шесть грехов malware-писателей

крис касперски ака мыщъх, no-email

подавляющее большинство червей/вирусов/шпионов рождаются беспомощными и абсолютно нежизнеспособными. вызывать крупные эпидемии удается далеко не всем. почему? попробуем разобраться! это не руководство по написанию малвари. это руководство по тому, как _не_ надо ее писать. в статье рассматриваются шесть наиболее "излюбленных" огрехов, палящих малварь и приводящих к ее гибели.

введение

Малварью мы будем называть все вредоносное программное обеспечение, занимающееся размножением, шпионажем, рассылкой рекламы и другими вещами, протекающими без ведома и согласия владельца машины. Достигнув технологического пика своего развития в середине девяностых (когда хакеры додумались до stealth-вирусов и полиморфизма), сейчас малварь предалась разврату и пришла в упадок. В основном она пишется начинающими программистами (пренебрежительно называемых "пионерами"), торчащих на языках высокого уровня типа DELPHI или Visual Basic'a. Как следствие — качество малвари упало ниже плинтуса и основная масса штаммов дохнет еще на излете.

Рисунок 1 дохлая малварь

Проблема отнюдь не в самом DELPHI или Visual Basic'e — это вполне достойные инструменты — проблема в том, что пионеры не умеют ими пользоваться, дружно наступая на один и те же грабли, краткий перечень которых приведен в этой статье.

грех 1 — не протестированный код

Редкая программа пишется без ошибок и начинает работать с полпинка, тем более, если речь идет о таких сложных механизмах как вирусы, черви, шпионы, по сути, являющиеся высоко-автономными роботами, вроде тех аппаратов, что летают на Венеру, Юпитер или Марс. Однажды выпущенная в сеть, зараза становится полностью предоставленной сама себе и допущенные ошибки исправить уже не удастся.

