Практический реверс-инжиниринг

Русский перевод практического руководства по x86, x64, ARM, ядру Windows, средствам обратной разработки, автоматизации отладки и обфускации.

Больше крутых материалов в моём телеграм-канале @grogrigs

Глава 1. x86 и x64

x86 - это архитектура с порядком байтов от младшего к старшему, основанная на процессоре Intel 8086. В рамках этой главы под x86 понимается 32-разрядная реализация архитектуры Intel (IA-32), определённая в руководстве Intel Software Development Manual. В общем случае она может работать в двух режимах: реальном и защищённом. Реальный режим - это состояние процессора сразу после включения питания; в нём поддерживается только 16-разрядный набор команд. Защищённый режим - это состояние процессора, в котором поддерживаются виртуальная память, страничная организация памяти и другие возможности; именно в этом состоянии работают современные операционные системы. 64-разрядное расширение архитектуры называется x64, или x86-64. В этой главе рассматривается архитектура x86, работающая в защищённом режиме.

x86 поддерживает разделение привилегий посредством абстракции, называемой уровнем кольца. Процессор поддерживает четыре уровня колец, пронумерованные от 0 до 3. (Кольца 1 и 2 обычно не используются, поэтому здесь они не рассматриваются.) Кольцо 0 соответствует наивысшему уровню привилегий и позволяет изменять все системные параметры. Кольцо 3 соответствует самому низкому уровню привилегий и позволяет читать и изменять лишь часть системных параметров. Поэтому в современных операционных системах разделение привилегий пользовательского режима и ядра обычно реализуется путём выполнения приложений пользовательского режима в кольце 3, а ядра - в кольце 0. Уровень кольца кодируется в регистре CS и в официальной документации иногда называется текущим уровнем привилегий (current privilege level, CPL).

В этой главе архитектура x86/IA-32 рассматривается в том виде, в котором она определена в томах 1-3 документа *Intel 64 and IA-32 Architectures Software Developer's Manual* (www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html).

Набор регистров и типы данных

При работе в защищённом режиме архитектура x86 располагает восемью 32-разрядными регистрами общего назначения (general-purpose registers, GPR): EAX, EBX, ECX, EDX, EDI, ESI, EBP и ESP. Некоторые из них можно дополнительно разделить на 8- и 16-разрядные регистры. Указатель команд хранится в регистре EIP. Набор регистров показан на рисунке 1-1. В таблице 1-1 описаны некоторые из этих регистров общего назначения и способы их использования.

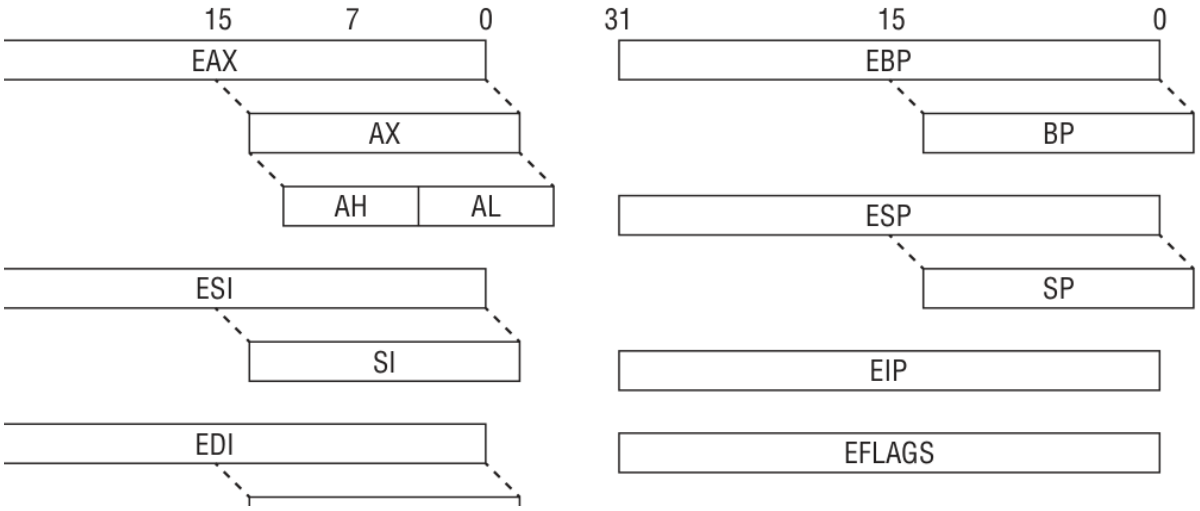

Рисунок 1-1. Набор регистров x86

Таблица 1-1. Некоторые регистры общего назначения и их применение

Регистр	Назначение
ECX	Счётчик в циклах
ESI	Источник в операциях со строками и памятью
EDI	Приёмник в операциях со строками и памятью
EBP	Базовый указатель кадра стека
ESP	Указатель стека

Распространены следующие типы данных:

• Байт - 8 бит. Примеры: AL, BL, CL.
• Слово - 16 бит. Примеры: AX, BX, CX.
• Двойное слово - 32 бита. Примеры: EAX, EBX, ECX.
• Учетверённое слово - 64 бита. Хотя в x86 нет 64-разрядных регистров общего назначения, в некоторых случаях два регистра, обычно EDX:EAX, можно объединить и рассматривать как 64-разрядное значение. Например, команда RDTSC записывает 64-разрядное значение в EDX:EAX.

32-разрядный регистр EFLAGS используется для хранения состояния арифметических операций и других состояний выполнения (например, флага трассировки). Так, если результатом предыдущей операции add был ноль, флаг ZF устанавливается в 1. Флаги EFLAGS в основном применяются для реализации условных переходов.

Помимо регистров общего назначения, EIP и EFLAGS, существуют регистры, управляющие важными низкоуровневыми системными механизмами, такими как виртуальная память, прерывания и отладка. Например, CR0 определяет, включена ли страничная организация памяти; CR2 содержит линейный адрес, вызвавший ошибку страницы; CR3 содержит базовый адрес структуры данных страничной организации памяти; а CR4 управляет параметрами аппаратной виртуализации. Регистры DR0-DR7 служат для установки точек останова по обращению к памяти. Мы вернёмся к этим регистрам далее, в разделе «Системные механизмы».

> **Примечание.** Хотя регистров отладки семь, система позволяет установить только четыре точки останова по обращению к памяти (DR0-DR3). Остальные регистры используются для хранения состояния.

Существуют также регистры, зависящие от модели (model-specific registers, MSR). Как следует из названия, набор этих регистров может различаться у разных процессоров Intel и AMD. Каждый MSR определяется именем и 32-разрядным номером, а чтение и запись выполняются командами RDMSR и WRMSR. Эти регистры доступны только коду, работающему в кольце 0, и обычно используются для хранения специальных счётчиков и реализации низкоуровневых функций. Например, команда SYSENTER передаёт управление по адресу, хранящемуся в регистре IA32_SYSENTER_EIP MSR (`0x176`); обычно это адрес обработчика системных вызовов операционной системы. По мере появления MSR будут рассматриваться на протяжении всей книги.

Набор команд

Набор команд x86 обеспечивает большую гибкость при перемещении данных между регистрами и памятью. Способы перемещения можно разделить на пять общих категорий:

• непосредственное значение в регистр;
• из регистра в регистр;
• непосредственное значение в память;
• из регистра в память и наоборот;
• из памяти в память.

Первые четыре способа поддерживаются всеми современными архитектурами, а последний присущ именно x86. Классическая RISC-архитектура, такая как ARM, может читать данные из памяти и записывать их в память только командами загрузки и сохранения (соответственно LDR и STR). Например, для такой простой операции, как увеличение находящегося в памяти значения на единицу, потребуются три команды:

1. Прочитать данные из памяти в регистр (LDR).
2. Прибавить единицу к регистру (ADD).
3. Записать регистр обратно в память (STR).

В x86 для такой операции потребуется всего одна команда (INC или ADD), поскольку она может обращаться к памяти непосредственно. Команда MOVS может одновременно читать память и записывать в неё.

ARM:

```
01: 1B 68         LDR      R3, [R3]
; прочитать значение по адресу R3
02: 5A 1C         ADDS     R2, R3, #1
; прибавить к нему 1
03: 1A 60         STR      R2, [R3]
; записать изменённое значение обратно по адресу R3
```

x86:

```
01: FF 00         inc      dword ptr [eax]
; непосредственно увеличить значение по адресу EAX
```

Ещё одна важная особенность x86 заключается в переменной длине команд: команда может занимать от 1 до 15 байт. В ARM длина команды составляет 2 или 4 байта.

Синтаксис

В зависимости от ассемблера или дизассемблера код на языке ассемблера x86 может записываться в одном из двух вариантов синтаксиса: Intel или AT&T.

Intel:

```
mov ecx, AABBCCDDh
mov ecx, [eax]
mov ecx, eax
```

AT&T:

```
movl $0xAABBCCDD, %ecx
movl (%eax), %ecx
movl %eax, %ecx
```

Важно отметить, что это одни и те же команды, просто записанные по-разному. Между нотациями Intel и AT&T есть несколько различий, наиболее заметные из которых перечислены ниже.

• В AT&T перед именем регистра ставится %, а перед непосредственным значением - $. В Intel этого нет.
• В AT&T к команде добавляется суффикс, указывающий разрядность операции. Например, MOVL (длинное слово), MOVB (байт) и т. д. В Intel этого нет.
• В AT&T операнд-источник указывается перед операндом-приёмником. В Intel порядок обратный.

Дизассемблеры, ассемблеры и другие средства реверс-инжиниринга в Windows (IDA Pro, OllyDbg, MASM и т. д.) обычно используют нотацию Intel, тогда как программы в UNIX часто придерживаются нотации AT&T (GCC). На практике преобладает нотация Intel, и именно она используется во всей книге.

Перемещение данных

Команды работают со значениями, поступающими из регистров или основной памяти. Самая распространённая команда перемещения данных - MOV. В простейшем случае с её помощью содержимое регистра или непосредственное значение помещается в регистр. Например:

```
01: BE 3F 00 0F 00   mov   esi, 0F003Fh ; установить ESI = 0xF003
02: 8B F1            mov   esi, ecx     ; установить ESI = ECX
```

Другой распространённый вариант - перемещение данных в память или из памяти. Как и в других соглашениях языков ассемблера, в x86 обращение к памяти обозначается квадратными скобками ([]). (Единственное исключение - команда LEA: в ней используются [], но фактического обращения к памяти не происходит.) Обращение к памяти можно задать несколькими способами, поэтому начнём с простейшего случая.

Ассемблер:

```
01: C7 00 01 00 00+  mov   dword ptr [eax], 1
; записать 1 в память по адресу EAX
02: 8B 08            mov   ecx, [eax]
; записать в ECX значение из памяти по адресу EAX
03: 89 18            mov   [eax], ebx
; записать EBX в память по адресу EAX
04: 89 46 34         mov   [esi+34h], eax
; записать EAX в память по адресу (ESI+34)
05: 8B 46 34         mov   eax, [esi+34h]
; записать в EAX значение из памяти по адресу (EAX+34)
06: 8B 14 01         mov   edx, [ecx+eax]
; записать в EDX значение из памяти по адресу (ECX+EAX)
```

Псевдокод на C:

```
01: *eax = 1;
02: ecx = *eax;
03: *eax = ebx;
04: *(esi+34) = eax;
05: eax = *(esi+34);
06: edx = *(ecx+eax);
```

Эти примеры показывают обращение к памяти через базовый регистр и смещение, причём смещением может быть регистр или непосредственное значение. Такая форма часто используется для доступа к членам структуры или буферам данных по адресу, вычисленному во время выполнения. Предположим, например, что ECX указывает на структуру типа KDPC со следующей компоновкой:

```
kd> dt nt!_KDPC
   +0x000 Type             : UChar
   +0x001 Importance       : UChar
   +0x002 Number           : Uint2B
   +0x004 DpcListEntry     : _LIST_ENTRY
   +0x00c DeferredRoutine  : Ptr32     void
   +0x010 DeferredContext  : Ptr32 Void
   +0x014 SystemArgument1  : Ptr32 Void
   +0x018 SystemArgument2  : Ptr32 Void
   +0x01c DpcData          : Ptr32 Void
```

и используется в следующем контексте.

Ассемблер:

```
01: 8B 45 0C         mov   eax, [ebp+0Ch]
02: 83 61 1C 00      and   dword ptr [ecx+1Ch], 0
03: 89 41 0C         mov   [ecx+0Ch], eax
04: 8B 45 10         mov   eax, [ebp+10h]
05: C7 01 13 01 00+  mov   dword ptr [ecx], 113h
06: 89 41 10         mov   [ecx+10h], eax
```

Псевдокод на C:

```
KDPC *p = ...;
p->DpcData = NULL;
p->DeferredRoutine = ...;
*(int *)p = 0x113;
p->DeferredContext = ...;
```

В строке 1 значение читается из памяти и сохраняется в EAX. В строке 3 это значение присваивается полю `DeferredRoutine`. В строке 2 поле `DpcData` очищается посредством операции AND с нулём. В строке 4 из памяти читается ещё одно значение и сохраняется в EAX. В строке 6 это значение присваивается полю `DeferredContext`.

В строке 5 по базовому адресу структуры записывается значение двойного слова `0x113`. Почему по базовому адресу записывается двойное слово, если размер первого поля составляет всего 1 байт? Разве при этом не будут неявно заданы также поля `Importance` и `Number`? Да, именно так. На рисунке 1-2 показан результат преобразования `0x113` в двоичное представление.

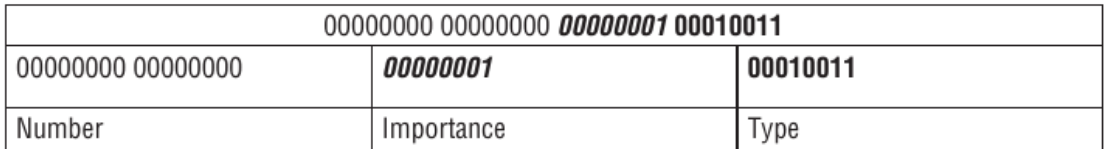

Рисунок 1-2. Размещение значения 0x113 в полях структуры

Полю Type присваивается 0x13 (биты, выделенные полужирным), полю Importance - 0x1 (подчёркнутые биты), а полю Number - 0x0 (оставшиеся биты). Записав одно значение, код сумел одной командой инициализировать три поля! Этот код можно было бы записать следующим образом:

```
01: 8B 45 0C         mov   eax, [ebp+0Ch]
02: 83 61 1C 00      and   dword ptr [ecx+1Ch], 0
03: 89 41 0C         mov   [ecx+0Ch], eax
04: 8B 45 10         mov   eax, [ebp+10h]
05: C6 01 13         mov   byte ptr [ecx],13h
06: C6 41 01 01      mov   byte ptr [ecx+1],1
07: 66 C7 41 02 00+  mov   word ptr [ecx+2],0
08: 89 41 10         mov   [ecx+10h], eax
```

Компилятор решил свернуть три команды в одну, поскольку значения констант были известны заранее и требовалось сэкономить место. Версия из трёх команд занимает 13 байт (дополнительный байт в строке 7 не показан), а версия из одной команды - 6 байт. Ещё одно интересное наблюдение: к памяти можно обращаться с тремя уровнями гранулярности - байт (строки 5-6), слово (строка 6) и двойное слово (строки 1-4 и 8). По умолчанию гранулярность составляет 4 байта, но с помощью префикса переопределения её можно изменить на 1 или 2 байта. В данном примере байты префиксов переопределения - C6 и 66 (выделены курсивом). Другие префиксы будут рассматриваться по мере их появления.

Следующая форма обращения к памяти часто применяется для доступа к объектам типа массивов. В общем виде она записывается так: `[база + индекс * масштаб]`. Лучше всего понять её на примерах:

```
01: 8B 34 B5 40 05+  mov   esi, _KdLogBuffer[esi*4]
; всегда записывается как mov esi, [_KdLogBuffer + esi * 4]
; _KdLogBuffer - базовый адрес глобального массива, а
; ESI - индекс; известно, что длина каждого элемента массива
; равна 4 байтам (отсюда и коэффициент масштабирования)

02: 89 04 F7         mov   [edi+esi*8], eax
; здесь EDI - базовый адрес массива, ESI - индекс
; массива; размер элемента равен 8.
```

На практике такая конструкция встречается в коде, перебирающем массив в цикле. Например:

```
01:                loop_start:
02: 8B 47 04         mov   eax, [edi+4]
03: 8B 04 98         mov   eax, [eax+ebx*4]
04: 85 C0            test  eax, eax
...
05: 74 14            jz    short loc_7F627F
06:                loc_7F627F:
07: 43               inc   ebx
08: 3B 1F            cmp   ebx, [edi]
09: 7C DD            jl    short loop_start
```

В строке 2 двойное слово читается со смещением +4 относительно EDI, а затем в строке 3 используется как базовый адрес массива; следовательно, можно заключить, что EDI, вероятно, указывает на структуру, в которой по смещению +4 находится массив. В строке 7 индекс увеличивается. В строке 8 индекс сравнивается со значением по смещению +0 в той же структуре. С учётом этих сведений небольшой цикл можно декомпилировать следующим образом:

```
typedef struct _FOO
{
    DWORD size;        // +0x00
    DWORD array[...];  // +0x04
} FOO, *PFOO;

PFOO bar = ...;
for (i = ...; i < bar->size; i++) {
    if (bar->array[i] != 0) {
    ...
    }
}
```

Команды MOVSB, MOVSW и MOVSD перемещают данные между двумя адресами памяти с гранулярностью соответственно 1, 2 или 4 байта. В них неявно используются EDI в качестве адреса назначения и ESI в качестве адреса источника. Кроме того, адреса источника и назначения автоматически изменяются в зависимости от флага направления (direction flag, DF) в регистре EFLAGS. Если DF равен 0, адреса уменьшаются; в противном случае они увеличиваются. Эти команды обычно применяются для реализации функций копирования строк или памяти, когда длина известна во время компиляции. Иногда вместе с ними используется префикс REP, повторяющий команду до ECX раз. Рассмотрим следующий пример:

Ассемблер:

```
01: BE 28 B5 41 00   mov   esi, offset _RamdiskBootDiskGuid
; ESI = указатель на RamdiskBootDiskGuid
02: 8D BD 40 FF FF+  lea   edi, [ebp-0C0h]
; EDI - некоторый адрес в стеке
03: A5               movsd
; копирует 4 байта из EDI в ESI; увеличивает каждый адрес на 4
04: A5               movsd
; то же, что выше
05: A5               movsd
; то же, что выше
06: A5               movsd
; то же, что выше
```

Псевдокод на C:

```
/* GUID - это 16-байтовая структура */
GUID RamDiskBootDiskGuid = ...; // глобальная переменная
...
GUID foo;
memcpy(&foo, &RamdiskBootDiskGuid, sizeof(GUID));
```

Строка 2 заслуживает особого внимания. Хотя в команде LEA используются `[]`, фактически она не читает данные по адресу памяти, а лишь вычисляет выражение в квадратных скобках и помещает результат в регистр-приёмник. Например, если бы EBP был равен `0x1000`, то после выполнения строки 2 EDI был бы равен `0xF40` (`=0x1000 - 0xC0`). Главное здесь в том, что, несмотря на вводящий в заблуждение синтаксис, LEA не обращается к памяти.

