основы самомодификации

крис касперски, no-email

самомодифицирующийся код встречается во многих вирусах, защитных механизмах, сетевых червях, кряксимах и прочих программах подобного типа. и хотя техника его создания не представляет большого секрета, качественных реализаций с каждым годом становится все меньше и меньше. выросло целое поколение хакеров, считающих, что самомодификация под Windows невозможна или слишком сложна. однако, в действительности это не так…

знакомство с самомодифицирующимся кодом

Окутанный мраком тайны, окруженный невообразимым количеством мифов, загадок и легенд, самомодифицирующийся код неотвратимо уходит в прошлое, медленно разлагаясь на свалке истории. Рассвет эпохи самомодификации уже позади. Во времена неинтерактивных отладчиков типа **debug.com** и пакетных дизассемблеров типа **Sourcer**, самомодификация, действительно, серьезно затрудняла анализ, однако, с появлением **IDA PRO** и **Turbo-Debugger'a** все изменилось.

Самомодификация не препятствует трассировке и для отладчика она полностью "прозрачна". Со статическим анализом дела обстоят несколько сложнее. *Дизассемблер отображает программу в том виде, в котором она была получена на момент снятия дампа или загрузки исходного файла, негласно рассчитывая на то, что ни одна из машинных команд не претерпит изменений в ходе своего выполнения.* В противном случае, реконструкция алгоритма будет выполнена неверно и хакерский корабль при спуске на воду даст колоссальную течь. Однако, если факт самомодификации будет обнаружен, скорректировать дизассемблерный листинг не составит большого труда.

Рассмотрим следующий пример:

```
    FE 05 : inc byte ptr DS:[fack_me] ; заменить jz (опкод 74 xx) на jnz (75 xx)
    33 C0   xor eax,eax               ; установить флаг нуля
fack_me:
    74 xx   jz fack_away              ; переход, если флаг нуля взведен
    E8 58 : call protect_proc         ; вызов секретной функции
```

Рисунок 1 пример нерационального использования самомодифицирующегося кода

Давайте проанализируем выделенные строки. Сначала программа обнуляет регистр EAX, устанавливая флаг нуля, а затем, если он взведен (а он взведен!) переходит к метке fack_away. На самом же деле, все происходит с точностью до наоборот. Мы упустили одну деталь. Конструкция "`INC BYTE PRT DS:[FACK_ME]`" инвертирует команду условного перехода и вместо метки fack_away управление получает процедура protect_proc. Блестящий защитный пример, не правда ли? Не хочу огорчать вас, но всякий нормальный хакер неминуемо заметит "`INC BYTE PRT DS:[FACK_ME]`" (уж слишком она бросается в глаза) и тут же разоблачит подвох.

А, что если расположить эту инструкцию совсем в другой ветке программы, далеко-далеко от модифицируемого кода? С другим дизассемблером такой фокус может быть и прокатит, но только не с IDA PRO! Взгляните на автоматически созданную ей перекрестную ссылку, ведущую непосредственно к строке "`INC BYTE PTR LOC_40100F`":

```
.text:00401005 ; ---------------------------------------------------------------------------
.text:00401005
.text:00401005 loc_401005:                             ; CODE XREF: start+AF↓p
.text:00401005                 push    ebp
.text:00401006                 mov     ebp, esp
.text:00401008                 push    ebx
.text:00401009                 push    esi
.text:0040100A                 push    edi
.text:0040100B                 db      3Eh
.text:0040100B                 inc     byte ptr loc_401014
.text:00401012                 xor     eax, eax
.text:00401014
.text:00401014 loc_401014:                             ; DATA XREF: .text:0040100B↑w
.text:00401014                 jnz     short loc_40101B ; ссылка на модифицируемый код
.text:00401016                 call    protect_proc
.text:0040101B
.text:0040101B loc_40101B:                             ; CODE XREF: .text:00401014↑j
.text:0040101B                 pop     edi
.text:0040101C                 pop     esi
.text:0040101D                 pop     ebx
.text:0040101E                 pop     ebp
.text:0040101F                 retn
```

Рисунок 2 IDA PRO автоматически распознала факт самомодификации кода

Так вот, хлопцы. Самомодификация в чистом виде ничего не решает и если не предпринять дополнительных защитных мер, ее участь предрешена. Лучшее средство борьбы с перекрестными ссылками – это учебник математики за первый класс. Без шуток! Простейшие арифметические операции с указателями ослепляют автоматический анализатор IDA PRO и перекрестные ссылки бьют мимо цели.