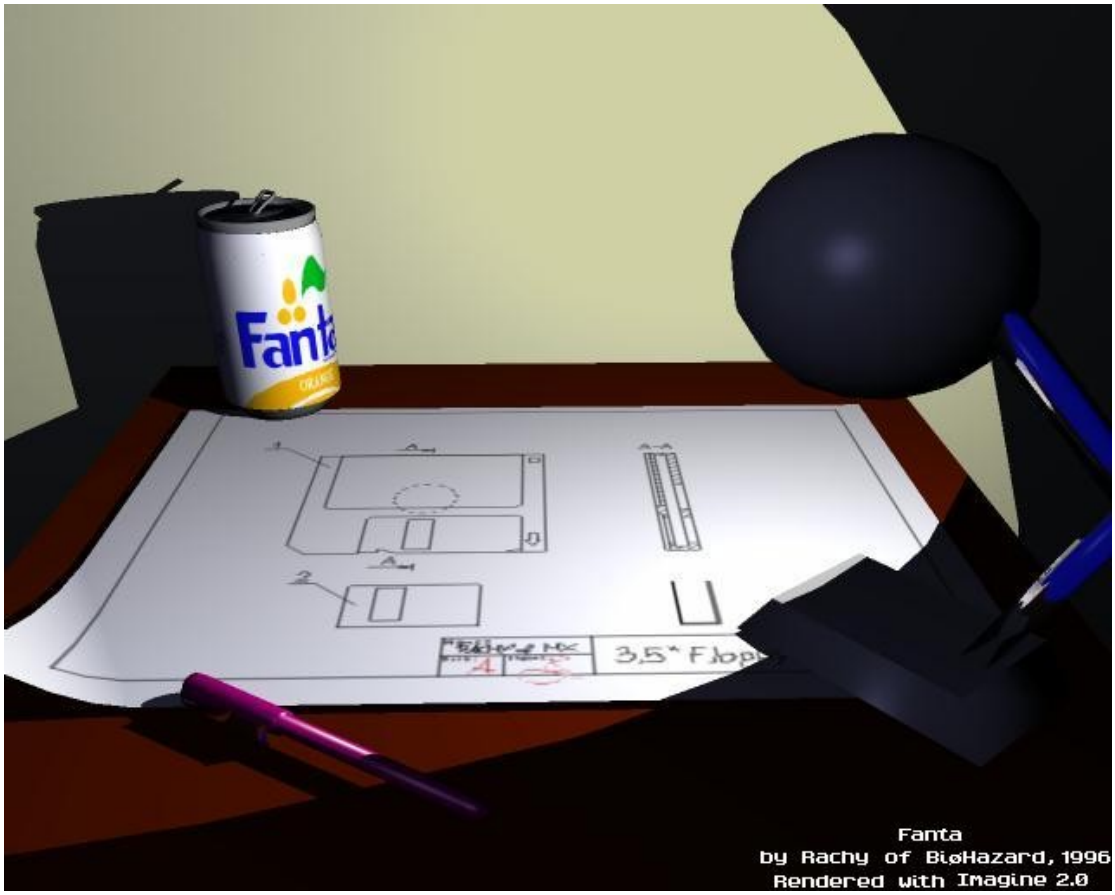

Рисунок 2 проектирование малвари – серьезный подход

А, значит, тестировать, тестировать и еще раз тестировать, пока хвост не согнется в штопор. Так ведь нет! Если малварь запускается и не падает — это уже хорошо! Ну древних (ныне - ископаемых) программистов еще можно как-то понять. У них был только IBM PC в количестве одна штука и "косые" флопы в качестве резервных носителей. Но даже в таких условиях создавались легендарные вирусы типа one-half. Сейчас же… любой может установить VM Ware, установив несколько версий операционных систем (98, W2K, XP, Server 2003) и связав их виртуальной сетью. Лучшего полигона для отладки малвари, пожалуй, и не придумать.

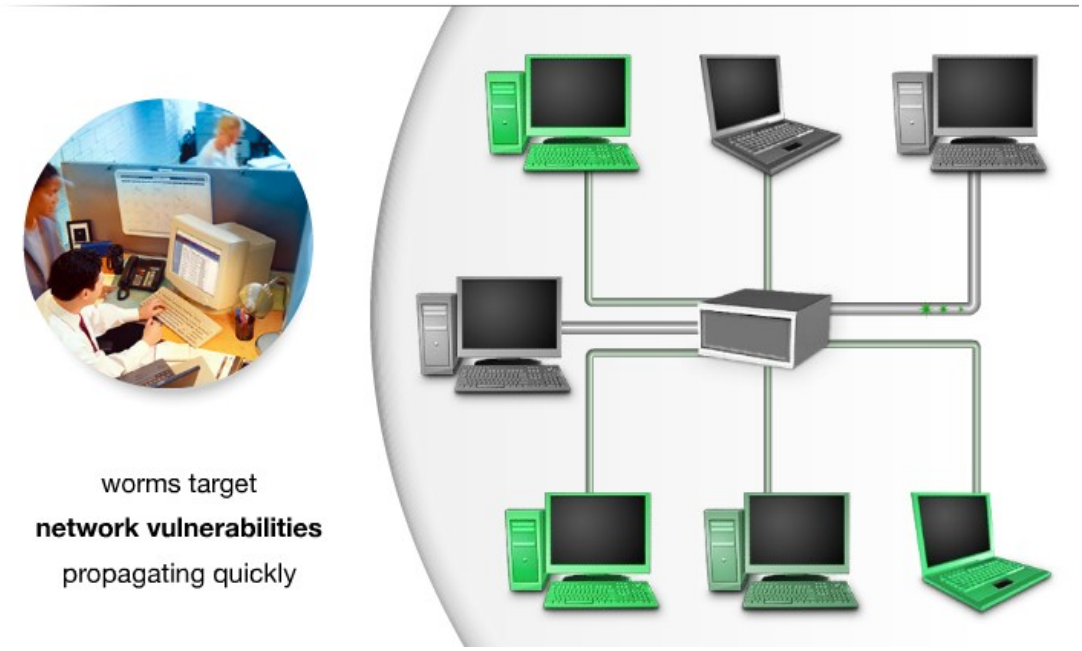

Рисунок 3 моделирование виртуальной сеть при помощи VM Ware

Среднестатистический пользователь обычно замечает малварь лишь тогда, когда его любимый компьютер начинает вести себя не так как всегда: некоторые приложения не запускаются, те же, что запускаются — жутко тормозят, на экран часто выпрыгивают сообщения о критических ошибках вплоть до полного выпадения в голубой экран смерти. Вот тут-то жертва и начинает лихорадочно устанавливать различные антивирусы, брандмауэры, и прочие сторожевые программы, призванные "найти-и-уничтожить", а если ничего не помогает — форматирует винт и переустанавливает систему начисто.