В следующем примере из `nt!KiInitSystem` используется префикс REP:

```
01: 6A 08            push   8    ; поместить 8 в стек (стеки будут
                                 ; объяснены позднее)
02: ...
03: 59               pop    ecx  ; извлечь значение из стека. По сути,
                                 ; установить ECX равным 8.
04: ...
05: BE 00 44 61 00   mov    esi, offset _KeServiceDescriptorTable
06: BF C0 43 61 00   mov    edi, offset _KeServiceDescriptorTableShadow
07: F3 A5            rep movsd   ; скопировать 32 байта (повторить movsd 8 раз)
; отсюда можно заключить, что, чем бы ни были эти два объекта,
; их размер, вероятно, составляет 32 байта.
```

Приблизительный эквивалент на C выглядел бы так:

```
memcpy(&KeServiceDescriptorTableShadow, &KeServiceDescriptorTable, 32);
```

В последнем примере, взятом из `nt!MmInitializeProcessAddressSpace`, используется сочетание этих команд, поскольку размер копируемых данных не кратен 4:

```
01: 8D B0 70 01 00+  lea   esi, [eax+170h]
; EAX, вероятно, является базовым адресом структуры. Вспомните,
; что мы говорили о LEA...
02: 8D BB 70 01 00+  lea   edi, [ebx+170h]
; EBX, вероятно, является базовым адресом другой структуры того же типа
03: A5               movsd
04: A5               movsd
05: A5               movsd
06: 66 A5            movsw
07: A4               movsb
```

После строк 1-2 можно заключить, что EAX и EBX, вероятно, указывают на объекты одного типа, поскольку один используется как источник, другой - как приёмник, а смещение в обоих случаях одинаково. Этот фрагмент кода просто копирует 15 байт из одного поля структуры в другое. Обратите внимание: тот же код можно было записать с помощью команды MOVSB с префиксом REP, установив ECX равным 15. Однако это было бы неэффективно, поскольку привело бы к 15 операциям чтения вместо всего пяти.

К другому классу команд перемещения данных с неявно заданными источником и приёмником относятся SCAS и STOS. Подобно MOVS, эти команды могут работать с гранулярностью 1, 2 или 4 байта. SCAS неявно сравнивает AL, AX или EAX с данными, начинающимися по адресу памяти EDI; EDI автоматически увеличивается или уменьшается в зависимости от бита DF в EFLAGS. В силу своей семантики SCAS обычно применяется вместе с префиксом REP для поиска байта, слова или двойного слова в буфере. Например, функцию C `strlen()` можно реализовать следующим образом:

```
01: 30 C0            xor    al, al
; установить AL в 0 (байт NUL). Такая конструкция XOR reg, reg
; часто встречается в коде.
02: 89 FB            mov    ebx, edi
; сохранить исходный указатель на строку
03: F2 AE            repne  scasb
; последовательно просматривать строку по одному байту, пока AL не совпадёт
; с байтом по адресу EDI; когда команда завершится, это будет означать,
; что достигнут байт NUL в буфере строки
04: 29 DF            sub    edi, ebx
; теперь EDI указывает на байт NUL. Вычесть из него исходный указатель,
; чтобы получить длину.
```

STOS действует так же, как SCAS, но записывает значение AL, AX или EAX по адресу EDI. Обычно она используется для заполнения буфера постоянным значением (например, в `memset()`). Рассмотрим пример:

```
01: 33 C0            xor    eax, eax
; установить EAX в 0
02: 6A 09            push   9
; поместить 9 в стек
03: 59               pop    ecx
; извлечь его обратно в ECX. Теперь ECX = 9.

04: 8B FE            mov    edi, esi
; задать адрес назначения
05: F3 AB            rep stosd
; записать 36 нулевых байтов в буфер назначения (повторить STOSD 9 раз)
; это эквивалентно memset(edi, 0, 36)
```

LODS - ещё одна команда из того же семейства. Она читает значение размером 1, 2 или 4 байта по адресу ESI и сохраняет его соответственно в AL, AX или EAX.

Упражнение

1. В этой функции для выполнения работы используется сочетание SCAS и STOS. Сначала объясните, каковы типы `[EBP+8]` и `[EBP+C]` соответственно в строках 1 и 8. Затем объясните, что делает этот фрагмент.

```
01: 8B 7D 08         mov   edi, [ebp+8]
02: 8B D7            mov   edx, edi
03: 33 C0            xor   eax, eax
04: 83 C9 FF         or    ecx, 0FFFFFFFFh
05: F2 AE            repne scasb
06: 83 C1 02         add   ecx, 2
07: F7 D9            neg   ecx
08: 8A 45 0C         mov   al, [ebp+0Ch]
09: 8B FA            mov   edi, edx
10: F3 AA            rep stosb
11: 8B C2            mov   eax, edx
```

Арифметические операции

Основные арифметические операции - сложение, вычитание, умножение и деление - поддерживаются набором команд непосредственно. Поразрядным операциям AND, OR, XOR, NOT, а также сдвигам влево и вправо тоже соответствуют отдельные машинные команды. За исключением умножения и деления, применение остальных команд не представляет сложности. Эти операции поясняются следующими примерами:

```
01: 83 C4 14         add   esp, 14h          ; esp = esp + 0x14
02: 2B C8            sub   ecx, eax          ; ecx = ecx - eax
03: 83 EC 0C         sub   esp, 0Ch          ; esp = esp - 0xC
04: 41               inc   ecx               ; ecx = ecx + 1
05: 4F               dec   edi               ; edi = edi - 1
06: 83 C8 FF         or    eax, 0FFFFFFFFh   ; eax = eax | 0xFFFFFFFF
07: 83 E1 07         and   ecx, 7            ; ecx = ecx & 7
08: 33 C0            xor   eax, eax          ; eax = eax ^ eax
09: F7 D7            not   edi               ; edi = ~edi
10: C0 E1 04         shl   cl, 4             ; cl = cl << 4
11: D1 E9            shr   ecx, 1            ; ecx = ecx >> 1
12: C0 C0 03         rol   al, 3             ; циклически сдвинуть AL влево на 3 позиции
13: D0 C8            ror   al, 1             ; циклически сдвинуть AL вправо на 1 позицию
```

Команды сдвига влево и вправо (строки 11-12) заслуживают некоторых пояснений, поскольку часто встречаются в реальном коде. Обычно они применяются для оптимизации операций умножения и деления, в которых множитель или делитель является степенью двойки. Такой вид оптимизации иногда называют понижением мощности операций (strength reduction), поскольку вычислительно дорогая операция заменяется более дешёвой. Например, целочисленное деление - сравнительно медленная операция, но если делитель является степенью двойки, её можно свести к сдвигу битов вправо: `100/2` эквивалентно `100>>1`. Аналогичным образом умножение на степень двойки можно свести к сдвигу битов влево: `100*2` эквивалентно `100<<1`.

Умножение без знака и со знаком выполняется соответственно командами MUL и IMUL. Общий вид команды MUL таков: `MUL регистр/память`. Иными словами, она может работать только со значениями в регистре или памяти. Значение регистра умножается на AL, AX или EAX, а результат, в зависимости от разрядности операнда, сохраняется в AX, DX:AX или EDX:EAX. Например:

```
01: F7 E1          mul   ecx                ; EDX:EAX = EAX * ECX
02: F7 66 04       mul   dword ptr [esi+4]  ; EDX:EAX = EAX * dword_at(ESI+4)
03: F6 E1          mul   cl                 ; AX = AL * CL
04: 66 F7 E2       mul   dx                 ; DX:AX = AX * DX
```

Рассмотрим ещё несколько конкретных примеров:

```
01: B8 03 00 00 00   mov   eax,3          ; установить EAX=3
02: B9 22 22 22 22   mov   ecx,22222222h  ; установить ECX=0x22222222
03: F7 E1            mul   ecx            ; EDX:EAX = 3 * 0x22222222 =
                                          ; 0x66666666
                                          ; следовательно, EDX=0, EAX=0x66666666
04: B8 03 00 00 00   mov   eax,3          ; установить EAX=3
05: B9 00 00 00 80   mov   ecx,80000000h  ; установить ECX=0x80000000
06: F7 E1            mul   ecx            ; EDX:EAX = 3 * 0x80000000 =
                                          ; 0x180000000
                                          ; следовательно, EDX=1, EAX=0x80000000
```

При 32-разрядном умножении результат сохраняется в EDX:EAX потому, что он потенциально может не поместиться в один 32-разрядный регистр, как показано в строках 4-6.

У IMUL есть три формы:

• `IMUL регистр/память` - то же, что MUL;
• `IMUL регистр1, регистр2/память` - `регистр1 = регистр1 * регистр2/память`;
• `IMUL регистр1, регистр2/память, непосредственное_значение` - `регистр1 = регистр2 * непосредственное_значение`.

Некоторые дизассемблеры сокращают список параметров. Например:

```
01: F7 E9            imul  ecx        ; EDX:EAX = EAX * ECX
02: 69 F6 A0 01 00+  imul  esi, 1A0h  ; ESI = ESI * 0x1A0

03: 0F AF CE         imul  ecx, esi   ; ECX = ECX * ESI
```

Деление без знака и со знаком выполняется соответственно командами DIV и IDIV. Они принимают только один параметр (делитель) и имеют следующий вид: `DIV/IDIV регистр/память`. В зависимости от размера делителя команда DIV использует в качестве делимого AX, DX:AX или EDX:EAX, а полученные частное и остаток сохраняются соответственно в AL/AH, AX/DX или EAX/EDX. Например:

```
01: F7 F1            div   ecx     ; EDX:EAX / ECX, частное в EAX,
02: F6 F1            div   cl      ; AX / CL, частное в AL, остаток в AH
03: F7 76 24         div   dword ptr [esi+24h] ; см. строку 1

04: B1 02            mov   cl,2    ; установить CL = 2
05: B8 0A 00 00 00   mov   eax,0Ah ; установить EAX = 0xA
```

```
06: F6 F1            div   cl      ; AX/CL = A/2 = 5 в AL (частное),
                                   ; AH = 0 (остаток)
07: B1 02            mov   cl,2    ; установить CL = 2
08: B8 09 00 00 00   mov   eax,09h ; установить EAX = 0x9
09: F6 F1            div   cl      ; AX/CL = 9/2 = 4 в AL (частное),
                                   ; AH = 1 (остаток)
```

Операции со стеком и вызов функций

Стек - фундаментальная структура данных в языках программирования и операционных системах. Например, локальные переменные в C хранятся в пространстве стека функции. Когда операционная система переходит из кольца 3 в кольцо 0, она сохраняет в стеке сведения о состоянии. Концептуально стек представляет собой структуру данных типа «последним пришёл - первым вышел», поддерживающую две операции: помещение в стек и извлечение из стека. Поместить в стек означает положить что-либо на его вершину, а извлечь из стека - удалить элемент с вершины. Если говорить конкретно, в x86 стек - это непрерывная область памяти, на которую указывает ESP и которая растёт в сторону уменьшения адресов. Операции помещения в стек и извлечения из него выполняются командами PUSH и POP, которые неявно изменяют ESP. Команда PUSH уменьшает ESP, а затем записывает данные по адресу, на который указывает ESP; POP читает данные и увеличивает ESP. По умолчанию значение автоматического увеличения или уменьшения равно 4, но с помощью префикса переопределения его можно изменить на 1 или 2. На практике оно почти всегда равно 4, поскольку операционная система требует выравнивания стека по границе двойного слова.

Предположим, что изначально ESP указывает на `0xb20000` и имеется следующий код:

```
; начальное значение ESP = 0xb20000
01: B8 AA AA AA AA  mov    eax,0AAAAAAAAh
02: BB BB BB BB BB  mov    ebx,0BBBBBBBBh
03: B9 CC CC CC CC  mov    ecx,0CCCCCCCCh
04: BA DD DD DD DD  mov    edx,0DDDDDDDDh
05: 50              push   eax
; по адресу 0xb1fffc будет находиться значение 0xAAAAAAAA, а ESP
; станет равен 0xb1fffc (=0xb20000-4)

06: 53              push   ebx
; по адресу 0xb1fff8 будет находиться значение 0xBBBBBBBB, а ESP
; станет равен 0xb1fff8 (=0xb1fffc-4)
07: 5E              pop    esi
; ESI будет содержать значение 0xBBBBBBBB, а ESP станет равен 0xb1fffc
; (=0xb1fff8+4)
08: 5F              pop    edi
; EDI будет содержать значение 0xAAAAAAAA, а ESP станет равен 0xb20000
; (=0xb1fffc+4)
```

Компоновка стека показана на рисунке 1-3.

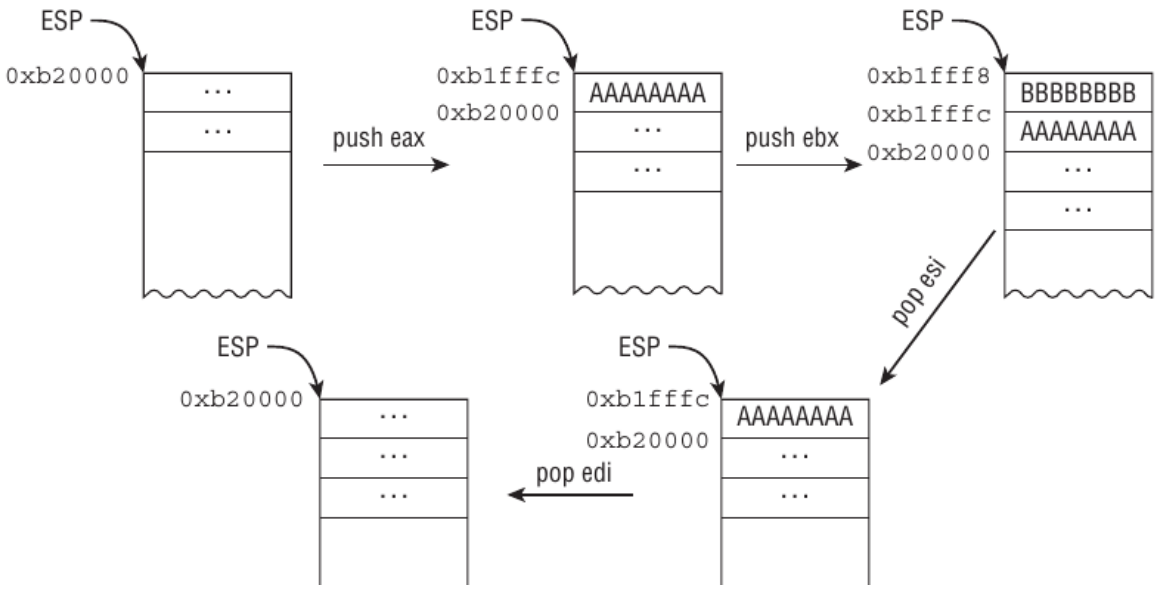

Рисунок 1-3. Изменение стека командами PUSH и POP

ESP можно изменять непосредственно и другими командами, например ADD и SUB. Хотя в языках программирования высокого уровня существует понятие функций, которые можно вызывать и из которых можно возвращаться, процессор такой абстракции не предоставляет. На самом низком уровне процессор работает только с конкретными объектами, такими как регистры или данные, поступающие из памяти. Как же функции представляются на машинном уровне? Они реализуются с помощью структуры данных стека! Рассмотрим следующую функцию.

C:

```
int
__cdecl addme(short a, short b)
{
    return a+b;
}
```

Ассемблер:

```
01: 004113A0 55           push   ebp

02: 004113A1 8B EC        mov    ebp, esp
03: ...
04: 004113BE 0F BF 45 08  movsx  eax, word ptr [ebp+8]
05: 004113C2 0F BF 4D 0C  movsx  ecx, word ptr [ebp+0Ch]
06: 004113C6 03 C1        add   eax, ecx
07: ...
08: 004113CB 8B E5        mov    esp, ebp
09: 004113CD 5D           pop    ebp
10: 004113CE C3           retn
```

Функция вызывается следующим кодом.

C:

```
sum = addme(x, y);
```

Ассемблер:

```
01: 004129F3 50              push  eax
02: ...
03: 004129F8 51              push  ecx
04: 004129F9 E8 F1 E7 FF FF  call  addme
05: 004129FE 83 C4 08        add   esp, 8
```

Прежде чем переходить к подробностям, сначала рассмотрим команды CALL и RET и соглашения о вызовах. Команда CALL выполняет две операции:

1. Помещает в стек адрес возврата (адрес сразу после команды CALL).
2. Записывает в EIP адрес назначения вызова. Тем самым управление фактически передаётся цели вызова и начинается выполнение кода по этому адресу.

RET просто извлекает хранящийся на вершине стека адрес в EIP и передаёт туда управление (буквально как `POP EIP`, хотя такой последовательности команд в x86 не существует). Например, если требуется начать выполнение по адресу `0x12345678`, достаточно сделать следующее:

```
01: 68 78 56 34 12   push  0x12345678
02: C3               ret
```

Соглашение о вызовах - это набор правил, определяющих работу вызовов функций на машинном уровне. Оно задаётся двоичным интерфейсом приложений (Application Binary Interface, ABI) конкретной системы. Например, следует ли передавать параметры через стек, в регистрах или обоими способами? В каком порядке передавать параметры: слева направо или справа налево? Где хранить возвращаемое значение: в стеке, регистрах или и там и там? Соглашений о вызовах много, но наиболее распространены CDECL, STDCALL, THISCALL и FASTCALL. (Компилятор также может создать собственное нестандартное соглашение о вызовах, однако здесь такие соглашения рассматриваться не будут.) Их семантика обобщена в таблице 1-2.