Обновленный вариант самомодифицирующегося кода может выглядеть, например, так:

```
    mov eax, offset fack_me + 669h ; нацеливаем eax на ложную мишень
    sub eax, 669h                  ; корректируем "прицел"
    inc byte ptr DS:[eax]          ; заменить jz на jnz
    ; //
    ; много-много строк кода
    ; //
    xor eax,eax                    ; установить флаг нуля
fack_me:
    jz fack_away                   ; переход, если флаг нуля взведен
    call protect_proc              ; вызов секретной функции
```

Рисунок 3 хитрый самомодифицирующий код, обманывающий IDA PRO

Что здесь происходит? Первым делом в регистр EAX загружаемся смещение модифицируемой команды, увеличенное на некоторую величину (условимся называть ее дельтой). Важно понять, что эти вычисления выполняются транслятором еще на стадии ассемблирования и в машинный код попадает только конечный результат. Затем из регистра EAX вычитается дельта, корректирующая "прицел" и нацеливающая EAX непосредственно на модифицируемый код. При условии, что дизассемблер не содержит в себе эмулятора ЦП и не трассирует указатели (а IDA PRO не делает ни того, ни другого), он создает единственную перекрестную ссылку, направленную на подложную мишень, расположенную далеко в стороне от театра боевых действий и никак не связанную с самомодифицирующимся кодом. Прием, *если подложная мишень будет расположена в области лежащей за пределами [Image Base; Image Base + Image Size], перекрестная ссылка вообще не будет создана!*

Вот листинг, полученный с помощью IDA PRO:

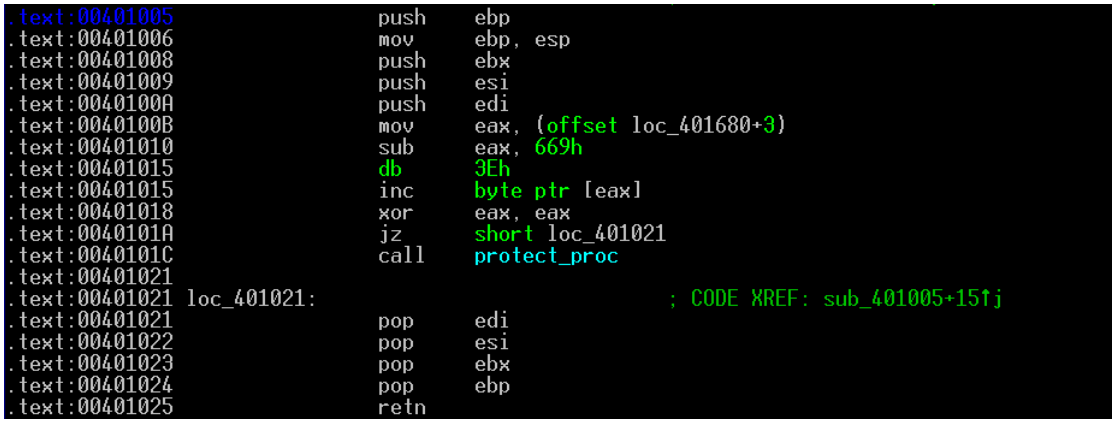

Рисунок 4 дизассемблерный листинг обманутой IDA PRO

Сгенерированная перекрестная ссылка ведет в глубину библиотечной функции _printf, случайно оказавшейся на этом месте. Сам же модифицируемый код ничем не выделяется на фоне остальных машинных команд, и взломщик на будет абсолютно уверен, что здесь находится именно "JZ", а не "JNZ"! Естественно, в данном случае это не сильно усложнит анализ, ведь защитная процедура (protect_proc) торчит у хакера под самым носом и если он не даун – обязательно полюбопытствует. Однако, если подвергнуть самомодификации алгоритм проверки серийных номеров, заменяя ROR на ROL, взломщик будет долго материться почему его хакерский генератор не срабатывает. А когда запустит отладчик – заругается еще громче, поскольку обнаружит, что его поимели, незаметно заменив одну машинную команду другой. Большинство хакеров, кстати сказать, именно так и поступают, запрягая отладчик и дизассемблер в одну упряжку.

