri);
        printf("dwFlags            : %Xh\n",thr.dwFlags);
}

main()
{
        HANDLE h; THREADENTRY32 thr; int a;

        // создаем "слепок" потоков
        h = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);

        // перебираем все потоки один за другим
        thr.dwSize = sizeof(THREADENTRY32); a = Thread32First(h, &thr);
        if (a && print_thr(thr)) while(Thread32Next(h, &thr)) print_thr(thr);
}
```

Листинг 3 фрагмент кода, ответственный за перечисление всех имеющихся потоков

Теперь нам необходимо прочесть контекст каждого из потоков, получив значение регистра ESP, указывающего куда-то внутрь стека (конкретно куда - не суть важно). Кстати говоря, считывать контекст можно и без остановки потока. На Windows 2000 (и ее потомках) это делается так (более ранние версии требуют использования native-API функции NtOpenThread() и, поскольку, доля таких систем сравнительного невелика, здесь они не рассматриваются):

```
HANDLE ht;
CONTEXT context;
context.ContextFlags = CONTEXT_CONTROL;

// преобразуем идентификатор потока в дескриптор
ht = OpenThread(THREAD_GET_CONTEXT, 0, thr.th32ThreadID);

// считываем регистровый контекст
GetThreadContext(ht,&context);

// закрываем дескриптор
CloseHandle(ht);
```

Листинг 4 фрагмент кода, считывающего значение ESP потока с идентификатором thr.th32ThreadID

Заметим, что API-функция OpenThread() не входит ни в заголовочные файлы, ни в библиотеку KERNEL32.LIB, поставляемую вместе с компилятором Microsoft Visual C++, поэтому, необходимо либо скачать свежий Platform SDK (а это очень-очень много мегабайт), либо загружать ее динамически через GetProcAddress(), либо преобразовать KERNEL32.DLL в KERNEL32.LIB (линкер unilink от Юрия Харона это сделает автоматически).

Зная идентификатор процесса, владеющий данным потоком (thr.th32OwnerProcessID), мы можем открыть его API-функцией OpenProcerss(), получив доступ к его адресному пространству (если, конечно, у нас на это есть права). Открывать мы будем с флагами PROCESS_QUERY_INFORMATION (просмотр виртуальной памяти) и PROCESS_VM_READ (чтение содержимого виртуальной памяти).

Следующий шаг — определение дна пользовательского стека. Передав API-функции VirtualQueryEx() значение ESP, полученное из регистрового контекста, мы узнаем базовый адрес выделенного блока (mbi.BaseAddress) и его размер в байтах (mbi.RegionSize). Путем алгебраического сложения базового адреса с его длинной, мы получим указатель на первый байт памяти, лежащий _за_ концом стека. Отступив на несколько двойных слов назад (например, на четыре), нам остается только прочитать его содержимое API-функцией ReadProcessMemory().

```
#define GET_FZ 4              // на сколько двойных слов отступать
DWORD buf[GET_FZ];
DWORD x; HANDLE hp;
MEMORY_BASIC_INFORMATION mbi;

// открываем процесс, владеющий данным потоком
hp = OpenProcess(PROCESS_VM_READ|PROCESS_QUERY_INFORMATION,0,thr.th32OwnerProcessID);

// определяем параметры блока памяти, на который указывает регистр ESP
VirtualQueryEx(hp, (void*)context.Esp, &mbi, sizeof(mbi));

// вычисляем положение дна стека
x = (DWORD) mbi.BaseAddress + mbi.RegionSize;

// читаем GET_FZ слов со дна стека в буфер buf
ReadProcessMemory(hp,(char*)x-GET_FZ*sizeof(DWORD),buf,GET_FZ*sizeof(DWORD),&a);

// закрываем дескриптор процесса
CloseHandle(hp);
```

Листинг 5 фрагмент кода, определяющий положение дна стека и считывающий GET_FZ байт с его конца (в которых хранится стартовый адрес потока)

Ввиду того, что положение стартового адреса относительно дна стека непостоянно — применяется следующий эвристический алгоритм: если третье двойное слово со дня стека равно нулю, то стартовый адрес потока находится во втором двойном слове, а указатель на аргументы — в первом, в противном случае, стартовый адрес находится в третьем двойном

слове, а указатель на аргументы — во втором. Конечно, предложенная схема не очень надежна: в некоторых случаях стартовый адрес находится в первом двойном слове, а бывает (хоть и редко), что на дне стека его вообще нет. В общем, этот вопрос еще требует дальнейших исследований и тщательной проработки, а пока воспользуйтесь тем, что дают:

```
DWORD st_adr;

// определяем стартовый адрес потока
st_adr = ((buf[GET_FZ-3])?buf[GET_FZ-3]:buf[GET_FZ-2]);

// выводим стартовый адрес на экран или DEADBEEF,
// если стартовый адрес определить не удалось
printf("start address      : %08Xh\n",st_adr?st_adr:0xDEADBEEF);