Таблица 1-2. Соглашения о вызовах

	CDECL	**STDCALL**	**FASTCALL**
Параметры	Помещаются в стек справа налево. После вызова стек должен очистить вызывающий код.	То же, что CDECL, за исключением того, что стек должна очистить вызываемая функция.	Первые два параметра передаются в ECX и EDX. Остальные - в стеке.
Возвращаемое значение	Хранится в EAX.	Хранится в EAX.	Хранится в EAX.
Сохраняемые регистры	EBP, ESP, EBX, ESI, EDI.	EBP, ESP, EBX, ESI, EDI.	EBP, ESP, EBX, ESI, EDI.

Теперь вернёмся к фрагменту кода и разберём, как вызывается функция `addme`. В строках 1 и 3 два параметра помещаются в стек; ECX и EAX являются соответственно первым и вторым параметрами. В строке 4 функция `addme` вызывается командой CALL. При этом в стек немедленно помещается адрес возврата `0x4129FE`, после чего начинается выполнение по адресу `0x4113A0`. Компоновка стека после выполнения строки 4 показана на рисунке 1-4.

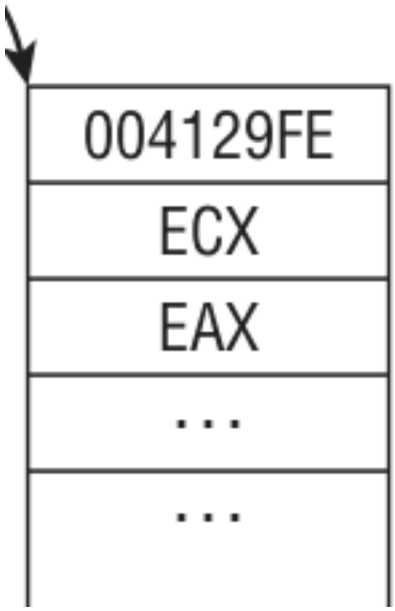

Рисунок 1-4. Стек после вызова функции addme

После выполнения строки 4 мы оказываемся в теле функции `addme`. В строке 1 EBP помещается в стек. В строке 2 регистру EBP присваивается текущее значение указателя стека. Эта последовательность из двух команд обычно называется прологом функции, поскольку она создаёт новый кадр функции. В строке 4 читается значение по адресу `EBP+8` - первый параметр в стеке; в строке 5 читается второй параметр. Обратите внимание: доступ к параметрам осуществляется с использованием EBP в качестве базового регистра. В таком контексте EBP называется базовым указателем кадра (см. строку 2), поскольку он указывает на кадр стека текущей функции, а доступ к параметрам и локальным переменным можно получать относительно него. Посредством оптимизации, называемой исключением указателя кадра (frame pointer omission), компилятору также можно предписать создавать код, в котором EBP не используется как базовый указатель кадра. При такой оптимизации доступ к локальным переменным и параметрам выполняется относительно ESP, а EBP можно использовать как регистр общего назначения, подобный EAX, EBX, ECX и т. д. В строке 6 числа складываются, а результат сохраняется в EAX. В строке 8 указателю стека присваивается значение базового указателя кадра. В строке 9 сохранённое в строке 1 значение EBP извлекается в EBP. Эта последовательность из двух команд обычно называется эпилогом функции, поскольку находится в конце функции и восстанавливает предыдущий кадр функции. Теперь на вершине стека находится адрес возврата, сохранённый командой CALL по адресу `0x4129F9`. В строке 10 выполняется RET, которая извлекает адрес из стека и возобновляет выполнение по адресу `0x4129FE`. В строке 5 фрагмента размер стека уменьшается на 8, поскольку по правилам соглашения о вызовах CDECL стек должен очищать вызывающий код.

Если бы в функции `addme` имелись локальные переменные, после строки 2 коду потребовалось бы увеличить стек, вычтя значение из ESP. После этого все локальные переменные были бы доступны по отрицательным смещениям относительно EBP.

Упражнения

1. Исходя из того, что вы узнали о CALL и RET, объясните, как можно прочитать значение EIP. Почему нельзя просто выполнить `MOV EAX, EIP`?
2. Придумайте не менее двух последовательностей команд, устанавливающих EIP в `0xAABBCCDD`.
3. Что произошло бы в функции `addme` из примера, если бы перед выполнением RET указатель стека не был правильно восстановлен?
4. Во всех рассмотренных соглашениях о вызовах возвращаемое значение хранится в 32-разрядном регистре EAX. Что происходит, если возвращаемое значение не помещается в 32-разрядный регистр? Напишите программу, чтобы провести эксперимент и проверить свой ответ. Изменяется ли этот механизм от компилятора к компилятору?

Поток управления

В этом разделе описывается, как система реализует условное выполнение для таких высокоуровневых конструкций, как `if/else`, `switch/case` и `while/for`. Все они реализуются с помощью команд CMP, TEST, JMP и Jcc и регистра EFLAGS. Ниже перечислены часто используемые флаги EFLAGS:

• ZF, флаг нуля - устанавливается, если результат предыдущей арифметической операции равен нулю.
• SF, флаг знака - принимает значение старшего бита результата.
• CF, флаг переноса - устанавливается, когда для представления результата требуется перенос. Применяется к беззнаковым числам.
• OF, флаг переполнения - устанавливается, если результат выходит за пределы максимального размера. Применяется к знаковым числам.

Арифметические команды изменяют эти флаги в соответствии с результатом. Например, команда `SUB EAX, EAX` установит ZF. Команды Jcc, где `cc` означает код условия, изменяют поток управления в зависимости от этих флагов. Всего может существовать до 16 кодов условий; наиболее распространённые из них описаны в таблице 1-3.

Таблица 1-3. Распространённые коды условий

Код условия	Описание на естественном языке	Машинное условие
B/NAE	Меньше / не больше и не равно. Используется для беззнаковых операций.	`CF=1`
NB/AE	Не меньше / больше или равно. Используется для беззнаковых операций.	`CF=0`
E/Z	Равно / ноль.	`ZF=1`
NE/NZ	Не равно / не ноль.	`ZF=0`
L	Меньше / не больше и не равно. Используется для знаковых операций.	`(SF ^ OF) = 1`
GE/NL	Больше или равно / не меньше. Используется для знаковых операций.	`(SF ^ OF) = 0`
G/NLE	Больше / не меньше и не равно. Используется для знаковых операций.	`((SF ^ OF) | ZF) = 0`

Поскольку в языке ассемблера нет определённой системы типов, коды условий служат одним из немногих способов распознать знаковые и беззнаковые типы.

Команда CMP сравнивает два операнда и устанавливает соответствующий код условия в EFLAGS; она сравнивает два числа, вычитая одно из другого, но не сохраняя результат. Команда TEST действует аналогично, однако выполняет над двумя операндами логическую операцию AND.

If-Else

Конструкции `if-else` распознаются довольно легко, поскольку содержат сравнение или проверку, за которыми следует Jcc. Например:

Ассемблер:

```
01:   mov   esi, [ebp+8]
02:   mov   edx, [esi]
03:   test  edx, edx
04:   jz    short loc_4E31F9
05:   mov   ecx, offset _FsRtlFastMutexLookasideList
06:   call  _ExFreeToNPagedLookasideList@8
07:   and   dword ptr [esi], 0
08:   lea   eax, [esi+4]
09:   push  eax
10:   call  _FsRtlUninitializeBaseMcb@4
11: loc_4E31F9:

12:   pop   esi
13:   pop   ebp
14:   retn  4
15: _FsRtlUninitializeLargeMcb@4 endp
```

Псевдокод на C:

```
if (*esi == 0) {
  return;
}
ExFreeToNPagedLookasideList(...);
*esi = 0;
...
return;
```

ИЛИ:

```
if (*esi != 0) {
  ...
  ExFreeToNPagedLookasideList(...);
  *esi = 0;
  ...
}
return;
```

В строке 2 значение по адресу ESI читается и сохраняется в EDX. В строке 3 над EDX и им самим выполняется операция AND, после чего в EFLAGS устанавливаются соответствующие флаги. Обратите внимание: такая конструкция часто применяется для проверки регистра на равенство нулю. Если `ZF=1`, строка 4 переходит к `loc_4E31F9` (строка 12). Если `ZF=0`, выполняется строка 5, а затем выполнение продолжается до возврата из функции.

Обратите внимание: этот фрагмент можно перевести на C двумя немного различающимися, но логически эквивалентными способами.

Switch-Case

Блок `switch-case` представляет собой последовательность операторов `if/else`. Например:

Switch-Case:

```
switch(ch) {
    case 'c':
        handle_C();
        break;
    case 'h':
        handle_H();
        break;
    default:
        break;
}
domore();
...
```

If-Else:

```
if (ch == 'c') {
    handle_C();
} else
if (ch == 'h') {
    handle_H();
}
domore();
...
```

Следовательно, машинный код будет представлять собой последовательность операторов `if/else`. Идею иллюстрирует следующий простой пример.

Ассемблер:

```
01:   push   ebp
02:   mov    ebp, esp
03:   mov    eax, [ebp+8]
04:   sub    eax, 41h
05:   jz     short loc_caseA
06:   dec    eax
07:   jz     short loc_caseB
08:   dec    eax
09:   jz     short loc_caseC
10:   mov    al, 5Ah
11:   movzx  eax, al
12:   pop    ebp
13:   retn
14: loc_caseC:
15:   mov    al, 43h
16:   movzx  eax, al
17:   pop    ebp
18:   retn
19: loc_caseB:
20:   mov    al, 42h
21:   movzx  eax, al
22:   pop    ebp
23:   retn
24: loc_caseA:
25:   mov    al, 41h
26:   movzx  eax, al
27:   pop    ebp
28:   retn
```

C:

```
unsigned char switchme(int a)
{
    unsigned char res;

    switch(a) {
    case 0x41:
        res = 'A';
        break;
    case 0x42:
        res = 'B';
        break;
    case 0x43:
        res = 'C';
        break;
    default:
        res = 'Z';
        break;
    }
    return res;
}
```

В реальных программах операторы `switch-case` могут быть сложнее, и для уменьшения количества сравнений и условных переходов компиляторы часто строят таблицу переходов. По существу, таблица переходов - это массив адресов, каждый из которых указывает на обработчик определённого варианта. Такую конструкцию можно наблюдать в образце J, в функции `sub_10001110`.

Ассемблер:

```
01:   cmp     edi, 5
02:   ja      short loc_10001141
03:   jmp     ds:off_100011A4[edi*4]
04: loc_10001125:
05:   mov     esi, 40h
06:   jmp     short loc_10001145
07: loc_1000112C:
08:   mov     esi, 20h
09:   jmp     short loc_10001145
10: loc_10001133:
11:   mov     esi, 38h
12:   jmp     short loc_10001145
13: loc_1000113A:
14:   mov     esi, 30h
15:   jmp     short loc_10001145
16: loc_10001141:
17:   mov     esi, [esp+0Ch]
18: ...
19: off_100011A4 dd offset loc_10001125
20:   dd offset loc_10001125
21:   dd offset loc_1000113A
22:   dd offset loc_1000112C
23:   dd offset loc_10001133
24:   dd offset loc_1000113A
```

Псевдокод на C:

```
switch(edi) {
  case 0:
  case 1:
    // перейти к loc_10001125;
    esi = 0x40;
    break;
  case 2:
  case 5:
    // перейти к loc_1000113A;
    esi = 0x30;
    break;
  case 3:
    // перейти к loc_1000112C;
    esi = 0x20;
    break;
  case 4:
    // перейти к loc_10001133;
    esi = 0x38;
    break;
  default:
    // перейти к loc_10001141;
    esi = *(esp+0xC)
    break;
}
...
```

В данном случае компилятору известно, что существует всего пять вариантов, а их значения идут последовательно; следовательно, он может построить таблицу переходов и обращаться к ней непосредственно по индексу (строка 3). Без таблицы переходов потребовалось бы 10 дополнительных команд для проверки каждого варианта и перехода к его обработчику. (Существуют и другие варианты оптимизации `switch/case`, однако здесь они рассматриваться не будут.)

Циклы

На машинном уровне циклы реализуются сочетанием команд Jcc и JMP. Иными словами, для их реализации используются конструкции `if/else` и `goto`. Лучше всего это понять, переписав цикл только с помощью `if/else` и `goto`. Рассмотрим следующий пример.

С использованием `for`:

```
for (int i=0; i<10; i++) {
    printf("%d\n", i);
}
printf("done!\n");
```

С использованием `if/else` и `goto`:

```
int i = 0;
loop_start:
    if (i < 10) {
        printf("%d\n", i);
        i++;
        goto loop_start;
    }
printf("done!n");
```

После компиляции обе версии оказываются идентичными на уровне машинного кода:

```
01: 00401002   mov    edi, ds:__imp__printf
02: 00401008   xor    esi, esi
03: 0040100A   lea    ebx, [ebx+0]
04: 00401010 loc_401010:
05: 00401010   push   esi
06: 00401011   push   offset Format                ; "%d\n"
07: 00401016   call   edi ; __imp__printf
08: 00401018   inc    esi
09: 00401019   add    esp, 8
10: 0040101C   cmp    esi, 0Ah
11: 0040101F   jl     short loc_401010
12: 00401021   push   offset aDone                 ; "done!\n"
13: 00401026   call   edi ; __imp__printf
14: 00401028   add    esp, 4
```

В строке 1 регистру EDI присваивается адрес функции `printf`. В строке 2 ESI устанавливается в 0. В строке 4 начинается цикл; однако обратите внимание, что он начинается не со сравнения. Сравнения здесь нет, поскольку компилятору известно, что счётчик инициализирован нулём (см. строку 2) и заведомо будет меньше 10, поэтому проверка пропускается. В строках 5-7 функция `printf` вызывается с нужными параметрами (спецификатором формата и нашим числом). В строке 8 число увеличивается. В строке 9 очищается стек, поскольку `printf` использует соглашение о вызовах CDECL. В строке 10 проверяется, меньше ли счётчик `0xA`. Если да, выполняется переход назад к `loc_401010`. Если счётчик не меньше `0xA`, выполнение продолжается со строки 12 и завершается вызовом `printf`.

Важно заметить, что по дизассемблированному коду удалось определить знаковый тип счётчика. В строке 11 используется код условия «меньше» (JL), поэтому сразу понятно, что сравнивались знаковые целые числа. Запомните: условия «выше/ниже» относятся к беззнаковым значениям, а «меньше/больше» - к знаковым. В образце L имеется небольшая функция `sub_1000AE3B` со следующим интересным циклом.

Ассемблер:

```
01: sub_1000AE3B proc near
02:   push    edi

03:   push    esi
04:   call    ds:lstrlenA
05:   mov     edi, eax
06:   xor     ecx, ecx
07:   xor     edx, edx
08:   test    edi, edi
09:   jle     short loc_1000AE5B
10: loc_1000AE4D:
11:   mov     al, [edx+esi]
12:   mov     [ecx+esi], al
13:   add     edx, 3
14:   inc     ecx
15:   cmp     edx, edi
16:   jl      short loc_1000AE4D
17: loc_1000AE5B:
18:   mov     byte ptr [ecx+esi], 0
19:   mov     eax, esi
20:   pop     edi
21:   retn
22: sub_1000AE3B endp
```

C:

```
char *sub_1000AE3B (char *str)
{
  int len, i=0, j=0;
  len = lstrlenA(str);
  if (len <= 0) {
    str[j] = 0;
    return str;
  }
  while (j < len) {
    str[i] = str[j];
    j = j+3;
    i = i+1;
  }
  str[i] = 0;
  return str;
}
```

Функция `sub_1000AE3B` имеет один параметр, передаваемый согласно нестандартному соглашению о вызовах (параметр хранится в ESI). В строке 2 сохраняется EDI. В строке 3 вызывается `lstrlenA` с этим параметром; следовательно, сразу становится понятно, что ESI имеет тип `char *`. В строке 5 возвращённое значение (длина строки) сохраняется в EDI. В строках 6-7 очищаются ECX и EDX. В строках 8-9 проверяется, меньше ли длина строки нуля или равна ему. Если да, управление передаётся строке 18, которая записывает 0 по адресу ECX+ESI. В противном случае выполнение продолжается со строки 11, где начинается цикл. Сначала читается символ по адресу ESI+EDX (строка 11), а затем он сохраняется по адресу ESI+ECX (строка 12). Далее EDX и ECX увеличиваются соответственно на три и на один. В строках 15-16 проверяется, меньше ли EDX длины строки; если да, выполнение возвращается к началу цикла. В противном случае выполнение продолжается со строки 18.

Сначала эта функция может показаться запутанной, однако она принимает обфусцированную строку, в которой деобфусцированное значение составлено из каждого третьего символа. Например, строка `SX]OTYFKPTY^W\\aAFKRW\\E` в действительности означает `SOFTWARE`. Назначение функции - противодействовать примитивным сканерам строк и избегать обнаружения. В качестве упражнения декомпилируйте эту функцию так, чтобы она выглядела более «естественно» (в отличие от нашего буквального перевода).

Помимо обычных конструкций Jcc, некоторые циклы можно реализовать с помощью команды LOOP. Команда LOOP выполняет блок кода до ECX раз. Например:

Ассемблер:

```
01: 8B CA         mov    ecx, edx
02:             loc_CFB8F:
03: AD            lodsd
04: F7 D0         not    eax
05: AB            stosd
06: E2 FA         loop   loc_CFB8F
```

Приблизительный код на C:

```
while (ecx != 0) {
    eax = *edi;
    edi++;
    *esi = ~eax;
    esi++;
    ecx--;
}
```

В строке 1 счётчик читается из EDX. В строке 3 начинается цикл: двойное слово по адресу памяти EDI читается и сохраняется в EAX; кроме того, EDI увеличивается на 4. В строке 4 над только что прочитанным значением выполняется операция NOT. В строке 5 изменённое значение записывается по адресу памяти ESI, а ESI увеличивается на 4. В строке 6 проверяется, равен ли ECX нулю; если нет, выполнение продолжается с начала цикла.