Более прогрессивные защитные технологии базируются на динамической шифровке кода. А шифровка – одна из разновидностей самомодификации! Очевидно, что вплоть до того момента пока двоичный код не будет полностью расшифрован, для дизассемблирования он непригоден. А если расшифровщик до верху нашпигован антиотладочными приемами, непосредственная отладка становится невозможной тоже.

Статическая шифровка (характерная для большинства навесных протекторов) в настоящее время признанна совершенно бесперспективной. Дождавшись момента завершения расшифровки, хакер снимает дамп и затем исследует его стандартными средствами. Естественно, защитные механизмы так или иначе пытаются этому противостоять. Они искажают таблицу импорта, затирают PE-заголовок, устанавливают атрибуты страниц в NO_ACCESS, однако, опытных хакеров такими фокусами надолго не задержишь. Любой, даже самый изощренный, навесной протектор вручную снимется без труда, а для некоторых имеются и автоматические взломщики.

Ни в какой момент времени весь код программы не должен быть расшифрован целиком! Возьмите себе за правило – расшифровывая один фрагмент, зашифровывайте другой. Причем, расшифровщик должен быть сконструирован так, чтобы хакер не мог использовать его для расшифровки программы. Это типичная уязвимость большинства защитных механизмов. Хакер находит точку входа в расшифровщик, восстанавливает его прототип, и пропускает через него все зашифрованные блоки, получая на выходе готовый к употреблению дамп. Причем, если расшифровщик представляет собой тривиальный XOR, хакеру будет достаточно определить место хранения ключей, а расшифровать программу он сможет и сам.

Чтобы этого не случилось, защитные механизмы должны использовать полиморфные технологии и генераторы кода. Автоматизировать расшифровку программы, состоящей из нескольких сотен фрагментов, зашифрованных крипторами, сгенерированными на "лету", практически невозможно. Однако, и реализовать подобный защитный механизм отнюдь не просто. Впрочем, прежде чем воздвигать перед собой грандиозные цели, давайте лучше разберемся с основами…

принципы построения самомодифицирующегося кода

Ранние модели x86 процессоров не поддерживали когерентности машинного кода и не отслеживали попыток модификации команд уже находящихся на конвейере. С одной стороны, это усложняло разработку самомодифицирующегося кода, с другой – позволяло одурачить отладчик, работающий в трассирующем режиме.

Продемонстрируем это на следующем примере:

```
    MOV AL, 90h
    LEA DI, fack_me
    STOSB
fack_me:
    INC AL
```

Рисунок 5 модификация машинной команды, уже находящейся на конвейере

При прогоне программы на живом процессоре, инструкция INC AL заменяется на NOP, однако, поскольку INC AL уже находится на конвейере, регистр AL все-таки увеличивается на единицу. Пошаговая трассировка программы ведет себя иначе. Отладочное исключение, сгенерированное непосредственно после выполнения инструкции STOSB, очищает конвейер и управление получает уже не INC AL, а NOP, вследствие чего увеличения регистра AL уже не происходит! Если значение AL используется для расшифровки программы, то отладчик скажет хакеру fuck!

Процессоры семейства Pentium отслеживают модификацию команд уже находящихся на конвейере и потому программная длина конвейера равна нулю. Как следствие – защитные механизмы конвейерного типа, попав на Pentium, ошибочно полагают, что всегда исполняются под отладчиком. Это вполне документированная особенность поведения процессора, рассчитывающая сохраниться и в последующих моделях. Использование самомодифицирующегося кода с формальной точки зрения вполне законно. Однако, следует помнить, что чрезмерное злоупотребление последним отрицательно влияет на производительность защищаемого приложения. Кодовый кэш первого уровня доступен только на чтение и пряма запись в него невозможна. При модификации машинных команд в памяти, в действительности модифицируется кэш данных! Затем происходит экстренный сброс кодового кэша и перезагрузка измененных кэш-линеек, на что расходуется достаточно большое количество процессорных тактов. Никогда не выполняйте самомодифицирующийся код в глубоко вложенном цикле, если конечно, вы не хотите затормозить свою программу до скорости асфальтового катка!