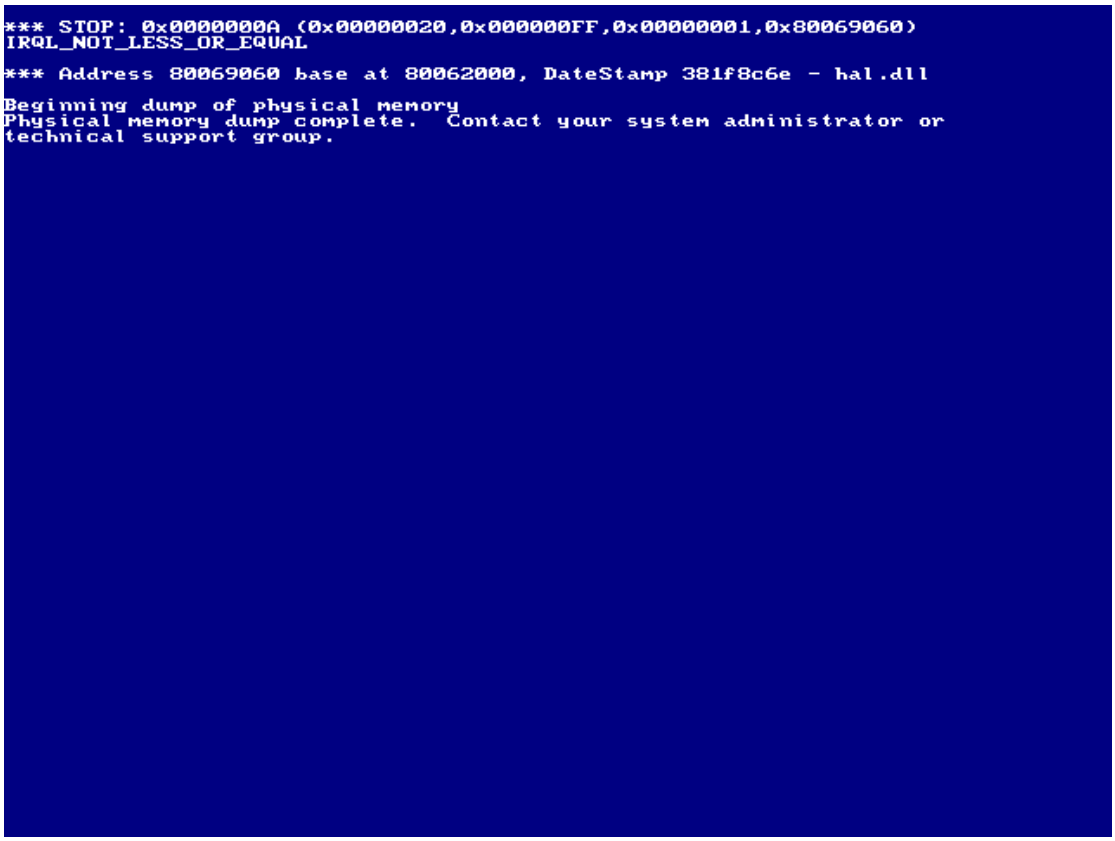

Рисунок 4 Голубой Экран Смерти, вызванный некорректно спроектированной малварью

Чем корректнее ведет себя малварь, тем больше у нее шансов остаться незамеченной, а для этого она должна быть тщательно протестирована, причем на разных процессорах! То, что мгновенно выполняется на Pentium-4, зачастую вызывает 100% загрузку Pentium-II/III, с существованием которых так же приходится считаться.

Рисунок 5 "да пусть хоть миллион вирусов ползает по компьютеру, лишь бы любимые игрушки работали нормально" — лозунг большинства пользователей

грех 2 — излишняя сложность

Чем сложнее механизм, тем больше времени от требует для своего создания и отладки, а конец у всех один. Как только малварь замечают — ее тут же заносят в антивирусную базу и злорадно прибивают. Не могут не прибить! Над этим целая индустрия работает, вербующая отнюдь не глупых людей. Известны случаи (и их достаточно много), когда вирус, разрабатываемый годами (!), палился антивирусной процедурой, созданной меньше, чем за день. Вот такой расклад и расстановка сил.

Технология отлова полиморфиков уже давно отработана. Продвинутые антивирусы (AVP, Dr.WEB) пропускают проверяемый код через эмулятор и гонят его на графы, приводя к тому или иному метаязыку, отражающему _суть_ программы, но _не_ способ ее достижения, поэтому даже самые крутые полиморфики гаснут как бычки в писсуаре, ведь изменить заложенный в них _алгоритм_ они не в силах.