Системные механизмы

В предыдущих разделах объяснялись механизмы и команды, доступные коду, работающему на любом уровне привилегий. Чтобы глубже понять архитектуру, в этом разделе рассматриваются два фундаментальных механизма системного уровня: преобразование виртуальных адресов и обработка исключений и прерываний. При первом чтении этот раздел можно пропустить.

Преобразование адресов

Физическая память компьютерной системы делится на блоки размером 4 Кбайт, называемые страницами. (Размер страницы может превышать 4 Кбайт, однако другие размеры здесь рассматриваться не будут.) Адреса памяти делятся на две категории: виртуальные и физические. Виртуальными называются адреса, которые используются выполняемыми процессором командами при включённой страничной организации памяти. Например:

```
01: A1 78 56 34 12  mov   eax, [0x12345678]; прочитать память по виртуальному
                                            ; адресу 0x12345678
01: 89 08           mov   [eax], ecx       ; записать ECX по виртуальному
                                           ; адресу EAX
```

Физические адреса - это фактические ячейки памяти, которыми процессор пользуется при обращении к памяти. Перед обращением модуль управления памятью процессора (memory management unit, MMU) незаметно преобразует каждый виртуальный адрес в физический. Пользователю виртуальный адрес может казаться просто очередным числом, однако с точки зрения MMU у него есть определённая структура. В системах x86 с поддержкой расширения физических адресов (physical address extension, PAE) виртуальный адрес памяти можно разделить на индексы трёх таблиц и смещение: таблицы указателей каталогов страниц (page directory pointer table, PDPT), каталога страниц (page directory, PD), таблицы страниц (page table, PT) и записи таблицы страниц (page table entry, PTE). PDPT представляет собой массив из четырёх 8-байтовых элементов, каждый из которых указывает на PD. PD - это массив из 512 элементов размером по 8 байт, каждый из которых указывает на PT. PT - массив из 512 элементов размером по 8 байт, каждый из которых содержит PTE. Например, виртуальный адрес `0xBF80EE6B` можно представить так, как показано на рисунке 1-5.

0×BF80EE6B			
10111111 10000000 11101110 01101011			
10 (0×2)	111111 100 (0×1FC)	00000 1110 (0×E)	1110 01101011 (0×E6B)
2 bits	9 bits	9 bits	12 bits
Index into PDPT	Index into PD	Index into PT	Page offset

Рисунок 1-5. Разбиение виртуального адреса 0xBF80EE6B

8-байтовые элементы этих таблиц содержат сведения о таблицах, разрешениях памяти и других характеристиках памяти. Например, отдельные биты определяют, доступна ли страница только для чтения или также для записи, является ли она исполняемой или неисполняемой, доступна ли пользователю и т. д.

Процесс преобразования адреса строится вокруг этих трёх таблиц и регистра CR3. В CR3 хранится физический базовый адрес PDPT. В оставшейся части этого раздела пошагово разбирается преобразование виртуального адреса `0xBF80EE6B` в реальной системе (см. рисунок 1-5):

```
kd> r @cr3           ; CR3 - физический базовый адрес PDPT
cr3=085c01e0
kd> !dq @cr3+2*8 L1  ; прочитать запись PDPT с индексом 2
# 85c01f0 00000000`0d66e001
```

Согласно документации, младшие 12 бит записи PDPT отведены под флаги и зарезервированные биты, а остальные используются как физический адрес базы PD. В режиме PAE бит 63 является флагом NX, поэтому его также необходимо сбросить. В данном примере мы этого не делали, поскольку он уже равен 0. (Мы рассматриваем страницы исполняемого кода.)

```
; 0x00000000`0d66e001 = 00001101 01100110 11100000 00000001
; после очистки младших 12 бит получаем
; 0x0d66e000          = 00001101 01100110 11100000 00000000
; это означает, что база PD находится по физическому адресу 0x0d66e000
kd> !dq 0d66e000+0x1fc*8 L1   ; прочитать запись PD с индексом 0x1FC
# d66efe0 00000000`0964b063
```

И снова, согласно документации, младшие 12 бит записи PD используются для флагов и зарезервированных битов, а остальные задают базу PT:

```
; 0x0964b063 = 00001001 01100100 10110000 01100011
; после очистки младших 12 бит получаем
; 0x0964b000 = 00001001 01100100 10110000 00000000
; это означает, что база PT находится по адресу 0x0964b000
kd> !dq 0964b000+e*8 L1      ; прочитать запись PT с индексом 0xE
# 964b070 00000000`06694021
```

И опять же, младшие 12 бит можно очистить, чтобы получить базу записи страницы:

```
; 0x06694021 = 00000110 01101001 01000000 00100001
; после очистки младших 12 бит получаем
; 0x06694000 = 00000110 01101001 01000000 00000000
; это означает, что база записи страницы находится по адресу 0x06694000
kd> !db 06694000+e6b L8      ; прочитать 8 байт из записи страницы со смещением
0xE6B
# 6694e6b 8b ff 55 8b ec 83 ec 0c ..U.....[].t....    ; наши данные в этой
                                                     ; физической странице
kd> db bf80ee6b L8           ; прочитать 8 байт по виртуальному адресу
bf80ee6b  8b ff 55 8b ec 83 ec ..U.....[].t....       ; те же данные!
```

По завершении всего процесса устанавливается, что виртуальный адрес `0xBF80EE6B` преобразуется в физический адрес `0x6694E6B`.

Современные операционные системы используют этот механизм для разделения адресных пространств процессов. С каждым процессом связан свой регистр CR3, вследствие чего преобразование виртуальных адресов выполняется отдельно для каждого процесса. Именно благодаря этому каждый процесс пребывает в иллюзии собственного адресного пространства. Надеемся, что в следующий раз, когда ваша программа обратится к памяти, вы будете ценить работу процессора немного больше!

Прерывания и исключения

Здесь прерывания и исключения рассматриваются лишь кратко, поскольку полное описание их реализации приведено в главе 3 «Ядро Windows».

В современных вычислительных системах процессор обычно соединён с периферийными устройствами шиной данных, например PCI Express, FireWire или USB. Когда устройству требуется внимание процессора, оно вызывает прерывание, которое заставляет процессор приостановить текущую работу и обработать запрос устройства. Как процессор узнаёт, каким образом обрабатывать запрос? На самом верхнем уровне прерывание можно представить связанным с определённым номером, который затем используется как индекс массива указателей на функции. Получив прерывание, процессор выполняет функцию с соответствующим прерыванию индексом, а затем возобновляет выполнение с того места, где находился до прерывания. Такие прерывания называются аппаратными, поскольку создаются аппаратными устройствами. По своей природе они асинхронны.

Во время выполнения команды процессор может столкнуться с исключениями. Например, команда может вызвать ошибку деления на ноль, обратиться к недопустимому адресу или инициировать переход между уровнями привилегий. Для целей этого обсуждения исключения можно разделить на две категории: сбои (faults) и ловушки (traps). Сбой - это исправимое исключение. Например, когда процессор выполняет команду, обращающуюся к допустимому адресу памяти, но данных нет в основной памяти (они были выгружены), возникает исключение ошибки страницы. Процессор обрабатывает его, сохраняя текущее состояние выполнения, вызывая обработчик ошибки страницы для исправления исключения (путём загрузки данных в память), а затем повторно выполняя ту же команду, которая теперь уже не должна вызвать ошибку страницы. Ловушка - исключение, возникающее при выполнении особых видов команд. Например, команда SYSENTER заставляет процессор начать выполнение общего обработчика системных вызовов; после завершения обработчика выполнение возобновляется с команды, непосредственно следующей за SYSENTER. Таким образом, главное различие между сбоем и ловушкой состоит в том, откуда возобновляется выполнение. Операционные системы обычно реализуют системные вызовы с помощью механизма прерываний и исключений.

Практический разбор

Завершим главу пошаговым разбором функции, содержащей менее 100 команд. Это процедура `DllMain` из образца J. У упражнения две цели. Во-первых, оно задействует почти все понятия, рассмотренные в этой главе (за исключением `switch-case`). Во-вторых, оно знакомит с важным требованием практического реверс-инжиниринга: умением читать технические руководства и документацию в Интернете. Вот эта функция:

```
01:    ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason,
       ; LPVOID lpvReserved)
02:                _DllMain@12 proc near
03: 55               push    ebp
04: 8B EC            mov     ebp, esp
05: 81 EC 30 01 00+  sub     esp, 130h
06: 57               push    edi
07: 0F 01 4D F8      sidt    fword ptr [ebp-8]
08: 8B 45 FA         mov     eax, [ebp-6]
09: 3D 00 F4 03 80   cmp     eax, 8003F400h

10: 76 10            jbe     short loc_10001C88 (строка 18)
11: 3D 00 74 04 80   cmp     eax, 80047400h
12: 73 09            jnb     short loc_10001C88 (строка 18)
13: 33 C0            xor     eax, eax
14: 5F               pop     edi
```

```
15: 8B E5            mov     esp, ebp
16: 5D               pop     ebp
17: C2 0C 00         retn    0Ch
18:                loc_10001C88:
19: 33 C0            xor     eax, eax
20: B9 49 00 00 00   mov     ecx, 49h
21: 8D BD D4 FE FF+  lea     edi, [ebp-12Ch]
22: C7 85 D0 FE FF+  mov     dword ptr [ebp-130h], 0
23: 50               push    eax
24: 6A 02            push    2
25: F3 AB            rep stosd
26: E8 2D 2F 00 00   call    CreateToolhelp32Snapshot
27: 8B F8            mov     edi, eax
28: 83 FF FF         cmp     edi, 0FFFFFFFFh
29: 75 09            jnz     short loc_10001CB9 (строка 35)
30: 33 C0            xor     eax, eax
31: 5F               pop     edi
32: 8B E5            mov     esp, ebp
33: 5D               pop     ebp
34: C2 0C 00         retn    0Ch
35:                loc_10001CB9:
36: 8D 85 D0 FE FF+  lea     eax, [ebp-130h]
37: 56               push    esi
38: 50               push    eax
39: 57               push    edi
40: C7 85 D0 FE FF+  mov     dword ptr [ebp-130h], 128h
41: E8 FF 2E 00 00   call    Process32First
42: 85 C0            test    eax, eax
43: 74 4F            jz      short loc_10001D24 (строка 70)
44: 8B 35 C0 50 00+  mov     esi, ds:_stricmp
45: 8D 8D F4 FE FF+  lea     ecx, [ebp-10Ch]
46: 68 50 7C 00 10   push    10007C50h
47: 51               push    ecx
48: FF D6            call    esi ; _stricmp
49: 83 C4 08         add     esp, 8
50: 85 C0            test    eax, eax
51: 74 26            jz      short loc_10001D16 (строка 66)
52:                loc_10001CF0:
53: 8D 95 D0 FE FF+  lea     edx, [ebp-130h]
54: 52               push    edx
55: 57               push    edi
56: E8 CD 2E 00 00   call    Process32Next
57: 85 C0            test    eax, eax
58: 74 23            jz      short loc_10001D24 (строка 70)
59: 8D 85 F4 FE FF+  lea     eax, [ebp-10Ch]
60: 68 50 7C 00 10   push    10007C50h
61: 50               push    eax
62: FF D6            call    esi ; _stricmp
63: 83 C4 08         add     esp, 8

64: 85 C0            test    eax, eax
65: 75 DA            jnz     short loc_10001CF0 (строка 52)
66:                loc_10001D16:
67: 8B 85 E8 FE FF+  mov     eax, [ebp-118h]
68: 8B 8D D8 FE FF+  mov     ecx, [ebp-128h]
69: EB 06            jmp     short loc_10001D2A (строка 73)
70:                loc_10001D24:
71: 8B 45 0C         mov     eax, [ebp+0Ch]
72: 8B 4D 0C         mov     ecx, [ebp+0Ch]
73:                loc_10001D2A:
74: 3B C1            cmp     eax, ecx
75: 5E               pop     esi
76: 75 09            jnz     short loc_10001D38 (строка 82)
77: 33 C0            xor     eax, eax
78: 5F               pop     edi
79: 8B E5            mov     esp, ebp
80: 5D               pop     ebp
81: C2 0C 00         retn    0Ch
82:                loc_10001D38:
83: 8B 45 0C         mov     eax, [ebp+0Ch]
84: 48               dec     eax
```

```
85: 75 15            jnz     short loc_10001D53 (строка 93)
86: 6A 00            push    0
87: 6A 00            push    0
88: 6A 00            push    0
89: 68 D0 32 00 10   push    100032D0h
90: 6A 00            push    0
91: 6A 00            push    0
92: FF 15 20 50 00+  call    ds:CreateThread
93:                loc_10001D53:
94: B8 01 00 00 00   mov     eax, 1
95: 5F               pop     edi
96: 8B E5            mov     esp, ebp
97: 5D               pop     ebp
98: C2 0C 00         retn    0Ch
99:                _DllMain@12 endp
```

В строках 3-4 формируется пролог функции: сохраняется предыдущий базовый указатель кадра и устанавливается новый. Строка 5 резервирует в стеке `0x130` байт. Строка 6 сохраняет EDI. В строке 7 выполняется команда SIDT, записывающая 6-байтовый регистр IDT в заданную область памяти. Строка 8 читает двойное слово по адресу `EBP-6` и сохраняет его в EAX. В строках 9-10 проверяется, меньше ли EAX значения `0x8003F400` или равен ему. Если да, управление передаётся строке 18; в противном случае продолжается выполнение строки 11. В строках 11-12 выполняется похожая проверка, но условием служит «не ниже `0x80047400`». Если условие выполняется, управление передаётся строке 18; в противном случае продолжается выполнение строки 13. Строка 13 очищает EAX. Строка 14 восстанавливает значение EDI, сохранённое в строке 6. Строки 15-16 восстанавливают предыдущий базовый указатель кадра и указатель стека. Строка 17 прибавляет к указателю стека `0xC` байт, а затем возвращает управление вызывающему коду.

Прежде чем обсуждать следующую часть, отметим несколько особенностей первых 17 строк. Команда SIDT (строка 7) записывает содержимое регистра IDT в 6-байтовую область памяти. Что представляет собой регистр IDT? В справочном руководстве Intel/AMD сказано, что IDT - это массив из 256 записей размером по 8 байт, каждая из которых содержит указатель на обработчик прерывания, селектор сегмента и смещение. При возникновении прерывания или исключения процессор использует номер прерывания как индекс IDT и вызывает обработчик, указанный в соответствующей записи. Регистр IDT имеет размер 6 байт: старшие 4 байта содержат базовый адрес массива, или таблицы, IDT, а младшие 2 байта - предел таблицы. Теперь понятно, что строка 8 в действительности читает базовый адрес IDT. В строках 9 и 11 проверяется, находится ли базовый адрес в диапазоне (`0x8003F400`, `0x80047400`). Что особенного в этих на первый взгляд случайных константах? Поиск в Интернете показывает, что `0x8003F400` - это базовый адрес IDT в 32-разрядной Windows XP. Это можно проверить в отладчике ядра:

```
0: kd> vertarget
Windows XP Kernel Version 2600 (Service Pack 3) MP (2 procs) Free x86 compat-
ible
Built by: 2600.xpsp.080413-2111
…
0: kd> r @idtr
idtr=8003f400
0: kd> ~1
1: kd> r @idtr
idtr=bab3c590
```

Зачем код проверяет такое поведение? Одно из возможных объяснений состоит в том, что разработчик считал базовый адрес IDT, попадающий в этот диапазон, «недопустимым» либо возникающим в результате виртуализации. Если IDTR «недопустим», функция автоматически возвращает ноль. Этот код можно декомпилировать в C следующим образом:

```
typedef struct _IDTR {
    DWORD base;
    SHORT limit;
} IDTR, *PIDTR;
BOOL __stdcall DllMain (HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    IDTR idtr;
    __sidt(&idtr);
    if (idtr.base > 0x8003F400 && idtr.base < 0x80047400h) { return FALSE; }
    // строка 18
    ...

}
```

> **Примечание.** Если внимательно прочитать руководство, можно заметить, что у каждого процессора есть собственные IDT и, следовательно, IDTR. Поэтому в многоядерной системе IDTR будет различаться для каждого ядра. Очевидно, `0x8003F400` допустим только для ядра 0 в Windows XP. Если выполнение команды будет запланировано на другом ядре, значение IDTR окажется равным `0xBAB3C590`. В более поздних версиях Windows базовые адреса IDT меняются между перезагрузками, поэтому жёсткое задание базовых адресов работать не будет.

Если базовый адрес IDT выглядит допустимым, выполнение кода продолжается со строки 18. В строках 19-20 EAX очищается, а ECX устанавливается в `0x49`. В строке 21 EDI присваивается адрес `EBP-0x12C`; поскольку EBP является базовым указателем кадра, `EBP-0x12C` - адрес локальной переменной. Строка 22 записывает ноль по адресу, на который указывает `EBP-0x130`. В строках 23-24 EAX и 2 помещаются в стек. Строка 25 заполняет нулями буфер размером `0x124` байта, начинающийся с `EBP-0x12C`. В строке 26 вызывается `CreateToolhelp32Snapshot`:

```
HANDLE WINAPI CreateToolhelp32Snapshot(
  _In_  DWORD dwFlags,
  _In_  DWORD th32ProcessID
);
```

Эта функция Win32 API принимает два целочисленных параметра. Как правило, функции Win32 API следуют соглашению о вызовах STDCALL. Следовательно, параметры `dwFlags` и `th32ProcessId` равны `0x2` (строка 24) и `0x0` (строка 23). Функция перечисляет все процессы в системе и возвращает дескриптор, который будет использоваться в `Process32Next`. В строках 27-28 возвращённое значение сохраняется в EDI и сравнивается с -1. Если оно равно -1, возвращаемое значение устанавливается в 0 и функция завершается (строки 30-34); в противном случае выполнение продолжается со строки 35. В строке 36 EAX присваивается адрес локальной переменной, ранее инициализированной нулём в строке 22; строка 40 инициализирует её значением `0x128`. В строках 37-39 ESI, EAX и EDI помещаются в стек. В строке 41 вызывается `Process32First`.