Ходят слухи, что самомодифицирующийся код возможен только в MS-DOS, а Windows запрещает нам это делать. И хотя какая-то доля правды в этом есть, при желании мы можем обойти все запреты и ограничения. Прежде всего разберемся с атрибутами доступа к страницам и сегментам. x86-процессоры поддерживают три атрибута для доступа к сегментам (чтение, запись и исполнение) и два – к страницам (доступ и запись). *Операционные системы семейства Windows, совмещают кодовой сегмент с сегментом данных в едином адресном пространстве, а потому атрибуты чтения и исполнения для них полностью эквивалентны.*

Исполняемый код может быть расположен в любой доступной области памяти – стеке, куче, области глобальных переменных и т. д. Стек с кучей по умолчанию доступы для записи и вполне пригодны для размещения самомодифицирующегося кода. Константные глобальные и статические переменные обычно размещаются в секции .rdata, доступной только на чтение (ну и на исполнение разумеется тоже), и всякая попытка их модификации завершается неизменным исключением.

Таким образом, все, что нам нужно – скопировать самомодифицирующийся код в стек (кучу) и там он сможет харакирить себя как захочет. Рассмотрим следующий пример:

```
// определяем размер самомодифицирующейся функции
#define SELF_SIZE      ((int) x_self_mod_end - (int) x_self_mod_code)

// начало самомодифицирующейся функции
// спецификатор naked, поддерживаемый компилятором MS VC,
// указывает компилятору на необходимость создания чистой
// ассемблерной функции, т.е. такой функции, куда компилятор
// не внедряет никакой посторонней отсебятины
__declspec( naked ) int x_self_mod_code(int a, int b )
{
__asm{
        begin_sm:                       ; начало самомодифицирующегося кода
                mov eax, [esp+4]        ; получаем первый аргумент
                call get_eip            ; определяем свое текущее положение в памяти
        get_eip:
                add eax, [esp + 8 + 4] ; складываем/вычитаем из первого аргумента второй
                pop edx                 ; в edx адрес начала инструкции add eax, …
                xor byte ptr [edx],28h ; меняем add на sub и наоборот
                ret                     ; возвращаемся в материнскую функцию
```

```
        }
} x_self_mod_end(){/* конец самомодифицирующейся функции */ }

main()
{

        int a;
        int (__cdecl *self_mod_code)(int a, int b);

        // раскомментируйте следующую строку, чтобы убедиться, что непосредственная
        // самомодификация под Windows невозможна (система выплюнет исключение)
        // self_mod_code(4,2);

        // выделяем память из кучи (в куче модификация кода разрешена)
        // с таким же успехом мы могли бы выделить память из стека:
        // self_mod_code[SELF_SIZE];
        self_mod_code = (int (__cdecl*)(int, int)) malloc(SELF_SIZE);

        // копируем самомодифицирующийся код в стек/кучу
        memcpy(self_mod_code, x_self_mod_code, SELF_SIZE);

        // вызываем самомодифицирующуюся процедуру 10 раз
        for (a = 1;a< 10;a++) printf("%02X ", self_mod_code(4,2)); printf("\n");
}
```

Листинг 1 самомодификация кода на стеке/куче

Самомодифицирующийся код заменяет машинную команду "ADD" на "SUB", а "SUB" на "ADD" и потому циклический вызов функции self_mod_code возвращает следующую последовательность чисел: "06 02 06 02…", подтверждая тем самым успешное завершение акта самомодификации (не путать с дефекацией и дефлорацией!).

Некоторые находят предложенную технологию слишком громоздкой. Некоторых возмущает то, что копируемый код должен быть полностью перемещаем, т. е. сохраняющим свою работоспособность независимо от текущего местоположения в памяти. Код, сгенерированный компилятором, в общем случае таковым не является, что вынуждает нас спускаться на чисто ассемблерный уровень. Каменный век! Неужели со времен неандертальцев, добывающих огонь трением и костяным шилом превращающих перфокарту в дуршлаг, программисты не додумались до более прогрессивных методик?! А як же!