Рисунок 6 антивирусные "вакцины" бьют известную им малварь наповал

Интеллектуально не отягощенные игроки антивирусного рынка просто размножают полиморфик в огромном количестве экземпляров (от 10 тыс. и больше), удаляют все повторы, а оставшиеся — заносят в базу (вот потому для ловли многих вирусов Norton'у под час требуется _сотни_ записей!). Уже никто не анализирует малварь и не потрошит ее дизассемблером, ну разве что самые популярные экземпляры, а для подавляющего большинства остальных антивирусные энциклопедии дают крайне невнятные описания. Загляните для сравнения в энциклопедии десяти-пятнадцатилетней давности. Какие там были описания! По несколько страниц, с фрагментами дизассемблерных листингов — красота!

Но стоит ли уподобляться Microsoft, стремиться к крутости ради крутизны и усложнять малварь без нужны? Зачем разрабатывать навороченные механизмы, когда и простые неплохо работают! Правило самолета (airplane rule) гласит, что: "сложность увеличивает вероятность поломки, двухмоторный самолет по сравнению с одномоторным имеет, по крайней мере вдвое больше проблем с двигателями". Простой пример: сравнение API операционных систем Windows и UNIX. Сложную систему может придумать каждый дурак, но только гений сумеет уложить весь необходимый функционал в минимум строк кода, реализовать и отладит которые уже не составит большого труда.

грех 3 — хвост и усы — вот мои документы

По непонятной, можно даже сказать, мистической, причине подавляющее большинство малваре-писателей _не_ заполняют секцию ресурсов, описывающую свойства файла, что отображается "проводником" в одноименной вкладке. Нормальные коммерческие программы так себя не ведут (для них — это большая редкость), поэтому малварь тут же палит себя. Навряд ли продвинутый пользователь согласится запускать файл "без документов". Если же малварь внедряется обходным путем, например, через дыру в Windows или подпущенную к компьютеру женщину (а женщины имеют тенденцию запускать все без разбора), то утилиты типа Anti-Spy.Info тут же внесут такие файлы в список подозреваемых.

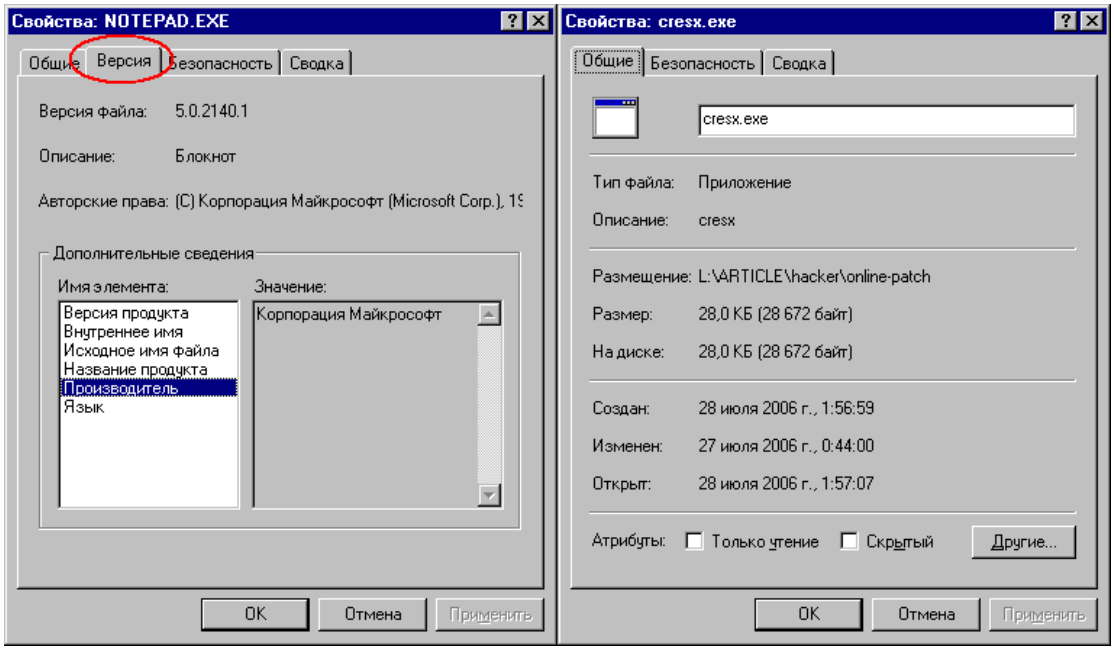

Рисунок 7 легальные программы обычно заполняют раздел VersionInfo в ресурсах и в свойствах файла появляется вкладка "версия" (слева), малварь же обычного этого не делает и вкладки "версия" у нее нет (справа)