Прототип функции:

```
BOOL WINAPI Process32First(
  _In_     HANDLE hSnapshot,
  _Inout_  LPPROCESSENTRY32 lppe
);
```

Соответствующее определение структуры:

```
typedef struct tagPROCESSENTRY32 {
  DWORD     dwSize;
  DWORD     cntUsage;
  DWORD     th32ProcessID;
  ULONG_PTR th32DefaultHeapID;
  DWORD     th32ModuleID;
  DWORD     cntThreads;
  DWORD     th32ParentProcessID;
  LONG      pcPriClassBase;
  DWORD     dwFlags;
  TCHAR     szExeFile[MAX_PATH];
} PROCESSENTRY32, *PPROCESSENTRY32;

00000000 PROCESSENTRY32 struc ; (sizeof=0x128)
00000000 dwSize dd ?
00000004 cntUsage dd ?
00000008 th32ProcessID dd ?

0000000C th32DefaultHeapID dd ?
00000010 th32ModuleID dd ?
00000014 cntThreads dd ?
00000018 th32ParentProcessID dd ?
0000001C pcPriClassBase dd ?
00000020 dwFlags dd ?
00000024 szExeFile db 260 dup(?)
00000128 PROCESSENTRY32 ends
```

Поскольку эта функция принимает два параметра, `hSnapshot` - это EDI (строка 39, ранее полученный от `CreateToolhelp32Snapshot` в строке 27 дескриптор), а `lppe` - адрес локальной переменной (`EBP-0x130`). Так как `lppe` указывает на структуру `PROCESSENTRY32`, сразу становится понятно, что локальная переменная по адресу `EBP-0x130` имеет тот же тип. Это также согласуется с документацией `Process32First`, где сказано, что перед вызовом функции полю `dwSize` следует присвоить размер структуры `PROCESSENTRY32` (то есть `0x128`). Теперь известно, что в строках 19-25 эта структура просто инициализировалась нулями. Кроме того, можно утверждать, что локальная переменная начинается по адресу `EBP-0x130` и заканчивается по адресу `EBP-0x8`.

Строка 42 проверяет возвращённое значение `Process32Next`. Если оно равно нулю, выполнение начинается со строки 70; в противном случае оно продолжается со строки 43. В строке 44 адрес функции `stricmp` сохраняется в ESI. В строке 45 ECX присваивается адрес локальной переменной (`EBP-0x10C`), которая оказывается полем структуры `PROCESSENTRY32` (см. предыдущий абзац). В строках 46-48 значения `0x10007C50` и ECX помещаются в стек, после чего вызывается `stricmp`. Известно, что `stricmp` принимает в качестве аргументов две символьные строки; следовательно, ECX должен указывать на поле `szExeFile` структуры `PROCESSENTRY32`, а `0x10007C50` является адресом строки:

```
.data:10007C50 65 78 70 6C 6F+Str2 db 'explorer.exe',0
```

Строка 49 очищает стек, поскольку `stricmp` использует соглашение о вызовах CDECL. Строка 50 проверяет возвращённое значение `stricmp`. Если оно равно нулю, то есть строка совпала с `"explorer.exe"`, выполнение начинается со строки 66; в противном случае оно продолжается со строки 52. Теперь строки 18-51 можно декомпилировать следующим образом:

```
HANDLE h;
PROCESSENTRY32 procentry;
h = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if (h == INVALID_HANDLE_VALUE) { return FALSE; }

memset(&procentry, 0, sizeof(PROCESSENTRY32));
procentry.dwSize = sizeof(procentry); // 0x128
if (Process32Next(h, &procentry) == FALSE) {
    // строка 70
    ...
}
if (stricmp(procentry.szExeFile, "explorer.exe") == 0) {
    // строка 66
    ...
}
// строка 52
```

Строки 52-65 почти совпадают с предыдущим блоком, но образуют цикл с двумя условиями выхода. Первое условие выполняется, когда `Process32Next` возвращает FALSE (строка 58), а второе - когда `stricmp` возвращает ноль. Строки 52-65 можно декомпилировать следующим образом:

```
while (Process32Next(h, &procentry) != FALSE) {
    if (stricmp(procentry.szExeFile, "explorer".exe") == 0)
        break;
}
```

После выхода из цикла выполнение возобновляется со строки 66. В строках 67-68 соответствующие процессу поля `th32ParentProcessID` и `th32ProcessID` структуры `PROCESSENTRY32` сохраняются в EAX и ECX, после чего выполнение продолжается со строки 37. Обратите внимание: строка 66 также является целью перехода из строки 43.

В строках 70-74 читается параметр `fdwReason` функции `DllMain` (`EBP+C`) и проверяется, равен ли он 0 (`DLL_PROCESS_DETACH`). Если да, возвращаемое значение устанавливается в 0 и функция завершается; в противном случае выполняется переход к строке 82. В строках 82-85 проверяется, превышает ли `fdwReason` единицу (то есть равен ли он `DLL_THREAD_ATTACH` или `DLL_THREAD_DETACH`). Если да, возвращаемое значение устанавливается в 1 и функция завершается; в противном случае выполнение продолжается со строки 86. В строках 86-92 вызывается `CreateThread`:

```
HANDLE WINAPI CreateThread(
  _In_opt_   LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_       SIZE_T dwStackSize,
  _In_       LPTHREAD_START_ROUTINE lpStartAddress,
  _In_opt_   LPVOID lpParameter,
  _In_       DWORD dwCreationFlags,
  _Out_opt_  LPDWORD lpThreadId
);
```

где lpStartAddress равен 0x100032D0. Этот блок можно декомпилировать следующим образом:

```
if (fdwReason == DLL_PROCESS_DETACH) { return FALSE; }
if (fdwReason == DLL_THREAD_ATTACH || fdwReason == DLL_THREAD_DETACH) {
   return TRUE; }
CreateThread(0, 0, (LPTHREAD_START_ROUTINE) 0x100032D0, 0, 0, 0);
return TRUE;
```

Проанализировав функцию, можно заключить, что исходный замысел разработчика был таким:

1. Определить, имеет ли целевая машина «нормальную» IDT.
2. Проверить, запущен ли в системе explorer.exe, то есть вошёл ли кто-либо в систему.
3. Создать основной поток, заражающий целевую машину.

Упражнения

1. Самостоятельно повторите практический разбор. Нарисуйте компоновку стека, включая параметры и локальные переменные.
2. В практическом разборе из примера мы почти взаимно однозначно перевели код на языке ассемблера в C. В качестве упражнения заново декомпилируйте всю функцию так, чтобы результат выглядел естественнее. Что можно сказать об уровне мастерства и опыте разработчика? Объясните свои выводы. Сможете ли вы выполнить работу лучше?
3. В некоторых ассемблерных листингах в имени функции имеется префикс @, за которым следует число. Объясните, когда и почему появляется такое оформление имени.
4. Реализуйте на языке ассемблера x86 следующие функции: strlen, strchr, memcpy, memset, strcmp, strset.
5. Декомпилируйте следующие процедуры ядра Windows:

• KeInitializeDpc;
• KeInitializeApc;
• ObFastDereferenceObject (и объясните используемое соглашение о вызовах);
• KeInitializeQueue;
• KxWaitForLockChainValid;
• KeReadyThread;
• KiInitializeTSS;
• RtlValidateUnicodeString.

6. Образец H. Функция sub_13846 ссылается на несколько структур, типы которых не вполне ясны. Сначала восстановите прототип функции, а затем попытайтесь реконструировать поля структур. Прочитав главу 3, вернитесь к этому упражнению и проверьте, изменилось ли ваше понимание. (Обратите внимание: этот образец предназначен для 32-разрядной Windows XP.)
7. Образец H. В функции sub_10BB6 имеется цикл, который что-то ищет. Сначала восстановите прототип функции, а затем определите типы на основании контекста. Подсказка: вероятно, стоит держать под рукой копию спецификации PE.
8. Образец H. Декомпилируйте sub_11732 и объясните, какая программная конструкция, скорее всего, использовалась в исходном коде.
9. Образец L. Объясните, что делает функция sub_1000CEA0, а затем декомпилируйте её обратно в C.
10. Если текущий уровень привилегий кодируется в CS, который может изменять код пользовательского режима, почему код пользовательского режима не может изменить CS и тем самым сменить CPL?
11. Прочитайте главу Virtual Memory в томе 3 *Intel Software Developer Manual* и в томе 2 *AMD64 Architecture Programmer's Manual: System Programming*. Самостоятельно выполните несколько преобразований виртуальных адресов в физические и проверьте результат с помощью отладчика

ядра. Объясните, как работает предотвращение выполнения данных (data execution prevention, DEP).
12. Любимая библиотека Брюса для дизассемблирования x86/x64 - BeaEngine от BeatriX (www.beaengine.org). Поэкспериментируйте с ней: напишите программу, дизассемблирующую двоичный файл начиная с его точки входа.

x64

x64 является расширением x86, поэтому большинство свойств архитектуры остаются прежними; небольшие различия касаются размера регистров и отсутствия некоторых команд (например, PUSHAD). В следующих разделах рассматриваются существенные различия.

Набор регистров и типы данных

Набор регистров содержит 18 64-разрядных регистров общего назначения и показан на рисунке 1-6. Обратите внимание: имена 64-разрядных регистров имеют префикс `R`.

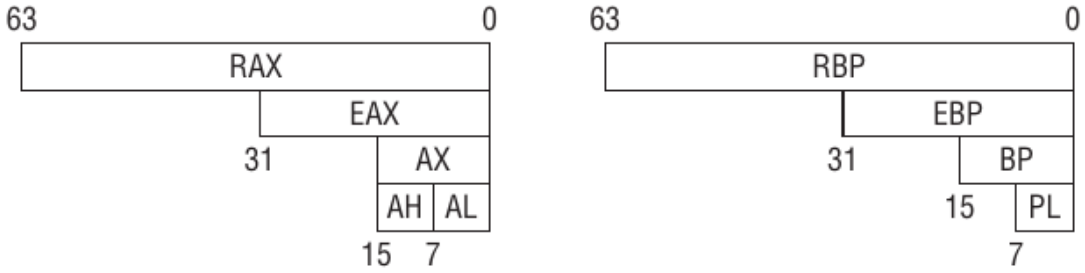

Рисунок 1-6. Регистры общего назначения x64

Хотя RBP по-прежнему можно использовать как базовый указатель кадра, в реальном коде, созданном компилятором, для этой цели он применяется редко. Большинство компиляторов x64 просто рассматривают RBP как ещё один регистр общего назначения, а к локальным переменным обращаются относительно RSP.

Перемещение данных

x64 поддерживает так называемую адресацию относительно RIP, которая позволяет командам ссылаться на данные, расположенные по смещению относительно RIP. Например:

```
01: 0000000000000000 48 8B 05 00 00+  mov     rax, qword ptr cs:loc_A
02:                                   ; первоначально записано как "mov rax,
[rip]"
03: 0000000000000007                loc_A:

04: 0000000000000007 48 31 C0         xor     rax, rax
05: 000000000000000A 90               nop
```

В строке 1 адрес `loc_A` (равный `0x7`) читается и сохраняется в RAX. Адресация относительно RIP в основном служит для создания позиционно-независимого кода.

Большинство арифметических команд автоматически расширяются до 64 разрядов, даже если операнды имеют размер всего 32 бита. Например:

```
48 B8 88 77 66+ mov   rax, 1122334455667788h
31 C0           xor   eax, eax  ; также очистит старшие 32 бита RAX,
                                ; то есть после этого RAX=0
48 C7 C0 FF FF+ mov   rax,0FFFFFFFFFFFFFFFFh
FF C0           inc   eax  ; после этого RAX=0
```

Канонический адрес

В x64 виртуальные адреса имеют ширину 64 бита, однако большинство процессоров не поддерживает полное 64-разрядное пространство виртуальных адресов. Современные процессоры Intel и AMD используют для адресного пространства только 48 бит. Все адреса виртуальной памяти должны иметь каноническую форму. Виртуальный адрес находится в канонической форме, если все биты от 63-го до старшего реализованного бита равны либо 1, либо 0. На практике это означает, что биты 48-63 должны совпадать с битом 47. Например:

```
0xfffff801`c9c11000 = 11111111 11111111 11111000 00000001 11001001 11000001
   00010000 00000000 ; канонический
0x000007f7`bdb67000 = 00000000 00000000 00000111 11110111 10111101 10110110
   01110000 00000000 ; канонический
0xffff0800`00000000 = 11111111 11111111 00001000 00000000 00000000 00000000
   00000000 00000000 ; неканонический
0xffff8000`00000000 = 11111111 11111111 10000000 00000000 00000000 00000000
   00000000 00000000 ; канонический
0xfffff960`000989f0 = 11111111 11111111 11111001 01100000 00000000 00001001
   10001001 11110000 ; канонический
```

Если код попытается разыменовать неканонический адрес, система вызовет исключение.

Вызов функций

Напомним, что некоторые соглашения о вызовах в x86 требуют передавать параметры в стеке. В x64 большинство соглашений о вызовах передают параметры через регистры. Например, в 64-разрядной Windows существует единственное соглашение о вызовах: первые четыре параметра передаются через RCX, RDX, R8 и R9, а остальные помещаются в стек справа налево. В Linux первые шесть параметров передаются через RDI, RSI, RDX, RCX, R8 и R9.

> **Примечание.** Дополнительные сведения об ABI x64 в Windows см. в разделе «x64 Software Conventions» на MSDN: http://msdn.microsoft.com/en-us/library/7kcdt6fy.aspx.

Упражнения

1. Объясните два способа получить указатель команд в x64. По меньшей мере один из них должен использовать адресацию относительно RIP.
2. Выполните преобразование виртуального адреса в физический в x64. Были ли существенные отличия от x86?

Глава 2. ARM

В конце 1980-х годов компания Acorn Computers разработала 32-разрядную RISC-архитектуру под названием Acorn RISC Machine (позднее переименованную в Advanced RISC Machine). Эта архитектура оказалась полезной далеко за пределами ограниченной линейки продукции компании, поэтому для лицензирования архитектуры и её применения в самых разных изделиях была создана компания ARM Holdings. Она широко используется во встраиваемых устройствах, таких как мобильные телефоны, автомобильная электроника, MP3-проигрыватели, телевизоры и т. д. Первая версия архитектуры появилась в 1985 году, а на момент написания книги актуальной была версия 7 (ARMv7). ARM разработала целый ряд конкретных ядер (например, ARM7, ARM7TDMI, ARM926EJS и Cortex), которые не следует путать с различными спецификациями архитектуры, пронумерованными от ARMv1 до ARMv7. Хотя версий несколько, большинство устройств основано на ARMv4, 5, 6 или 7. ARMv4 и v5 сравнительно «стары», но в то же время это самые распространённые и преобладающие версии процессора (согласно маркетинговым материалам ARM, существует «более 10 миллиардов» таких ядер). В популярных потребительских электронных устройствах обычно применяются более новые версии архитектуры. Например, Apple iPod Touch и iPhone третьего поколения работают на микросхеме ARMv6, а более поздние iPhone, iPad и устройства с Windows Phone 7 - на ARMv7.

В то время как Intel и AMD сами проектируют и производят процессоры, ARM придерживается несколько иной модели. ARM разрабатывает архитектуру и лицензирует её другим компаниям, которые затем производят процессоры и встраивают их в свои устройства. Такие компании, как Apple, NVIDIA, Qualcomm и Texas Instruments, продают собственные процессоры (соответственно A, Tegra, Snapdragon и OMAP), но лицензируют их базовую архитектуру у ARM. Все они реализуют основной набор команд и модель памяти, определённые в справочном руководстве по архитектуре ARM. Процессор можно дополнять расширениями; например, расширение Jazelle позволяет выполнять байт-код Java непосредственно на процессоре. Расширение Thumb добавляет команды шириной 16 или 32 бита, тем самым повышая плотность кода (собственные команды ARM всегда имеют ширину 32 бита). Расширение Debug позволяет инженерам анализировать физический процессор с помощью специального отладочного оборудования. Каждое расширение обычно обозначается буквой (J, T, D и т. д.). В зависимости от своих требований производитель может решить, нужны ли ему лицензии на эти дополнительные расширения. Именно поэтому после обозначений процессоров ARMv6 и более ранних версий стоят буквы (например, ARM1156T2 означает ARMv6 с расширением Thumb-2). В ARMv7 эти соглашения больше не применяются: вместо них используются три профиля (Application, Real-time и Microcontroller) и название модели Cortex с различными возможностями. Например, процессоры серии ARMv7 Cortex-A имеют прикладной профиль, а Cortex-M предназначены для микроконтроллеров и поддерживают выполнение только в состоянии Thumb.

В этой главе рассматривается архитектура ARMv7 в том виде, в котором она определена в документе *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition* (ARM DDI 0406B).

Основные особенности

Поскольку ARM является RISC-архитектурой, между ARM и CISC-архитектурами (x86/x64) есть несколько основных различий. (С практической точки зрения новые версии процессоров Intel тоже обладают некоторыми особенностями RISC, то есть не являются «чистыми» CISC.) Во-первых, набор команд ARM очень мал по сравнению с x86, зато регистров общего назначения больше. Во-вторых, команды имеют фиксированную ширину (16 или 32 бита в зависимости от состояния). В-третьих, для доступа к памяти ARM использует модель загрузки и сохранения. Это означает, что перед выполнением операций данные необходимо переместить из памяти в регистры, а обращаться к памяти могут только команды загрузки и сохранения. В ARM это команды LDR и STR. Чтобы увеличить 32-разрядное значение по определённому адресу памяти, нужно сначала загрузить значение по этому адресу в регистр, увеличить его и записать обратно. В отличие от x86, где большинство команд могут непосредственно работать с данными в памяти, для такой простой операции в ARM потребуются три команды (загрузка, увеличение и сохранение). Из этого можно заключить, что специалисту по реверс-инжинирингу придётся читать больше кода, но на практике после привыкания особой разницы нет.

ARM также предоставляет несколько уровней привилегий для реализации их изоляции. В x86 привилегии задаются четырьмя кольцами, причём кольцо 0 обладает наивысшими привилегиями, а кольцо 3 - наименьшими. В ARM привилегии определяются восемью различными режимами:

• пользовательский (User, USR);
• быстрый запрос прерывания (Fast interrupt request, FIQ);
• запрос прерывания (Interrupt request, IRQ);
• супервизор (Supervisor, SVC);
• монитор (Monitor, MON);
• аварийное завершение (Abort, ABT);
• неопределённый (Undefined, UND);
• системный (System, SYS).