Давайте для разнообразия попробуем создать простейшую зашифрованную процедуру, написанную полностью на языке высокого уровня (например, Си, хотя те же самые приемы пригодны и для Паскаля с его уродливым родственником DELHI). При этом мы будем исходить из следующих предположений: а) порядок размещения функций в памяти совпадает с очередностью их объявления в программе (практически все компиляторы так и поступают); б) шифруемая функция не содержит перемещаемых элементов, так же называемых fixup'ами или релокациями[1] (это справедливо для большинства исполняемых файлов, но динамическим библиотекам без релокаций никуда).

Для успешной расшифровки процедуры нам необходимо определить стартовый адрес ее размещения в памяти. Это легко. Современные языки высокого уровня поддерживают операции с указателями на функцию. На Си/Си++ это выглядит приблизительно так: "`void *p; p = (void*) func;`". Сложнее измерять длину функции. Это вам не удав и попугаи тут не подходят. Легальные средства языка не предоставляют такой возможности и приходится хитрить, определяя длину как разность двух указателей: указателя на зашифрованную функцию и указателя на функцию, расположенную непосредственно за ее концом. Разумеется, если компилятору захочется нарушить естественный порядок следования функций, этот прием не сработает и расшифровка пойдет прахом. Так что держите свой хакерский хвост в боевом состоянии!

И последнее. Ни один из всех известных мне компиляторов не позволяет генерировать зашифрованный код и эту операцию приходится осуществлять вручную – с помощью HIEW'a или своих собственных утилит. Но как мы найдем шифруемую функцию в двоичном файле? Хакеры используют несколько конкурирующих способов, в зависимости от ситуации отдавая предпочтение то одному, то другому из них.

В простейшем случае шифруемая функция окантовывается ***маркерами***, – уникальными байтовыми последовательностями, гарантированно не встречающимися в остальных частях программы. Обычно маркеры задается с помощью директивы _emit, представляющей собой

[1] от английского relocation – перемещение

аналог ассемблерного DB. Например, следующая конструкция создает текстовую строку "KPNC" – `__asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'`. Только не пытайтесь располагать маркеры *внутри* шифруемой функции. Процессор не поймет юмора и выплюнет исключение. Накаливайте маркеры на вершину и дно функции, но не трогайте ее тело!

Выбор алгоритма шифрования непринципиален. Кто-то использует XOR, кто-то тяготеет к DES или RSA. Естественно, XOR ломается намного проще, особенно если длина ключа невелика. Однако, в демонстрационном примере, приведенном ниже, мы остановимся именно на XOR, поскольку DES/RSA крайне громоздки и совершенно ненаглядны:

```
#define CRYPT_LEN ((int)crypt_end - (int)for_crypt)

// маркер начала
mark_begin(){__asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'}

// зашифрованная функция
for_crypt(int a, int b)
{
        return a + b;
} crypt_end(){}

// маркер конца
mark_end (){__asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'}


// расшифровщик
crypt_it(unsigned char *p, int c)
{
        int a;  for (a = 0; a < c; a++) *p++ ^= 0x66;
}

main()
{
        // расшифровываем защитную функцию
        crypt_it((unsigned char*) for_crypt, CRYPT_LEN);

        // вызываем защитную функцию
        printf("%02Xh\n",for_crypt(0x69, 0x66));

        // зашифровываем опять
        crypt_it((unsigned char*) for_crypt, CRYPT_LEN);
}
```

Листинг 2 самомодификация на службе шифрования

Откомпилировав эту программу обычным образом (например, "`cl.exe /c FileName.C`"), мы получим объектный файл FileName.obj. Теперь нам необходимо скомпоновать исполняемый файл, предусмотрительно отключив защиту кодовой секции от записи. В линкере Microsoft Link за это отвечает ключ /SECTION, за которым идет имя секции и назначаемые ей атрибуты, например, "`link.exe FileName.obj /FIXED /SECTION:.text,ERW`". Здесь: /FIXED – ключ, удаляющий перемещаемые элементы (мы ведь помним, что перемещаемые элементы необходимо удалять?[2]), ".text" – имя кодовой секции, а "ERW" – это первые буквы Executable, Readable, Writable, хотя при желании Executable можно и опустить – на работоспособность файла это никак не повлияет. Другие линкеры используют свои ключи, описание которых может быть найдено в документации. Имя кодовой секции не всегда совпадает с ".text", поэтому если у вас что-то не получается, используйте утилиту MS DUMPBIN для выяснения конкретных обстоятельств.