Заполнять "бланк" свойств лучше не абы как, а по образу и подобию Microsoft. Ее же и ставить в качестве компании-разработчика. Использовать вымышленные компании крайне нежелательно, поскольку беглый поиск googl'ом тут же разоблачает обман. Прикрываться брендами типа ATI тоже расковано. Вдруг человек предпочитает Matrox — вот он будет недоумевать откуда у него взялась эта гадость, эта ATI на его компьютере. А файлы от Microsoft есть у всех и никто не может сказать сколько их и зачем они нужны.

Другой тонкий момент — иконка. Голый исполняемый файл, изображающий из себя "стандартное приложение Windows", привлекает к себе намного больше внимания чем… морковка. Или редиска! Да что угодно, только лучше не из стандартного набора значков, входящих в состав Microsoft Visual Studio — опытным пользователем они хорошо известны. Надежнее взять что-то совершенно неожиданное — тут все от воображения и фантазии зависит!

грех 4 — дата создания файла

Штатным образом, Windows поддерживает три даты, связанные с каждым файлом — дата создания, дата модификации (доставшаяся ей в наследство от MS-DOS) и дата последнего доступа. При создании файла на диске ему автоматически присваивается текущая дата создания, что позволяет легко изобличить непрошеную заразу — просто загородим в каталог WINNT\System32 своим любимым FAR'ом, ждем <CTRL-F8> и файлы, созданные последними, оказываются наверху. Так же можно воспользоваться и стандартным поиском, интегрированным в "проводник".