// определяем указатель на аргументы потока и выводим его на экран
printf("point to args      : %08Xh\n",((buf[GET_FZ-3])?buf[GET_FZ-2]:buf[GET_FZ-1]));
```

Листинг 6 декодирование содержимого буфера buf и определение стартового адреса потока эвристическим методом

Остается последнее — "скормить" полученный стартовый адрес потока API-функции `VirtualQueryEx()` и определить тип региона, к которому он принадлежит (MEM_IMAGE, MEM_PRIVATE или MEM_MAPPED):

```
// определение типа региона памяти,
// к которому принадлежит стартовый адрес
VirtualQueryEx(hp, (void*)st_adr, &mbi, sizeof(mbi));

// декодирование полученного результата
// и вывод его на экран
printf("type               : %s\n",
(mbi.Type==MEM_IMAGE)?"MEM_IMAGE":
(mbi.Type==MEM_MAPPED)?"MEM_MAPPED":
(mbi.Type==MEM_PRIVATE)?"MEM_PRIVATE":"UNKNOWN");
```

Листинг 7 фрагмент кода, определяющий тип блока памяти, к которому принадлежит стартовый адрес потока

Объединив все фрагменты мозаики воедино, мы получим вполне работоспособный сканер, показавший при тестировании на большой коллекции малвари, вполне удовлетворительный результат — ни одного ложного срабатывания и до 90% обнаруженной "заразы" (естественно, речь идет только о малвари, внедряющийся в уже существующие процессы, а не создающие новый).

заключение

Описанный метод выявления вторжения обладает рядом существенных недостатков, которые автор этой статьи даже и не пытается скрывать. Первое, и самое неприятное — стартовый адрес потока попадает на дно пользовательского стека лишь по "недоразумению". Никому он там не нужен и всякий поток может смело его обнулить или подделать. Но с этим еще можно хоть как-то бороться, например, просканировать все MEM_PRIVATE блоки и попытаться найти в них машинный код, поискать в стеке адреса возврата, смотрящие в MEM_PRIVATE, и т. д., гораздо хуже, что существует возможность внедрения в атакуемый процесс _без_ создания нового потока!

Самое простое, что только приходит в голову — прочитать текущий `EIP` одного из потоков атакуемого процесса, сохранить лежащие под ним машинные команды, после чего записать крохотный код, вызывающий `LoadLibrary()` для загрузки зловредной библиотеки в текущий контекст и устанавливающий таймер API-функций `SetTimer()` для передачи зловредному коду управления через регулярные промежутки времени. Сделав это, малварь восстанавливает оригинальные машинные команды и процесс продолжает выполняться как ни в чем не бывало.

При желании можно обойтись и без таймера. Достаточно воспользоваться асинхронными сокетами. В отличии от синхронных, они не блокируют выполнение текущего потока, а немедленно отдают управление, вызывая call-back процедуру при наступлении определенного события (например, при подключении удаленного пользователя по back-door порту, открытого малварью).

Так же малварь может внедриться в процедуру диспетчеризации сообщений или подменить адрес оконной процедуры для главного окна GUI-приложения. Во всех этих случаях зловредный код будет выполняться в контексте уже существующего потока и малварь сможет обойтись без создания нового. Такой способ внедрения предложенный сканер не обнаруживает. Правда, это не сильно ему мешает, поскольку в живой природе подобной малвари до сих пор замечено не было. Универсальных способов борьбы против нее нет. Единственное, что можно предложить, – периодически выводить список динамических библиотек и если вдруг среди них появилась новая — это сигнал! Но если малварь откажется от загрузки DLL, размещая свой код в свободном месте (например, в конце кодовой секции), мы опять окажемся в пролете.

Впрочем, не стоит пытаться решить проблемы задолго до их появления. Возможно, завтра случится глобальное оледенение (землетрясение, наводнение), мы все умрем и бороться с малварью станет некому и незачем.

>>> врезка ссылки на программы, упомянутые в статье

- **OllyDbg**:
  - замечательный и абсолютно бесплатный отладчик, собравший вокруг себя целое сообщество поклонников: http://www.ollydbg.de;
- **soft-ice**:
  - 3 апреля 2006 Compuware прекратила продажу DriverStudio, похоронив soft-ice и отрезав все пути его легального приобретения, тем не менее, прежние версии до сих пор можно найти в Сети, в том числе и в магазинах, торгующих лицензионными дисками, завалявшимися на складе, или нелицензионными. в любом случае — техническая поддержка прекращена и пиратский продукт ничем не отличается от легального, разве что стоит на порядка на два дешевле;
- **process explorer**:
  - в июне 2006 года Марк Руссинович продался Microsoft и хотя его утилиты обещают остаться бесплатными, скорее всего они будет бесплатны только для легальных пользователей Windows, так что спешите скачивать: http://www.sysinternals.com/Utilities/ProcessExplorer.html;