Сформированный линкером файл еще не пригоден для запуска, ведь защищенная функция еще не зашифрована! Чтобы ее зашифровать, запустим HIEW, переключимся в HEX-режим и запустим контекстный поиск маркерной строки (<F7>, "KPNC", <ENTER>). Вот она! (см. рис.6). Теперь остается лишь зашифровать все, что расположено внутри маркеров "KPNC". Нажимая <F3>, мы переходим в режим редактирования, а затем давим <F8> и задаем маску шифрования (в данном случае она равна 66h). Каждое последующее нажатие на <F8> зашифровывает один байт, перемещая курсор по тексту. <F9> – сохраняет изменения на диске.

[2]при линковке исполняемых файлов, MS link автоматически подставляет этот ключ по умолчанию, так что если мы забудем его употребить ничего ужасного не случиться

После того, как файл будет зашифрован, потребность в маркерах отпадает и при желании их можно затереть бессмысленным кодом, чтобы защищенная процедура поменьше бросалась в глаза.

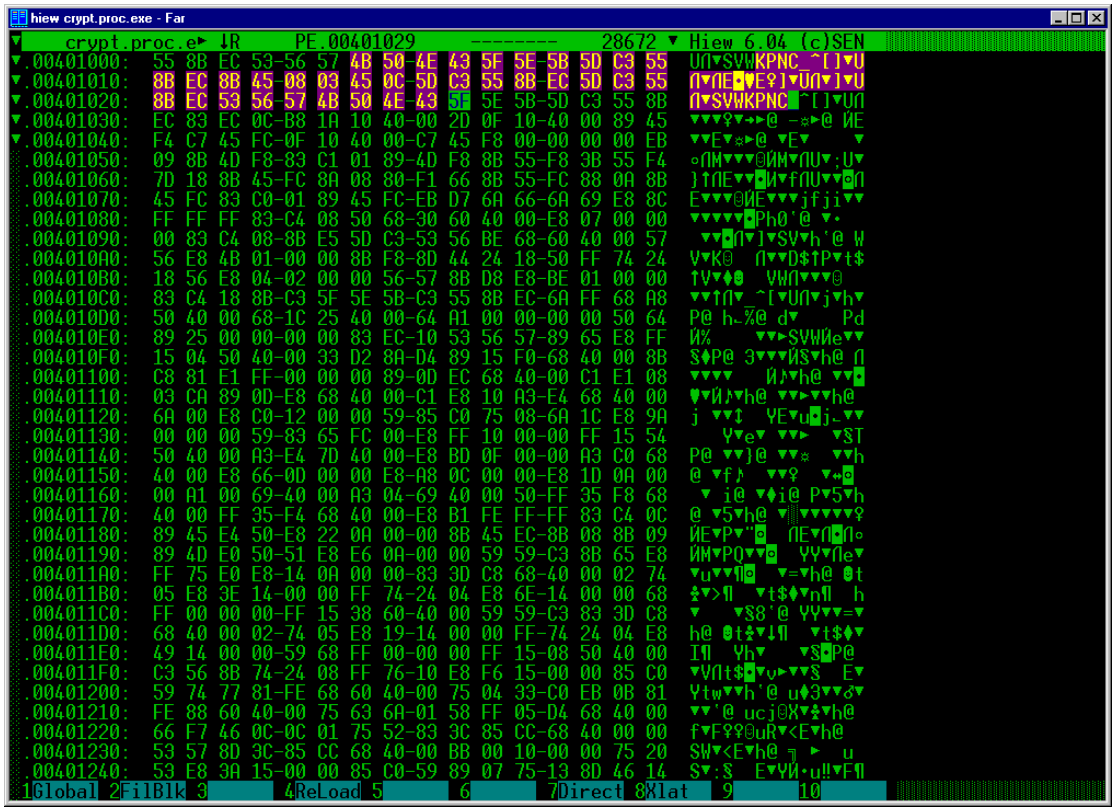

Рисунок 6 шифровка защитной процедуры в HIEW'e

Вот теперь наш файл готов к выполнению. Запустим его и… конечно же, он откажет в работе. Что ж! Первый блин всегда комом, особенно если тесто замешено на самомодифицирующемся коде! Призвав на помощь отладчик, здравый смыл и дизассемблер, попытайтесь определить, что именно вы сделали неправильно. Как говориться "Everybody falls the first time" (c) The Matrix.

Добившись успеха, загрузим исполняемый файл в IDA PRO и посмотрим как выглядит зашифрованная функция. Бред сивой кобылы да и только:

Рисунок 7 внешний вид защифрованной процедуры

Естественно, заклятье наложенной шифровки легко снять (опытные хакеры сделают это, даже не выходя из IDA PRO – подробности см. в "Образ мышления IDA"), так что не стоит переоценивать степень своей защищенности. К тому же, защита кодой секции от записи была придумана неслучайно и ее отключение разумным действием не назовешь.

API-функция **`VirtualProtect`** позволяет манипулировать атрибутами страниц по нашему усмотрению. С ее помощью мы можем присваивать атрибут Writeable только тем страницам которые реально нуждаются в модификации и сразу же после завершения расшифровки отбирать его обратно.