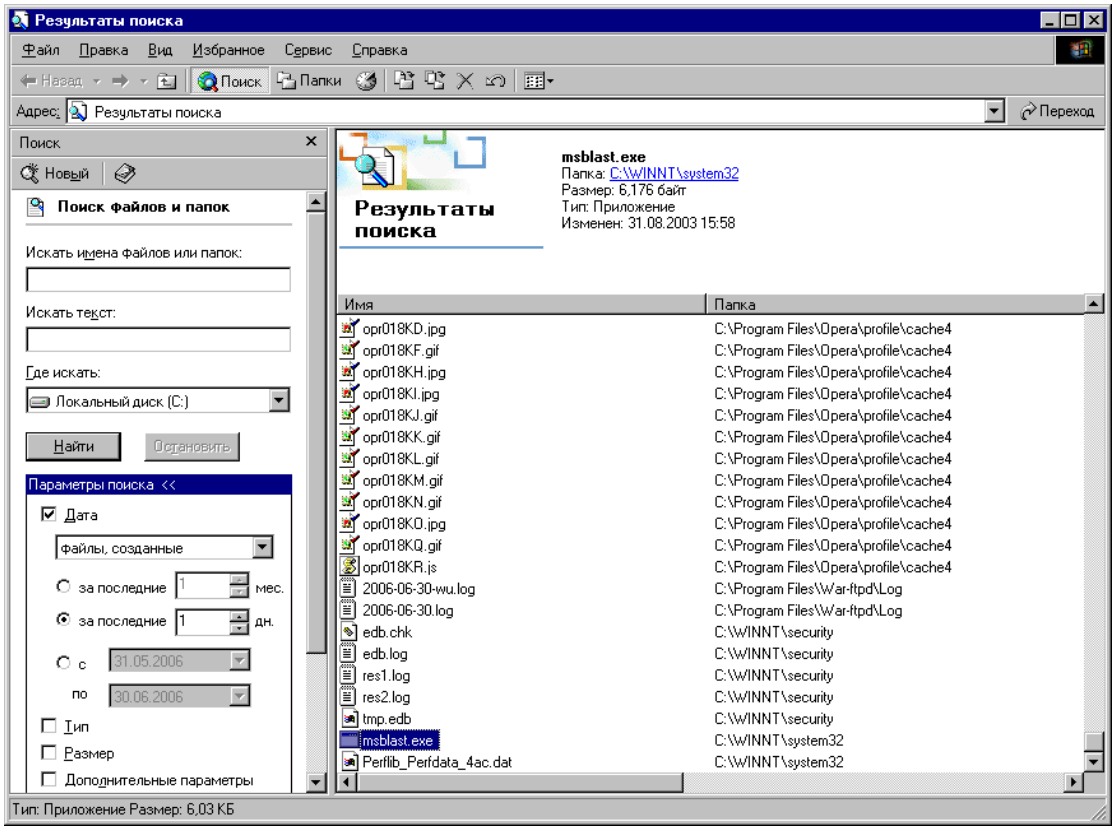

Рисунок 8 червь msblast, забывающий скорректировать дату создания файла, чем и выдающий себя с головой

Умная малварь поступает так: она считывает время создания KERNEL32.DLL (или любого другого системного файла Windows) и вызывает _стандартную_ и притом _документированную_ API-функцию SetFileTime, чтобы хоть как-то замаскироваться.

```
BOOL SetFileTime
(
        HANDLE hFile,                       // handle to the file
        CONST FILETIME *lpCreationTime      // time the file was created
        CONST FILETIME *lpLastAccessTime,   // time the file was last accessed
        CONST FILETIME *lpLastWriteTime     // time the file was last written
);
```

Листинг 1 прототип функции SetFileTime, позволяющий легальным образом манипулировать с датой создания файла

Но тут есть один нюанс. Настолько тонкий, что почти незаметный. На NTFS-разделах, с каждым файлом ассоциирован ряд скрытых атрибутов, недоступных стандартным функциям API, и среди прочей полезной информации хранящих дату создания данной файловой записи, совпадающей по времени с датой создания самого файла (подробнее см. соседнюю статью "контрразведка с soft-ice в руках"). Расхождение в датах указывает на факт подделки, не свойственный честным программам, и разоблачающий "умную малварь". Если и быть умным — то до конца! Изменив перед созданием файла системное время, а затем возвратив его обратно, малварь обеспечит себе наивысшую скрытность, и прочно оккупирует компьютер, не опасаясь быть замеченной.

грех 5 — недокументированные возможности

Использование недокументированных возможностей оправдано тогда и _только_ тогда, когда без них обойтись _невозможно_ или же создатель малвари на 100% уверен, что на всех целевых операционных системах эти возможности реализованы одинаково, что вовсе не факт. Даже установка очередного пакета обновления приводит к значительным изменениям в поведении оси.

Вот только один пример. При запуске exe-файла, доступ к нему блокируется и если он вдруг захочет себя удалить, без посторонней помощи ему это ни за что не сделать, поскольку удаление становится возможным только после снятия блокировки, то есть после завершения процесса. Конечно, можно создать bat-файл, но только это некрасиво (хоть и надежно), а можно воспользоваться недокументированной особенностью Windows 9x/NT, позволяющей освобождать страничный образ файла, снимая тем самым с него блокировку. В 9x это делается функцией `FreeLibrary`, в NT и W2k – `UnmapViewOfFile`. Правда, выполнение кода в освобожденной секции становится невозможным и любая попытка обращения к принадлежащей ей памяти возбуждает исключение. А нам еще `DeleteFile` и `ExitProcess` выполнить надо! Как быть?! Приходится разрывая себе задницу напополам, "заряжать" стек.

Конкретная реализация показана ниже. Попробуйте разобраться как _это_ вообще работает и почему (ведь с первого взгляда не должно) и только потом обращайтесь к разгадке (только чур, не подглядывать!).

```
module = GetModuleHandle(0);
GetModuleFileName(module, buf, MAX_PATH);

if(0x80000000 & GetVersion())
{
        // для Win9x
        fnFreeOrUnmap = FreeLibrary;
}
        else
{
        // для WinNT
        fnFreeOrUnmap = UnmapViewOfFile;
        CloseHandle((HANDLE)4);
}

__asm
{
        lea     eax, buf
        push    0
        push    0
        push    eax
        push    ExitProcess
        push    module
        push    DeleteFile
        push    fnFreeOrUnmap
        ret
}
```

Листинг 2 код, удаляющий текущий процесс

Начнем раскрутить головоломку с конца. Очевидно, что `ret` передает управление по адресу, который был занесен в стек перед ним, то есть вызывает функцию `fnFreeOrUnmap`, которой в зависимости от версии Windows оказывается либо `FreeLibrary`, либо `UnmapViewOfFile`. Получив управление, функция смотрит на стек и думает: ага, "`DeleteFile`" это адрес возврата, а вот "`module`" — это мой аргумент. Освободив страничный образ, она передает управление по адресу возврата (на месте которого лежит адрес `DeleteFile`) и увеличивает значение указателя стека на 4 (размер переданных ей аргументов).