Код, работающий в определённом режиме, имеет доступ к некоторым привилегиям и регистрам, недоступным в других режимах. Например, коду в режиме USR не разрешается изменять системные регистры (обычно они изменяются только в режиме SVC). USR - режим с наименьшими привилегиями. Несмотря на множество технических различий, для простоты можно считать USR аналогом кольца 3, а SVC - аналогом кольца 0. Большинство операционных систем реализует режим ядра в SVC, а пользовательский режим - в USR. Так поступают и Windows, и Linux.

Как вы, возможно, помните из главы 1, процессоры x64 могут выполнять 32-разрядный и 64-разрядный код либо попеременно работать с обоими. Процессоры ARM похожи на них тем, что также могут работать в двух состояниях: ARM и Thumb. Состояние ARM или Thumb определяет только набор команд, но не уровень привилегий. Например, код в режиме SVC может находиться как в состоянии ARM, так и в состоянии Thumb. В состоянии ARM команды всегда имеют ширину 32 бита; в состоянии Thumb ширина команд может составлять 16 или 32 бита. Состояние, в котором работает процессор, зависит от двух условий:

• Если при переходе по команде BX или BLX младший значащий бит регистра назначения равен 1, процессор переключается в состояние Thumb. (Хотя команды выровнены по границе 2 или 4 байт, процессор игнорирует младший значащий бит, поэтому проблем с выравниванием не возникает.)
• Если установлен бит T в текущем регистре состояния программы (current program status register, CPSR), процессор находится в режиме Thumb. Семантика CPSR объясняется в следующем разделе, а пока его можно считать расширенным аналогом регистра EFLAGS в x86.

При загрузке ядро ARM чаще всего входит в состояние ARM и остаётся в нём до явного или неявного переключения в Thumb. На практике во многих современных операционных системах используется преимущественно код Thumb, поскольку требуется более высокая плотность кода (смесь 16- и 32-разрядных команд может занимать меньше места, чем одни 32-разрядные команды); приложения могут работать в любом желаемом режиме.

Хотя у большинства команд Thumb и ARM одинаковые мнемоники, 32-разрядные команды Thumb имеют суффикс `.W`.

> **Примечание.** Распространено ошибочное мнение, будто Thumb подобен реальному режиму, а ARM - защищённому режиму x86/x64. Такое представление неверно. Большинство операционных систем на платформе x86/x64 работает в защищённом режиме и редко, если вообще когда-либо, возвращается в реальный. Операционные системы и приложения на платформе ARM могут попеременно выполнять код как в состоянии ARM, так и в состоянии Thumb. Кроме того, эти состояния совершенно не совпадают с режимами привилегий, рассмотренными в предыдущем абзаце (USR, SVC и т. д.).
>
> Существуют две версии Thumb: Thumb-1 и Thumb-2. Thumb-1 использовалась в ARMv6 и более ранних архитектурах, а её команды всегда имеют ширину 16 бит. Thumb-2 расширяет её, добавляя новые команды и позволяя им иметь ширину 16 или 32 бита. Для ARMv7 требуется Thumb-2, поэтому всякий раз, когда далее говорится о Thumb, подразумевается Thumb-2.
>
> Между состояниями ARM и Thumb есть ещё несколько различий, но рассмотреть их все здесь невозможно. Например, некоторые команды доступны в состоянии ARM, но отсутствуют в Thumb, и наоборот. Подробные сведения можно найти в официальной документации ARM.

Помимо различных состояний выполнения, ARM поддерживает условное выполнение. Это означает, что в команде кодируются определённые арифметические условия, которые должны выполняться, чтобы команда была исполнена. Например, команда может указывать, что будет выполнена только в том случае, если результат предыдущей команды равен нулю. Сравните это с x86, где почти каждая команда выполняется безусловно. (У Intel есть несколько команд, непосредственно поддерживающих условное выполнение: CMOV и SETNE.) Условное выполнение полезно, поскольку уменьшает количество команд перехода (которые обходятся очень дорого) и число выполняемых команд (что повышает плотность кода). В состоянии ARM условное выполнение поддерживают все команды, однако по умолчанию они выполняются безусловно. В состоянии Thumb для включения условного выполнения требуется специальная команда IT.

Ещё одна уникальная особенность ARM - барабанный сдвигатель. Некоторые команды могут «содержать» другую арифметическую команду, сдвигающую или циклически сдвигающую регистр. Это полезно, поскольку позволяет свернуть несколько команд в одну. Предположим, например, что нужно умножить регистр на 2, а затем сохранить результат в другом регистре. Обычно для этого потребовались бы две команды (умножение, а затем перемещение), но с помощью барабанного сдвигателя умножение (сдвиг влево на 1) можно включить в команду MOV. Такая команда выглядела бы примерно так:

```
MOV R1, R0, LSL #1            ; R1 = R0 * 2
```

Типы данных и регистры

Подобно языкам высокого уровня, ARM поддерживает операции над разными типами данных. Поддерживаются следующие типы: 8-разрядный (байт), 16-разрядный (полуслово), 32-разрядный (слово) и 64-разрядный (двойное слово).

Архитектура ARM определяет шестнадцать 32-разрядных регистров общего назначения, пронумерованных R0, R1, R2, ..., R15. Хотя программисту приложений доступны все они, на практике первые 12 регистров служат для общего применения (подобно EAX, EBX и т. д. в x86), а последние три имеют в архитектуре особое значение:

• R13 обозначается как указатель стека (stack pointer, SP). Это аналог ESP/RSP в x86/x64. Он указывает на вершину стека программы.
• R14 обозначается как регистр связи (link register, LR). Обычно во время вызова функции в нём хранится адрес возврата. Некоторые команды неявно используют этот регистр. Например, BL перед переходом к адресу назначения всегда сохраняет адрес возврата в LR. В x86/x64 аналогичного регистра нет, поскольку адрес возврата всегда сохраняется в стеке. В коде, где LR не используется для хранения адреса возврата, его можно применять как регистр общего назначения.
• R15 обозначается как счётчик команд (program counter, PC). При выполнении в состоянии ARM PC равен адресу текущей команды плюс 8 (на две команды ARM вперёд); в состоянии Thumb он равен адресу текущей команды плюс 4 (на две 16-разрядные команды Thumb вперёд). Он подобен EIP/RIP в x86/x64, за исключением того, что те всегда указывают на адрес следующей выполняемой команды. Другое важное различие состоит в том, что код может непосредственно читать регистр PC и записывать в него. Запись адреса в PC немедленно запускает выполнение по этому адресу. Чтобы избежать путаницы, рассмотрим это немного подробнее на следующем фрагменте в состоянии Thumb:

```
1: 0x00008344 push    {lr}
2: 0x00008346 mov     r0, pc
3: 0x00008348 mov.w   r2, r1, lsl #31
4: 0x0000834c pop     {pc}
```

После выполнения строки 2 R0 будет содержать значение `0x0000834a` (=`0x00008346+4`):

```
(gdb) br main

Breakpoint 1 at 0x8348
...
Breakpoint 1, 0x00008348 in main ()
(gdb) disas main

Dump of assembler code for function main:
   0x00008344 <+0>:     push    {lr}
   0x00008346 <+2>:     mov     r0, pc
=> 0x00008348 <+4>:     mov.w   r2, r1, lsl #31
   0x0000834c <+8>:     pop     {pc}
```

```
   0x0000834e <+10>:    lsls    r0, r0, #0
End of assembler dump.
(gdb) info register pc
pc             0x8348   0x8348 <main+4>
(gdb) info register r0
r0             0x834a   33610
```

Здесь точка останова установлена по адресу `0x00008348`. Когда она срабатывает, выводятся регистры PC и R0. Как видно, PC указывает на третью команду по адресу `0x00008348` (которая должна быть выполнена следующей), а R0 содержит ранее прочитанное значение PC. Из этого примера видно: при непосредственном чтении PC ведёт себя согласно определению, но во время отладки PC указывает на команду, которую предстоит выполнить.

Причина этой особенности - унаследованная конвейерная обработка в старых процессорах ARM, которые всегда выбирали две команды, следующие за выполняемой в данный момент. Современные конвейеры устроены гораздо сложнее, поэтому теперь это не имеет большого значения, однако ARM сохраняет такое определение ради совместимости с более ранними процессорами.

Как и другие архитектуры, ARM хранит сведения о текущем состоянии выполнения в текущем регистре состояния программы (current program status register, CPSR). С точки зрения программиста приложений CPSR подобен регистру EFLAGS/RFLAG в x86/x64. В некоторых документах может упоминаться регистр состояния прикладной программы (application program status register, APSR), который является псевдонимом некоторых полей CPSR. В CPSR имеется множество флагов; часть из них показана на рисунке 2-1 (остальные рассматриваются в последующих разделах).

• E (бит порядка байтов) - ARM может работать как с порядком байтов от старшего к младшему, так и от младшего к старшему. Для порядка от младшего к старшему этот бит равен 0, а для порядка от старшего к младшему - 1. Чаще всего используется порядок от младшего к старшему, поэтому бит равен 0.
• T (бит Thumb) - устанавливается в состоянии Thumb; в противном случае процессор находится в состоянии ARM. Один из способов явно перейти из Thumb в ARM (и наоборот) - изменить этот бит.
• M (биты режима) - задают текущий режим привилегий (USR, SVC и т. д.).

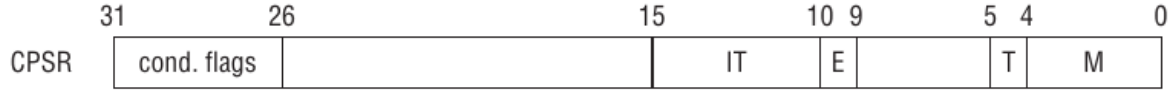

Рисунок 2-1. Некоторые поля регистра CPSR

Управление и параметры системного уровня

Для поддержки дополнительных команд и параметров системного уровня ARM предоставляет понятие сопроцессоров. Например, если система поддерживает модуль управления памятью (MMU), его параметры должны быть доступны загрузочному коду или коду ядра. В x86/x64 эти параметры хранятся в CR0 и CR4, а в ARM - в сопроцессоре 15. Архитектура ARM содержит 16 сопроцессоров, каждый из которых обозначается номером: CP0, CP1, ..., CP15. (В коде они обозначаются как P0, ..., P15.) Первые 13 являются либо необязательными, либо зарезервированными ARM; необязательные сопроцессоры производители могут использовать для реализации собственных команд и возможностей. Например, CP10 и CP11 обычно обеспечивают поддержку операций с плавающей точкой и NEON. Каждый сопроцессор содержит дополнительные «коды операций» и регистры, которыми можно управлять с помощью специальных команд ARM. CP14 и CP15 используются для отладки и системных параметров; в CP15, обычно называемом сопроцессором управления системой, хранится большинство системных параметров (кэширование, страничная организация памяти, исключения и т. д.).

> **Примечание.** NEON предоставляет набор команд SIMD (одна команда - множество данных), обычно используемый в мультимедийных приложениях. Он подобен командам SSE/MMX в архитектурах на основе x86.

У каждого сопроцессора есть 16 регистров и восемь соответствующих кодов операций. Семантика этих регистров и кодов операций зависит от конкретного сопроцессора. Обращаться к сопроцессорам можно только командами MRC (чтение) и MCR (запись); они принимают номер сопроцессора, номер регистра и коды операций. Например, чтобы прочитать регистр базы преобразования (подобный CR3 в x86/x64) и сохранить его в R0, используется следующая команда:

```
MRC p15, 0, r0, c2, c0, 0 ; сохранить TTBR в r0
```

Она означает: «прочитать регистр C2/C0 сопроцессора 15 с помощью кода операции 0/0 и сохранить результат в регистре общего назначения R0». Поскольку внутри каждого сопроцессора очень много регистров и кодов операций, для определения точного назначения каждого из них необходимо читать документацию. Некоторые регистры (C13/C0) зарезервированы для операционных систем и предназначены для хранения данных, относящихся к определённому процессу или потоку.

Хотя команды MRC и MCR не требуют высоких привилегий (то есть могут выполняться в режиме USR), некоторые регистры и коды операций сопроцессоров доступны только в режиме SVC. Попытка прочитать определённые регистры без достаточных привилегий приводит к исключению. На практике в коде пользовательского режима эти команды встречаются редко; обычно они присутствуют в очень низкоуровневом коде: ПЗУ, загрузчиках, микропрограммах или коде режима ядра.

Знакомство с набором команд

Теперь можно перейти к важным командам ARM. Помимо условного выполнения и барабанных сдвигателей, у команд есть ещё несколько особенностей, отсутствующих в x86. Во-первых, некоторые команды могут последовательно работать с диапазоном регистров. Например, чтобы сохранить пять регистров R6-R10 в определённой области памяти, на которую указывает R1, нужно записать `STM R1, {R6-R10}`. R6 будет сохранён по адресу памяти R1, R7 - по адресу R1+4, R8 - по адресу R1+8 и т. д. Несмежные регистры также можно перечислить через запятую (например, `{R1,R5,R8}`). В синтаксисе языка ассемблера ARM диапазоны регистров обычно указываются в фигурных скобках. Во-вторых, некоторые команды могут при необходимости обновлять базовый регистр после операции чтения или записи. Обычно для этого после имени регистра ставится восклицательный знак (`!`). Например, если переписать предыдущую команду в виде `STM R1!, {R6-R10}` и выполнить её, то R1 будет обновлён адресом, непосредственно следующим за областью, где был сохранён R10. Для ясности рассмотрим пример:

```
01: (gdb) disas main
02: Dump of assembler code for function main:
03: => 0x00008344 <+0>:     mov     r6,  #10
04:    0x00008348 <+4>:     mov     r7,  #11
05:    0x0000834c <+8>:     mov     r8,  #12
06:    0x00008350 <+12>:    mov     r9,  #13
07:    0x00008354 <+16>:    mov     r10, #14
08:    0x00008358 <+20>:    stmia   sp!, {r6, r7, r8, r9, r10}
09:    0x0000835c <+24>:    bx      lr
10: End of assembler dump.
11: (gdb) si
12: 0x00008348 in main ()
13: ...
14: 0x00008358 in main ()
15: (gdb) info reg sp
16: sp             0xbedf5848       0xbedf5848
17: (gdb) si
18: 0x0000835c in main ()
19: (gdb) info reg sp
20: sp             0xbedf585c       0xbedf585c
21: (gdb) x/6x 0xbedf5848
22: 0xbedf5848:     0x0000000a      0x0000000b      0x0000000c
0x0000000d
23: 0xbedf5858:     0x0000000e      0x00000000
```

В строке 15 показано значение SP (`0xbedf5848`) до выполнения команды STM; в строках 17 и 19 выполняется STM и выводится изменённое значение SP. Строка 21 выводит шесть слов, начиная со старого значения SP. Обратите внимание: R6 был сохранён по старому адресу SP, R7 - по адресу SP+0x4, R8 - по адресу SP+0x8, R9 - по адресу SP+0xc, а R10 - по адресу SP+0x10. Новый SP (`0xbedf585c`) указывает сразу за областью, где был сохранён R10.

> **Примечание.** STMIA и STMEA - псевдокоманды STM, то есть имеют одинаковое значение. Дизассемблер может выбрать для отображения любую из них. Некоторые дизассемблеры показывают STMEA, если базовым регистром является SP, и STMIA для других регистров; одни всегда используют STM, другие - STMIA. Строгого правила нет, поэтому при работе с несколькими дизассемблерами к этому необходимо привыкнуть.

Загрузка и сохранение данных

В предыдущем разделе упоминалось, что ARM является архитектурой загрузки и сохранения, а значит, перед выполнением операций данные необходимо загружать в регистры. Обращаться к памяти могут только команды загрузки и сохранения; все остальные команды способны работать лишь с регистрами. Загрузить означает прочитать данные из памяти и сохранить их в регистре; сохранить - записать содержимое регистра в ячейку памяти. В ARM командами загрузки и сохранения являются LDR/STR, LDM/STM и PUSH/POP.

LDR и STR

Эти команды могут загружать из памяти и сохранять в неё 1, 2 или 4 байта. Их полный синтаксис довольно сложен, поскольку существует несколько способов задать смещение и побочные эффекты обновления базового регистра. Рассмотрим простейший случай:

```
01: 03 68         LDR             R3, [R0] ; R3 = *R0
02: 23 60         STR             R3, [R4] ; *R4 = R3;
```

В команде из строки 1 R0 является базовым регистром, а R3 - приёмником; значение слова по адресу R0 загружается в R3. В строке 2 R4 является базовым регистром, а R3 - приёмником; значение из R3 сохраняется по адресу памяти R4. Этот пример прост, поскольку адрес памяти задаётся базовым регистром.

На фундаментальном уровне команды LDR/STR принимают базовый регистр и смещение; существует три формы смещения и три режима адресации для каждой формы. Начнём с форм смещения: непосредственное значение, регистр и масштабированный регистр.

В первой форме в качестве смещения используется непосредственное значение, то есть обычное целое число. Оно прибавляется к базовому регистру или вычитается из него, чтобы обратиться к данным по смещению, известному во время компиляции. Чаще всего такая форма применяется для доступа к определённому полю структуры или таблицы виртуальных функций. Общий формат выглядит так:

- `STR Ra, [Rb, imm]`
- `LDR Ra, [Rc, imm]`

Rb - базовый регистр, а `imm` - смещение, которое прибавляется к Rb.

Предположим, например, что R0 содержит указатель на структуру KDPC и имеется следующий код.