Обновленный вариант функции crypt_it может выглядеть например, так:

```
crypt_it(unsigned char *p, int c)
{

        int a;

        // отключаем защиту от записи
        VirtualProtect(p, c, PAGE_READWRITE, (DWORD*) &a);

        // расшифровываем функцию
        for (a = 0; a < c; a++) *p++ ^= 0x66;

        // восстанавливаем защиту
        VirtualProtect(p, c, PAGE_READONLY, (DWORD*) &a);
}
```

Листинг 3 использование VirtualProtect для кратковременного отключения защиты от записи на локальном участке

Откомпилировав файл обычным образом, зашифруйте его по методике описанной выше, и запустите на выполнение. Будем надеяться, что он заработает с первого раза.

проблемы обновления кода через интернет

Техника самомодификации тесно связана с задачей автоматического обновления кода через Интернет. Это очень сложная задача, требующая обширных знаний и инженерного мышления. Вот неполный перечень подводных камней с которыми нам придется столкнуться: как встроить двоичный код в исполняемый файл? как оповестить все экземпляры удаленной программы о факте обновления? как защитится от поддельных обновлений? По-хорошему эта тема требует отдельной книги, здесь же мы можем лишь очертить проблему.

Начнем с того, что концепции модульного и процедурного программирования (без которых сейчас никуда) нуждаются в определенных механизмах межпроцедурного

взаимодействия. По меньшей мере одна процедура должна уметь вызывать другую. Вот например:

```
my_func()
{
    printf("fuck away and piss off\n");
}
```

Рисунок 8 классический метод вызова функций делает код неперемещаемым

Что здесь неправильно? А вот что! Функция printf находится вне функции my_func и ее адрес наперед неизвестен. В обычной жизни эту задачу решает линкер, однако, мы же ведь не собираемся встраивать его в обновляемую программу, верно? Поэтому, необходимо разработать собственный механизм импорта/экспорта всех необходимых функций. Не пугайтесь! Это намного легче запрограммировать чем произнести.