Получив управление, `DeleteFile` смотри на стек и думает: ага, `ExitProcess` – это адрес возврата, а вот "`push eax`" — мой аргумент с именем файла, которая я должна удалить! (и ведь удаляет, поскольку модуль к этому времени уже освобожден).

Следующей (и последней) управление получает функция `ExitProcess`, завершающая выполнение программы, которой уже нет.

Рисунок 9 файл, удаляющий себя сам

Элегантно, не правда ли?! Никаких тебе временных bat-файлов и прочей дисковой активности (которую, кстати, могут заметить всякие недружелюбно настроенные мониторы). Но! Разве кто-нибудь гарантировал нам (документация или лично Билл Гейтс), что `UnmapViewOfFile` позволяет освобождать образ exe-файла? На NT и W2K это работало лишь потому, что ядро хранило ссылку на обработчик объекта-секции (не путать с секциями PE-файла) и `UnmapViewOfFile` послушно его освобождало. Начиная с XP ядро обращается к обработчику секции через указатель, обламывая вызов `UnmapViewOfFile`, а вместе с ним весь наш кайф.

Отсюда вывод — решение, построенное на недокументированных возможностях, может рухнуть в любой момент, поэтому, используя его необходимо как минимум предусмотреть обходной путь на тот случай, если оно не сработает.

грех 6 — незаконнорожденные потоки

Число 6 находится у язычников и сатанистов на особом почете, поэтому шестой грех будет самым объемистым, каким, он, в действительности и является.

Чтобы не порождать отдельный процесс, некоторая малварь внедряется в один из уже существующий, порождая в нем свой поток, причем делает это настолько неумело, что сразу же обращает на себя внимание и легко обнаруживается утилитой "Process Explorer" Марка Руссиновича или любым отладчиком (OlyDbg, soft-ice). А все потому, что память, в которой малварь размещает свой код, в 99% случаях выделяется через `VirtualAlloc/VirtualAllocEx`, то есть берется из динамической памяти, в то время как нормальные потоки вращаются в пределах образов исполняемых файлов или DLL.

Чтобы хоть как-то замаскировать торчащий из норы хвост, малварь должна поместить свое тело в DLL, закинуть ее в системный каталог Windows (или куда-нибудь в другой место) и загрузить внутрь атакуемого процесса через `LoadLibrary`. Естественно, делать это следует из контекста атакуемого процесса, поскольку, ни сама `LoadLibrary`, ни ее расширенная версия `LoadLibraryEx` не принимают обработчик процесса в качестве одного из своих аргументов. То есть, прежде чем загружать DLL, в процесс прежде необходимо как-то внедриться. Проще всего это сделать через уже упомянутую `VirtualAllocEx`: выделить блок в атакуемом процессе, скопировать туда код загрузчика и, либо вызвать `CreateRemoteThread`, либо скорректировать контекст активного потока, передав загрузчику управление путем вызова функции `GetThreadContext` и коррекции регистра EIP.

Получив управление, загрузчик должен вызвать `LoadLibrary` для загрузки своей DLL, вызвать `CreateThread`, указав в качестве стартового адреса одну из функции динамической библиотеки, после чего "замести следы": освободить блок памяти, выделенный `VirtualAllocEx`, завершить поток, порожденный `CreateRemoteThread` (или вернуть EIP на место). Естественно, поскольку после освобождения региона памяти исполнение оставшегося в нем кода становится невозможным, мы не сможем вызвать `TerminateThread` не схлопотав исключение, если только… не воспользуется приемом, описанным в листинге 2. На этот раз он не использует _никаких_ недокументированных возможностей и полностью законен, сохраняя

работоспособность даже на машинах с включенным механизмов DEP, препятствующим выполнению кода в стеке. Однако, никакого _кода_ в стеке у нас нет! Одни лишь адреса возврата вместе с аргументами вызываемых функций! Правда, на 64-битных версиях Windows это уже не сработает, поскольку там API-функции принимают аргументы через регистры и стек отдыхает (разумеется, сказанное относится только к 64-битным приложениям! старые, 32-битные приложения, будут работать как обычно)