Определение структуры:

```
0:000> dt ntkrnlmp!_KDPC
   +0x000 Type             : UChar
   +0x001 Importance       : UChar
   +0x002 Number           : Uint2B
   +0x004 DpcListEntry     : _LIST_ENTRY
   +0x00c DeferredRoutine  : Ptr32     void
   +0x010 DeferredContext  : Ptr32 Void
   +0x014 SystemArgument1  : Ptr32 Void
   +0x018 SystemArgument2  : Ptr32 Void
   +0x01c DpcData          : Ptr32 Void
```

Код:

```
01: 13 23         MOVS    R3, #0x13
02: 03 70         STRB    R3, [R0]
03: 01 23         MOVS    R3, #1
04: 43 70         STRB    R3, [R0,#1]
05: 00 23         MOVS    R3, #0
06: 43 80         STRH    R3, [R0,#2]
07: C3 61         STR     R3, [R0,#0x1C]
08: C1 60         STR     R1, [R0,#0xC]
09: 02 61         STR     R2, [R0,#0x10]
```

В данном случае R0 - базовый регистр, а непосредственными значениями являются `0x1`, `0x2`, `0xC`, `0x10` и `0x1C`. Фрагмент можно перевести в C следующим образом:

```
KDPC *obj = ...;             /* R0 - это obj */
obj->Type = 0x13;
obj->Importance = 0x1;
obj->Number = 0x0;
obj->DpcData = NULL;
obj->DeferredRoutine = R1;   /* R1 нам неизвестен */
obj->DeferredContext = R2;   /* R2 нам неизвестен */
```

Эта форма смещения подобна `MOV Reg, [Reg + Imm]` в x86/x64.

Во второй форме смещения используется регистр. Она часто встречается в коде, которому требуется обращаться к массиву, но индекс вычисляется во время выполнения. Общий формат выглядит так:

- `STR Ra, [Rb, Rc]`
- `LDR Ra, [Rb, Rc]`

В зависимости от контекста базовым регистром или смещением может быть Rb либо Rc. Рассмотрим два примера.

Пример 1:

```
01: 03 F0 F2 FA   BL  strlen
02: 06 46         MOV R6, R0
; R0 - возвращённое значение strlen
03: ...
04: BB 57         LDRSB R3, [R7,R6]
; в данном случае R6 - смещение
```

Пример 2:

```
01: B3 EB 05 08   SUBS.W  R8, R3, R5
02: 2F 78         LDRB    R7, [R5]
03: 18 F8 05 30   LDRB.W  R3, [R8,R5]
; здесь R5 - база, а R8 - смещение
04: 9F 42         CMP     R7, R3
```

Это подобно форме `MOV Reg, [Reg + Reg]` в x86/x64.

В третьей форме смещения используется масштабированный регистр. Она часто встречается в циклах, перебирающих массив. Для масштабирования смещения применяется барабанный сдвигатель. Общий формат выглядит так:

- `LDR Ra, [Rb, Rc, <shifter>]`
- `STR Ra, [Rb, Rc, <shifter>]`

Rb является базовым регистром, Rc - непосредственным значением, а `<shifter>` - операцией над непосредственным значением, обычно сдвигом влево или вправо для его масштабирования. Например:

```
01: 0E 4B         LDR     R3, =KeNumberNodes
02: ...
03: 00 24         MOVS    R4, #0
04: 19 88         LDRH    R1, [R3]
05: 09 48         LDR     R0, =KeNodeBlock
06: 00 23         MOVS    R3, #0
07:             loop_start
08: 50 F8 23 20   LDR.W   R2, [R0,R3,LSL#2]
09: 00 23         MOVS    R3, #0
10: A2 F8 90 30   STRH.W  R3, [R2,#0x90]
11: 92 F8 89 30   LDRB.W  R3, [R2,#0x89]
12: 53 F0 02 03   ORRS.W  R3, R3, #2
13: 82 F8 89 30   STRB.W  R3, [R2,#0x89]
14: 63 1C         ADDS    R3, R4, #1
15: 9C B2         UXTH    R4, R3
16: 23 46         MOV     R3, R4
17: 8C 42         CMP     R4, R1
18: EF DB         BLT     loop_start
```

`KeNumberNodes` и `KeNodeBlock` являются соответственно глобальным целым числом и массивом указателей на KNODE.

Строки 1 и 5 просто загружают эти глобальные объекты в регистры (этот синтаксис объясняется далее). Строка 8 перебирает массив `KeNodeBlock` (база находится в R0), а R3 является индексом, умноженным на 2 (поскольку это массив указателей, а размер указателя на данной платформе составляет 4 байта). В строках 10-13 инициализируются некоторые поля элемента KNODE. Строка 14 увеличивает индекс. В строке 17 индекс сравнивается с размером массива (размер находится в R1; см. строку 4), и, если индекс меньше размера, цикл продолжается.

Этот фрагмент можно приблизительно перевести в C следующим образом:

```
int KeNumberNodes = …;
KNODE *KeNodeBlock[KeNumberNodes] = …;
for (int i=0; i < KeNumberNodes; i++) {
    KeNodeBlock[i].x = …;
    KeNodeBlock[i].y = …;
    …
}
```

Это подобно форме `MOV, Reg, [Reg + idx * scale]` в x86/x64.

Рассмотрев три формы смещения, в оставшейся части раздела обсудим режимы адресации: со смещением, с предварительной индексацией и с последующей индексацией. Они различаются лишь тем, изменяется ли базовый регистр и, если да, каким образом. Во всех предыдущих примерах смещения применялся режим адресации со смещением, при котором базовый регистр никогда не изменяется. Это самый простой и распространённый режим. Его легко узнать по отсутствию восклицательного знака (`!`) и по тому, что непосредственное значение находится внутри квадратных скобок. (В некоторых публикациях эти режимы называются предварительной индексацией, предварительной индексацией с обратной записью и последующей индексацией. Используемая здесь терминология соответствует официальной документации ARM.) Общий синтаксис режима со смещением: `LDR Rd, [Rn, offset]`.

Режим адресации с предварительной индексацией означает, что базовый регистр будет обновлён окончательным адресом памяти, использованным в операции обращения. По семантике он очень похож на префиксную форму унарного оператора ++ или `--` в C. Синтаксис этого режима: `LDR Rd, [Rn, offset]!`. Например:

```
12 F9 01 3D   LDRSB.W R3, [R2 ,#-1]! ; R3 = *(R2-1)
                                      ; R2 = R2-1
```

Режим адресации с последующей индексацией означает, что базовый регистр используется как окончательный адрес, после чего обновляется вычисленным смещением. Это очень похоже на постфиксную форму унарного оператора ++ или `--` в C. Синтаксис этого режима: `LDR Rd, [Rn], offset`. Например:

```
10 F9 01 6B   LDRSB.W R6, [R0],#1  ; R6 = *R0
                                   ; R0 = R0+1
```

Формы с предварительной и последующей индексацией обычно встречаются в коде, который многократно обращается к смещениям в одном буфере. Предположим, например, что коду требуется в цикле проверять, совпадает ли символ строки с одним из пяти символов; компилятор может обновлять базовый указатель, чтобы исключить отдельную команду увеличения.

> **Примечание.** Вот подсказка, помогающая распознавать и запоминать разные режимы адресации LDR/STR: если присутствует `!`, это предварительная индексация; если базовый регистр находится в квадратных скобках один, это последующая индексация; во всех остальных случаях используется режим со смещением.

Другие способы применения LDR

Как объяснялось ранее, LDR служит для загрузки данных из памяти в регистр, однако иногда она встречается в следующих формах:

```
01: DF F8 50 82   LDR.W   R8, =0x2932E00 ; LDR R8, [PC, x]
02: 80 4A         LDR     R2, =a04d ; "%04d" ; LDR R2, [PC, y]
03: 0E 4B         LDR     R3, =__imp_realloc ; LDR R3, [PC, z]
```

Очевидно, согласно предыдущему разделу такой синтаксис недопустим. С технической точки зрения это псевдокоманды, используемые дизассемблерами для упрощения ручного анализа. Внутри они используют форму LDR с непосредственным смещением и PC в качестве базового регистра; иногда это называется адресацией относительно PC (или относительно RIP в x64). В двоичных файлах ARM обычно имеется пул литералов - область памяти в разделе, где хранятся константы, строки и смещения, на которые другой код может ссылаться позиционно-независимым образом. (Пул литералов является частью кода, поэтому находится в том же разделе.) В предыдущем фрагменте код ссылается на 32-разрядную константу, строку и смещение импортированной функции, хранящиеся в пуле литералов. Эта псевдокоманда удобна тем, что позволяет одной командой поместить 32-разрядную константу в регистр. Для ясности рассмотрим следующий фрагмент:

```
01: .text:0100B134 35 4B   LDR    R3, =0x68DB8BAD
                ; в действительности LDR R3, [PC, #0xD4]
                ; в этот момент PC = 0x0100B138
02: ...
03: .text:0100B20C AD 8B DB 68 dword_100B20C DCD 0x68DB8BAD
```

Как дизассемблер сократил первую команду из `LDR R3, [PC, #0xD4]` до альтернативной формы? Поскольку код находится в состоянии Thumb, PC равен адресу текущей команды плюс 4, то есть `0x0100B138`; используется форма с непосредственным смещением относительно PC, а значит, выполняется попытка прочитать слово по адресу `0x0100B20C` (=`0x100B138+0xD4`), где как раз и находится требуемая константа.

Ещё одна связанная с этим команда - ADR, которая получает адрес метки или функции и помещает его в регистр. Например:

```
01: 00009390 65 A5         ADR     R5, dword_9528
02: 00009392 D5 E9 00 45   LDRD.W  R4, R5, [R5]
03: ...
04: 00009528 00 CE 22 A9+dword_9528 DCD 0xA922CE00 , 0xC0A4
```

Эта команда обычно применяется при реализации таблиц переходов и обратных вызовов, когда адрес функции необходимо передать другой функции. Внутри команда просто вычисляет смещение относительно PC и сохраняет его в регистре-приёмнике.

LDM и STM

LDM и STM подобны LDR/STR, но загружают и сохраняют несколько слов по заданному базовому адресу. Они удобны для перемещения нескольких блоков данных из памяти и в память. Общий синтаксис выглядит так:

• `LDM<mode> Rn[!], {Rm}`
• `STM<mode> Rn[!], {Rm}`

Rn является базовым регистром и содержит адрес памяти, из которой выполняется загрузка или в которую выполняется сохранение; необязательный восклицательный знак (`!`) означает, что базовый регистр следует обновить новым адресом (обратная запись). Rm - диапазон регистров для загрузки или сохранения. Существует четыре режима:

• IA (увеличение после, Increment After) - данные сохраняются начиная с ячейки памяти, заданной базовым адресом. При обратной записи записывается адрес на 4 байта выше последней ячейки. Если режим не указан, по умолчанию используется именно этот.
• IB (увеличение до, Increment Before) - данные сохраняются начиная с ячейки памяти на 4 байта выше базового адреса. При обратной записи записывается адрес последней ячейки.
• DA (уменьшение после, Decrement After) - данные сохраняются таким образом, что последней ячейкой является базовый адрес. При обратной записи записывается адрес на 4 байта ниже самой младшей ячейки.
• DB (уменьшение до, Decrement Before) - данные сохраняются таким образом, что последняя ячейка расположена на 4 байта ниже базового адреса. При обратной записи записывается адрес первой ячейки.

Сначала всё это может показаться немного запутанным, поэтому пошагово рассмотрим пример в отладчике:

```
01: (gdb) br main
02: Breakpoint 1 at 0x8344
03: (gdb) disas main
04: Dump of assembler code for function main:

05:    0x00008344 <+0>:     ldr     r6, =mem  ; слегка отредактировано
06:    0x00008348 <+4>:     mov     r0, #10
07:    0x0000834c <+8>:     mov     r1, #11
08:    0x00008350 <+12>:    mov     r2, #12
09:    0x00008354 <+16>:    ldm     r6, {r3, r4, r5}  ; режим IA
10:    0x00008358 <+20>:    stm     r6, {r0, r1, r2}  ; режим IA
11: ...
```

```
12: (gdb) r
13: Breakpoint 1, 0x00008344 in main ()
14: (gdb) si
15: 0x00008348 in main ()
16: (gdb) x/3x $r6
17: 0x1050c <mem>:  0x00000001      0x00000002      0x00000003
18: (gdb) si
19: 0x0000834c in main ()
20: ...
21: (gdb)
22: 0x00008358 in main ()
23: (gdb) info reg r3 r4 r5
24: r3             0x1      1
25: r4             0x2      2
26: r5             0x3      3
27: (gdb) si
28: 0x0000835c in main ()
29: (gdb) x/3x $r6
30: 0x1050c <mem>:  0x0000000a      0x0000000b      0x0000000c
```

Строка 5 сохраняет адрес памяти в R6; содержимое этой области памяти (`0x1050c`) представляет собой три слова (строка 17). В строках 6-8 регистрам R0-R2 присваиваются несколько констант. Строка 9 загружает три слова в R3-R5, начиная с ячейки памяти, заданной R6. Как показано в строках 24-26, R3-R5 содержат ожидаемые значения. Строка 10 сохраняет R0-R2, начиная с ячейки памяти, заданной R6. Строка 29 показывает, что были записаны ожидаемые значения. Результат предыдущих операций показан на рисунке 2-2.

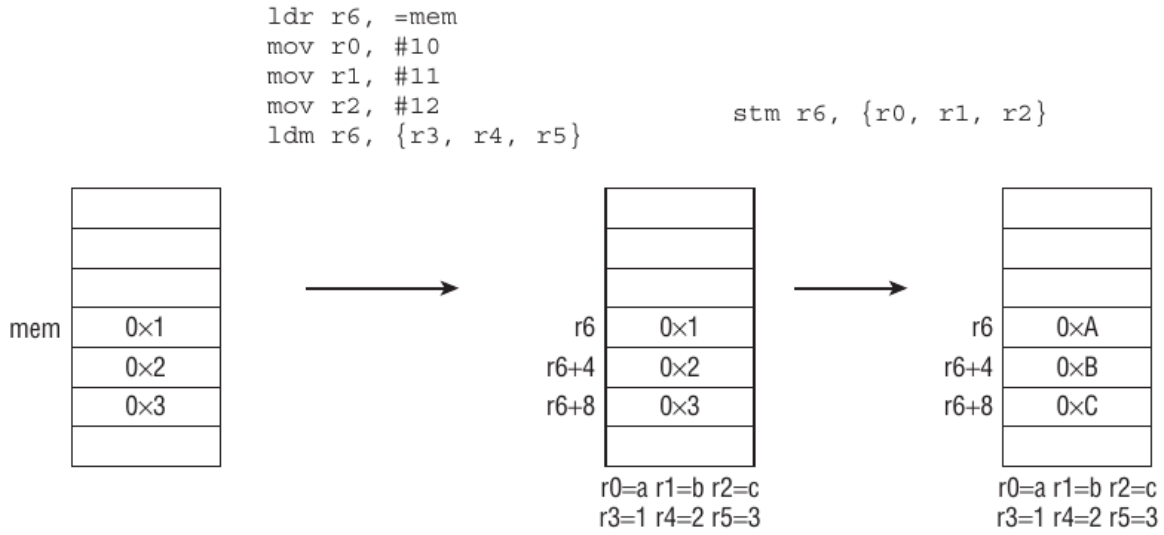

Рисунок 2-2. Загрузка и сохранение нескольких регистров без обратной записи

Ниже приведён тот же эксперимент с обратной записью:

```
01: (gdb) br main
02: Breakpoint 1 at 0x8344
03: (gdb) disas main
04: Dump of assembler code for function main:
05:    0x00008344 <+0>:     ldr     r6, =mem   ; слегка отредактировано
06:    0x00008348 <+4>:     mov     r0, #10
07:    0x0000834c <+8>:     mov     r1, #11
08:    0x00008350 <+12>:    mov     r2, #12
09:    0x00008354 <+16>:    ldm     r6!, {r3, r4, r5} ; режим IA с обратной записью
10:    0x00008358 <+20>:    stmia   r6!, {r0, r1, r2} ; режим IA с обратной записью
11: ...
12: (gdb) r
13: Breakpoint 1, 0x00008344 in main ()
14: (gdb) si
15: 0x00008348 in main ()
16: ...
```

```
17: (gdb)
18: 0x00008354 in main ()
19: (gdb) x/3x $r6
20: 0x1050c <mem>:  0x00000001      0x00000002      0x00000003
21: (gdb) si
22: 0x00008358 in main ()
23: (gdb) info reg r6
24: r6             0x10518  66840
25: (gdb) si
26: 0x0000835c in main ()
27: (gdb) info reg $r6
28: r6             0x10524  66852
29: (gdb) x/4x $r6-12
30: 0x10518 :       0x0000000a      0x0000000b      0x0000000c
0x00000000
```

В строке 9 используется режим IA с обратной записью, поэтому R6 обновляется адресом на 4 байта выше последней ячейки (строка 23). То же можно наблюдать в строках 10, 27 и 30. Результат предыдущего фрагмента показан на рисунке 2-3.

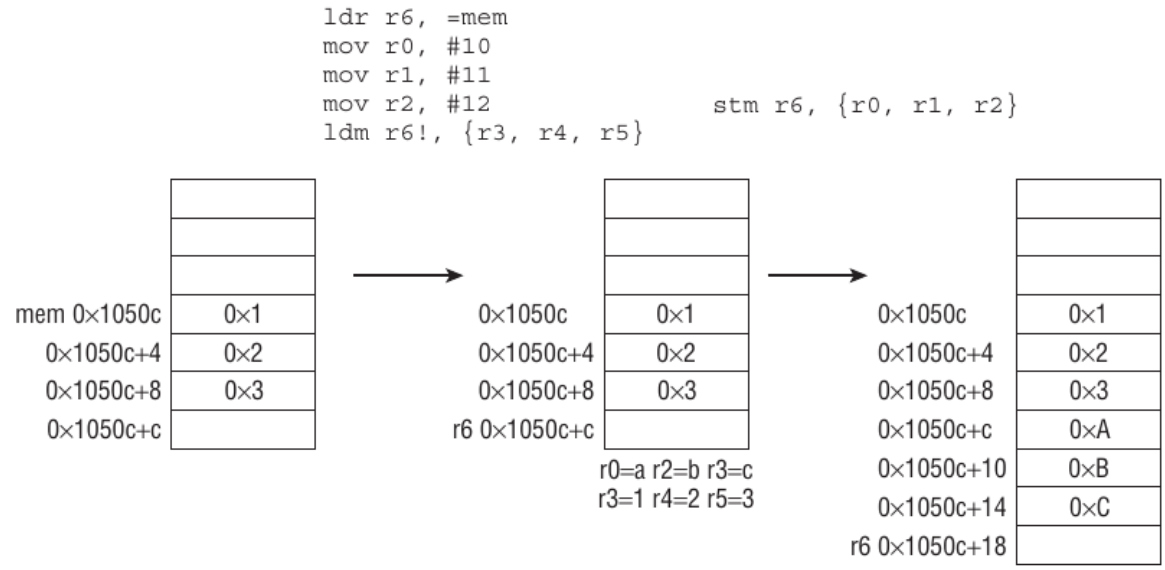

Рисунок 2-3. Загрузка и сохранение нескольких регистров с обратной записью

Поскольку LDM и STM могут перемещать сразу несколько слов, они обычно используются в операциях блочного копирования или перемещения. Например, иногда с их помощью встраивается `memcpy`, если длина копируемых данных известна во время компиляции. Они подобны команде MOVS с префиксом REP в x86. Рассмотрим следующие фрагменты кода, созданные двумя разными компиляторами из одного исходного файла.