В простейшем случае будет достаточно передать нашей функции указатели на все необходимые ей функции как аргументы, тогда она не будет привязана к своему местоположению в памяти и станет полностью перемещаема (см. рисунок 9). Глобальные и статические переменные и константные строки использовать запрещается (компилятор размещает их в другой секции). Так же необходимо убедиться, что компилятор в порядке проявления собственной инициативы не впендюрил в код никакой отсебятины наподобие вызова функций, контролирующих границы стека на предмет переполнения. Впрочем, в большинстве случаев такая инициатива легко отключается через ключи командной строки, описанных в прилагаемой к компилятору документации.

```
my_func(void *f1, void *f2, void *f3, void *f4, void *f5;)
{
    int (__cdecl *f_1)(int a);
    //...
    f_1 = (int (__cdecl*)(int))f1;
    //...
    f_1(0x666);
}
```

Рисунок 9 вызов функций по указателям, переданным через аргумент, обеспечивает коду перемещаемость

Откомпилировав полученный файл, мы должны скомпоновать его в 32-разрядный бинарник. Далеко не каждый линкер способен на такое и зачастую двоичный код выдирается из исполняемого файла любым подручным HEX-редактором (например, тем же HIEW'ом).

ОК, мы имеем готовый модуль обновления, имеем обновляемую программу. Остается только добавить первое ко второму. Поскольку, Windows блокирует запись во все исполняющиеся в данный момент файлы, обновить сам себя файл не может и эту операцию приходится выполнять в несколько стадий. Сначала исполняемый файл (условно обозначенный нами как A) переименовывает себя в файл B (переименованию запущенных файлов Windows не мешает), затем файл B создает свою копию под именем A, дописывает модуль обновления в его конец как оверлей (более опытные хакеры могут скорректировать значение поля ImageSize), после чего завершает свое выполнение, передавая бразды правления файлу A, удаляющему временный файл B с диска. Разумеется, это не единственно возможная схема и, кстати говоря, далеко не самая лучшая из всех, но на первых порах сойдет и она.

Более актуальным представляется вопрос распространения обновлений по Интернету. Но почему бы просто не выкладывать обновления на такой-то сервер? Путь удаленные приложения (например, те же черви) периодически посещают его, вытягивая свежачок… Ну и сколько такой сервер просуществует? Если он не рухнет под натиском бурно размножающихся червей, его закроет разъяренный администратор. Нет! Тут необходимо действовать строго по распределенной схеме.

Простейший алгоритм выглядит так: пусть каждый червь сохраняет в своем теле IP-адреса заражаемых машин, тогда "родители" будут знать своих "детей", а "дети" помнить "родителей" вплоть до последнего колена. Впрочем, обратное утверждение неверно. "Дедушки" и "бабушки" знают лишь своих непосредственных "детей", но не имеют никакого представления о внуках, если, конечно, внуки явным образом не установят с ними соединение и не сообщат свои адреса… Главное – рассчитать интенсивность обмена информацией так, чтобы не сожрать весь сетевой трафик. Тогда, обновив одного червя, мы сможем достучаться и до всех остальных, причем, противостоять этому будет очень и очень непросто. Распределенная система

обновлений не имеет единого координационного центра и даже будучи уничтоженной на 99,999% сохраняет свою работоспособность.

Правда, для борьбы с червями может быть запущенно обновление-камикадзе, автоматически уничтожающее всех червей, которые успели его заголить. Поэтому, прогрессивно настроенные вирусописатели активно используют механизмы цифровой подписи и несимметричные криптоалгоритмы. Если лень разрабатывать свой собственный движок, можно использовать PGP (благо ее исходные тексты открыты).

Главное – иметь идеи и уметь держать компилятор с отладчиком в руках. Все остальное – вопрос времени.

заключение

Без притока новых идей техника самомодификации обречена на вымирание. Чтобы поддержать ее на плаву, необходимо найти правильную точку применения сил, используя самомодифицирующийся код там и только там, где его действительно нужно использовать!

>>>> выноски

- самомодифицирующийся код возможен только на компьютерах Фон-Немоновской архитектуры (одни и те же ячейки памяти в различное время могут трактоваться и как код, и как данные);
- представители процессоров племени Pentium в действительности построены по Гавардской архитектуре (код и данные обрабатываются раздельно) и Фон-Неймоновскую они только эмулируют, поэтому самомодифицирующийся код резко снижает их производительность;
- фанаты ассемблера уверяют, что ассемблер поддерживает самомодифицирующийся код. это неверно! у ассемблера нет никаких средств для работы с самомодифицирующимся кодом, кроме директивны DB. ха! такая с позволения сказать "поддержка" есть и в Си!