Кстати, о DEP. Выделяя блок памяти с помощью `VirtualAllocEx`, присвойте ему атрибуты `PAGE_READWRITE`, а перед передачей управления смените их на `PAGE_EXECUTE` через VirtualProtectEx. На машинах без DEP, атрибуты `PAGE_READ` и `PAGE_EXECUTE` взаимно эквиваленты, поскольку процессоры старого поколения поддерживали только два атрибута страниц: ACCESS (страница доступна для чтения/исполнения или недоступна) и write (страница доступна/недоступна для записи). Атрибут EXECUTE был прерогативой таблиц селекторов (то есть сегментов). Во времена разработки 80386 никому и в голову не могло прийти, что массовая операционная система сведет все три сегмента (кода, стека и данных) в линейную память общего адресного пространства! Фактически, атрибут `PAGE_EXECUTE` был высосан разработчиками win32 из пальца и не соответствовал реальному положению вещей, но сейчас все изменилось. И хотя по умолчанию DEP включен только для системных процессов (да и то не для всех), привыкать к атрибуту `PAGE_EXECUTE` следует уже сейчас. А почему мы не установили сразу все три атрибута на страницу `PAGE_EXECUTE_READWRITE`? Ведь `VirtualProtectEx` это позволяет, да и процессор не против. _Сейчас_ да, никто не против и не возражает, но в Microsoft уже вынашивает планы по совершенствованию защиты своей оси, и попытка присвоения всех трех атрибутов будет либо вызывать исключение, либо требовать специальных привилегий.

Но все равно, создание дополнительного протока это как-то слишком заметно. Может ли малварь без него обойтись? Может! И для этого существует множество путей. Не все из них ведут в рай, но все-таки… Самое простое — внедрившись в атакуемый процесс через `VirtualAllocEx` и загрузив свою DLL, установить таймер, воспользовавшись API-функцией `SetTimer`, и периодически получать таймерные сообщения, обрабатывая их в контексте основного потока, точнее того потока, с которым ассоциирован фокус ввода. Но это уже технические делали, в которые можно не вдаваться. Главное, что новый поток _не_ создается. Правда, остается таймер… А таймер это такая штука… не самая распространенная среди прикладных приложений. Так зачем же лишний раз обращать на себя внимание?

Хитрая малварь действует очень скрытно. Получив дескриптор главного окна программы она вызывает API-функцию `GetWindowLong` с параметром `GWL_WNDPROC`, получая адрес оконной процедуры (где происходит обработка сообщений) и тут же меняет его на свой через `SetWindowLong`. Этим она не только перехватывает все сообщения (в том числе передвижения мыши и нажатия клавиш, что очень полезно для шпионской деятельности), но и гарантированно обеспечивает себя процессорным временем, не создавая ни таймеров, ни новых потоков. Правда, пытливый исследователь, вооруженный soft-ice, может забеспокоиться: с чего бы это, главное окно обрабатывается какой-то там посторонней DLL? Однако, при компонентом подходе к программированию такие случаи встречаются достаточно часто и в легальных программах, особенно, если они написаны на DELPHI.

А вот еще один путь: асинхронные сокеты — практически неиспользуемые, но очень мощные. Главное их достоинство в том, что ожидая подключения клиента или передавая/принимая данные по сети, сокет немедленно возвращает управление, сигнализируя о завершении процесса приема/передачи через специальный CALLBACK. В практическом плане это означает, что малварь может установить асинхронный сокет и тут же возвратить основному потоку управления, будучи при этом абсолютно уверенной, что в нужный момент, операционная система вспомнит о ней и передаст управление CALLBACK-процедуре, расположенной внутри загруженной малварью динамической библиотеке. Внешне все выглядит чики-чики: никаких тебе дополнительных потоков, никаких перехватов чего бы то ни было, вообще ничего подозрительного.

Рисунок 10 под атакой малвари

заключение

Перечисленными шестью "грехами" огрехи малваре-писателей конечно же не ограничиваются. Ошибок в продуктах их творчества _очень_ много и все они тупы и до неприличия однообразны. Хорошая малварь все еще встречается, но редко. И с каждым годом все реже и реже. Дизассемблировать нечего. Ковыряться в мегабайтах неаппетитного кода — неинтересно. В общем, скука смертная и никакого позитивного продвижения вперед.