Компилятор A:

```
01: A4 46         MOV      R12, R4
02: 35 46         MOV      R5, R6
03: BC E8 0F 00   LDMIA.W  R12!, {R0-R3}
04: 0F C5         STMIA    R5!, {R0-R3}
05: BC E8 0F 00   LDMIA.W  R12!, {R0-R3}
06: 0F C5         STMIA    R5!, {R0-R3}
07: 9C E8 0F 00   LDMIA.W  R12, {R0-R3}
08: 85 E8 0F 00   STMIA.W  R5, {R0-R3}
```

Компилятор B:

```
01: 30 22         MOVS   R2, #0x30
02: 21 46         MOV    R1, R4
03: 30 46         MOV    R0, R6
04: 23 F0 17 FA   BL     memcpy
```

Всё это лишь копирует 48 байт из одного буфера в другой: первый компилятор использует LDM/STM с обратной записью, загружая и сохраняя по 16 байт за раз, а второй просто вызывает свою реализацию `memcpy`. При реверс-инжиниринге встроенную форму `memcpy` можно обнаружить по тому, что LDM/STM используют одни и те же указатели источника и назначения с одинаковым набором регистров. Этот полезный приём стоит запомнить, поскольку встречается он часто.

Ещё одно распространённое место, где встречаются LDM/STM, - начало и конец функций в состоянии ARM. В таком контексте они используются как пролог и эпилог. Например:

```
01: F0 4F 2D E9   STMFD   SP!, {R4-R11,LR}  ; сохранить регистры и адрес возврата
02: ...
03: F0 8F BD E8   LDMFD   SP!, {R4-R11,PC}  ; восстановить регистры и вернуться
```

STMFD и LDMFD являются псевдокомандами соответственно STMDB и LMDIA/LDM.

> **Примечание.** После STM/LDM часто встречаются суффиксы FD, FA, ED или EA. Это всего лишь псевдокоманды для LDM/STM в разных режимах (IA, IB и т. д.). Соответствия таковы: STMFD/STMDB, STMFA/STMIB, STMED/STMDA, STMEA/STMIA, LDMFD/LDMIA, LDMFA/LDMDA и LDMEA/LDMDB. Запомнить их бывает непросто; наиболее действенный способ - нарисовать схему для каждой команды.

PUSH и POP

Последний набор команд загрузки и сохранения - PUSH и POP. Они подобны LDM/STM, за исключением двух особенностей:

• в качестве базового адреса неявно используется SP;
• SP обновляется автоматически.

Как и в архитектуре x86/x64, стек растёт вниз, в сторону меньших адресов. Общий синтаксис: `PUSH/POP {Rn}`, где Rn может быть диапазоном регистров.

PUSH сохраняет регистры в стеке таким образом, что последняя ячейка находится на 4 байта ниже текущего указателя стека, и обновляет SP адресом первой ячейки. POP загружает регистры начиная с текущего указателя стека и обновляет SP адресом на 4 байта выше последней ячейки. Фактически PUSH/POP эквивалентны STMDB/LDMIA с обратной записью и SP в качестве базового указателя. Ниже приведён краткий пошаговый пример работы команд:

```
01: (gdb) disas main
02: Dump of assembler code for function main:
03:    0x00008344 <+0>:     mov.w   r0, #10
04:    0x00008348 <+4>:     mov.w   r1, #11
05:    0x0000834c <+8>:     mov.w   r2, #12
06:    0x00008350 <+12>:    push    {r0, r1, r2}
07:    0x00008352 <+14>:    pop     {r3, r4, r5}
08: ...
09: (gdb) br main
10: Breakpoint 1 at 0x8344
11: (gdb) r
12: Breakpoint 1, 0x00008344 in main ()
13: (gdb) si
14: 0x00008348 in main ()
15: ...
16: (gdb)
```

```
17: 0x00008350 in main ()
18: (gdb) info reg sp        ; текущий указатель стека
19: sp             0xbee56848       0xbee56848
20: (gdb) si
21: 0x00008352 in main ()
22: (gdb) x/3x $sp           ; после push регистр sp обновлён
23: 0xbee5683c:     0x0000000a      0x0000000b      0x0000000c
24: (gdb) si                 ; извлечь значения в регистры
25: 0x00008354 in main ()
26: (gdb) info reg r3 r4 r5  ; новые значения регистров
27: r3             0xa      10
28: r4             0xb      11
29: r5             0xc      12
30: (gdb) info reg sp        ; новый sp (на 4 байта выше последней ячейки)
31: sp             0xbee56848       0xbee56848
32: (gdb) x/3x $sp-12
33: 0xbee5683c:     0x0000000a      0x0000000b      0x0000000c
```

Предыдущий фрагмент показан на рисунке 2-4.

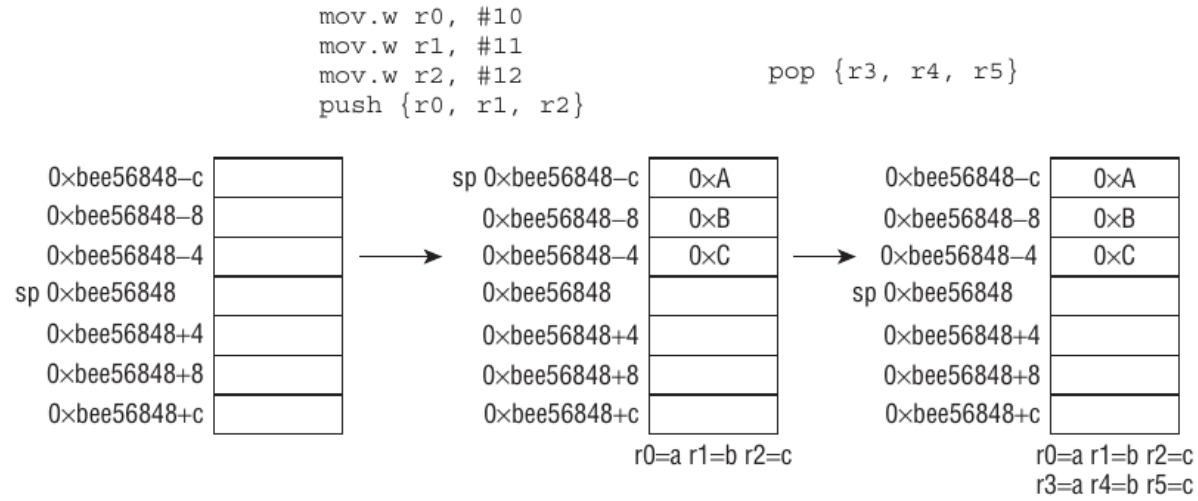

Рисунок 2-4. Работа команд PUSH и POP

Чаще всего PUSH/POP встречаются в начале и конце функций. В таком контексте они служат прологом и эпилогом (подобно STMFD/LDMFD в состоянии ARM). Например:

```
01: 2D E9 F0 4F   PUSH.W  {R4-R11,LR} ; сохранить регистры и адрес возврата
02: ...
03: BD E8 F0 8F   POP.W   {R4-R11,PC} ; восстановить регистры и вернуться
```

Некоторые дизассемблеры действительно используют эту конструкцию как эвристику для определения границ функций.

Функции и вызов функций

В отличие от x86/x64, где для вызова функций (CALL) и переходов (JMP) существует лишь по одной команде, ARM предлагает несколько вариантов в зависимости от кодирования адреса назначения. При вызове функции процессору необходимо знать, откуда возобновить выполнение после возврата из функции; это место обычно называется адресом возврата. В x86 команда CALL перед переходом к целевой функции неявно помещает адрес возврата в стек; завершив выполнение, целевая функция возобновляет его по адресу возврата, извлекая этот адрес из стека в EIP.

Механизм ARM по существу тот же, но имеет несколько небольших отличий. Во-первых, адрес возврата можно хранить в стеке или регистре связи (LR); чтобы возобновить выполнение после

вызова, адрес возврата явно извлекается из стека в PC либо выполняется безусловный переход к LR. Во-вторых, переход может переключать состояние ARM и Thumb в зависимости от младшего значащего бита адреса назначения. В-третьих, ARM определяет стандартное соглашение о вызовах: первые четыре 32-разрядных параметра передаются через регистры R0-R3, а остальные - в стеке. Возвращаемое значение хранится в R0.

Для вызовов функций используются команды B, BX, BL и BLX.

Хотя B редко встречается в контексте вызова функции, её можно использовать для передачи управления. Это обычный безусловный переход, идентичный команде JMP в x86. Обычно B применяется внутри циклов и условных конструкций для возврата к началу или выхода, а также может вызывать функцию, которая никогда не возвращается. В качестве адреса назначения B может использовать только смещения меток, но не регистры. В данном контексте синтаксис B выглядит так: `B imm`, где `imm` - смещение относительно текущей команды. (Здесь не учитываются флаги условного выполнения, которые рассматриваются в разделе «Переходы и условное выполнение».) Важно помнить: поскольку команды ARM и Thumb выровнены соответственно по границе 4 и 2 байт, целевое смещение должно быть чётным. Ниже показан фрагмент с применением B:

```
01: 0001C788   B         loc_1C7A8
02: 0001C78A
03: 0001C78A loc_1C78A
04: 0001C78A   LDRB      R7, [R6,R2]
05: ...
06: 0001C7A4   STRB.W    R7, [R3,#-1]
07: 0001C7A8
08: 0001C7A8 loc_1C7A8
09: 0001C7A8   MOV       R7, R3
10: 0001C7AA   ADDS      R3, #2
11: 0001C7AC   CMP       R2, R4
12: 0001C7AE   BLT       loc_1C78A
```

В строке 1 B используется как безусловный переход, запускающий цикл. Остальные команды пока можно не рассматривать.

BX означает Branch and Exchange - переход и переключение. Подобно B, эта команда передаёт управление по целевому адресу, но может переключаться между состояниями ARM и Thumb, а целевой адрес хранится в регистре. Буква X в конце команды перехода означает способность переключаться между состояниями. Если младший значащий бит целевого адреса равен 1, процессор автоматически переключается в состояние Thumb; в противном случае выполнение происходит в состоянии ARM. Формат команды: `BX <register>`, где `register` содержит адрес назначения. Два самых распространённых применения этой команды - возврат из функции путём перехода к LR (то есть `BX LR`) и передача управления коду в другом режиме (то есть переход из ARM в Thumb или наоборот). В скомпилированном коде в конце функций почти всегда встречается `BX LR`; по сути, это аналог RET в x86.

BL означает Branch with Link - переход со связью. Она подобна B, но перед передачей управления по целевому смещению также сохраняет адрес возврата в LR. Вероятно, это ближайший аналог команды CALL в x86, и она часто используется для вызова функций. Формат команды такой же, как у B (то есть она принимает только смещения). Ниже приведён короткий фрагмент, демонстрирующий вызов функции и возврат из неё:

```
01: 00014350   BL              foo ; LR = 0x00014354
02: 00014354   MOVS            R4, #0x15
03: ...
04: 0001B224 foo
05: 0001B224   PUSH            {R1-R3}
06: 0001B226   MOV             R3, 0x61240
07: ...
08: 0001B24C   BX              LR  ; вернуться к 0x00014354
```

В строке 1 функция `foo` вызывается командой BL; перед передачей управления по адресу назначения BL сохраняет адрес возврата (`0x000014354`) в LR. `foo` выполняет некоторую работу и возвращает управление вызывающей функции (`BX LR`).

BLX означает Branch with Link and Exchange - переход со связью и переключением. Она подобна BL, но дополнительно может переключать состояние. Главное различие состоит в том, что адресом назначения перехода BLX может быть как регистр, так и смещение; если BLX использует смещение, процессор всегда меняет состояние (ARM на Thumb и наоборот). Поскольку остальные её свойства совпадают со свойствами BL, её также можно считать эквивалентом команды CALL в x86. На практике для вызова функций применяются и BL, и BLX. BL обычно используется, если функция находится в пределах 32 Мбайт, а BLX - когда дальность цели заранее неизвестна (например, при вызове через указатель на функцию). При работе в состоянии Thumb BLX обычно используется для вызова библиотечных процедур; в состоянии ARM вместо неё применяется BL.

Рассмотрев все команды, связанные с безусловными переходами и непосредственными вызовами функций, а также возврат из функции (`BX LR`), можно закрепить знания на примере полной процедуры:

```
01: 0100C388             ; void *__cdecl mystery(int)
02: 0100C388             mystery
03: 0100C388 2D E9 30 48   PUSH.W  {R4,R5,R11,LR}
04: 0100C38C 0D F2 08 0B   ADDW    R11, SP, #8
05: 0100C390 0C 4B         LDR     R3, =__imp_malloc
06: 0100C392 C5 1D         ADDS    R5, R0, #7
07: 0100C394 6F F3 02 05   BFC.W   R5, #0, #3
08: 0100C398 1B 68         LDR     R3, [R3]
09: 0100C39A 15 F1 08 00   ADDS.W  R0, R5, #8
10: 0100C39E 98 47         BLX     R3
11: 0100C3A0 04 46         MOV     R4, R0
12: 0100C3A2 24 B1         CBZ     R4, loc_100C3AE
13: 0100C3A4 EB 17         ASRS    R3, R5, #0x1F
14: 0100C3A6 63 60         STR     R3, [R4,#4]
15: 0100C3A8 25 60         STR     R5, [R4]
16: 0100C3AA 08 34         ADDS    R4, #8
17: 0100C3AC 04 E0         B       loc_100C3B8
18: 0100C3AE             loc_100C3AE
19: 0100C3AE 04 49         LDR     R1, =aFailed ; "failed..."
20: 0100C3B0 2A 46         MOV     R2, R5
21: 0100C3B2 07 20         MOVS    R0, #7

22: 0100C3B4 01 F0 14 FC   BL      foo
23: 0100C3B8
24: 0100C3B8             loc_100C3B8
25: 0100C3B8 20 46         MOV     R0, R4
26: 0100C3BA BD E8 30 88   POP.W   {R4,R5,R11,PC}
27: 0100C3BA             ; конец функции mystery
```

Эта функция охватывает несколько рассмотренных ранее идей (остальные команды пока можно не учитывать):

• строка 3 является прологом, использующим последовательность `PUSH {..., LR}`, а строка 26 - эпилогом;
• строка 10 вызывает `malloc` посредством BLX;
• строка 22 вызывает `foo` посредством BL;
• строка 26 выполняет возврат с помощью последовательности `POP {..., PC}`.

Арифметические операции

Загрузив значение из памяти в регистр, код может перемещать его и выполнять над ним операции. Простейшая операция - перемещение в другой регистр командой MOV. Источником может быть константа, регистр или значение, обработанное барабанным сдвигателем. Вот примеры её применения:

```
01: 4F F0 0A 00   MOV.W   R0, #0xA  ; r0 = 0xa
02: 38 46         MOV     R0, R7    ; r0 = r7
03: A4 4A A0 E1   MOV     R4, R4, LSR #21 ; r4 = (r4>>21)
```

В строке 3 операнд-источник перед перемещением в приёмник обрабатывается барабанным сдвигателем. Операции барабанного сдвигателя включают сдвиг влево (LSL), сдвиг вправо (LSR, ASR) и циклический сдвиг (ROR, RRX). Барабанный сдвигатель полезен тем, что позволяет команде работать с константами, которые обычно нельзя закодировать в форме непосредственного значения. Команды ARM и Thumb имеют ширину 16 или 32 бита, поэтому не могут непосредственно принимать 32-разрядные константы как параметр; с помощью барабанного сдвигателя непосредственное значение можно преобразовать в большее и переместить в другой регистр. Другой способ поместить 32-разрядную константу в регистр - разделить её на две 16-разрядные половины и перемещать их по одной; обычно для этого используются команды MOVW и MOVT. MOVT задаёт старшие 16 бит регистра, а MOVW - младшие 16 бит.

Основные арифметические и логические операции - ADD, SUB, MUL, AND, ORR и EOR. Примеры их применения:

```
01: 4B 44         ADD      R3, R9           ; r3 = r3+r9
02: 0D F2 08 0B   ADDW     R11, SP, #8      ; r11 = sp+8
03: 04 EB 80 00   ADD.W    R0, R4, R0,LSL#2 ; r0 = r4 + (r0<<2)
04: EA B0         SUB      SP, SP, #0x1A8   ; sp = sp-0x1a8

05: 03 FB 05 F2   MUL.W    R2, R3, R5       ; r2 = r3*r5 (32-разрядный результат)
06: 14 F0 07 02   ANDS.W   R2, R4, #7       ; r2 = r4 & 7 (флаг)
07: 83 EA C1 03   EOR.W    R3, R3, R1,LSL#3 ; r3 = r3 ^ (r1<<3)
08: 53 40         EORS     R3, R2           ; r3 = r3 ^ r2 (флаг)
09: 43 EA 02 23   ORR.W    R3, R3, R2,LSL#8 ; r3 = r3 | (r2<<8)
10: 53 F0 02 03   ORRS.W   R3, R3, #2       ; r3 = r3 | 2 (флаг)
11: 13 43         ORRS     R3, R2           ; r3 = r3 | r2 (флаг)
```

Обратите внимание на `S` после некоторых команд. В отличие от x86, арифметические команды ARM по умолчанию не устанавливают флаги условий. Суффикс `S` означает, что команда должна установить арифметические флаги условий (ноль, отрицательное значение и т. д.) в соответствии с результатом. Обратите внимание: команда MUL усекает результат, поэтому в регистре-приёмнике сохраняются только младшие 32 бита; для полного 64-разрядного умножения используются команды SMULL и UMULL (подробности см. в техническом справочном руководстве ARM).

А где команда деления? В ARM нет встроенной команды деления. (В ядрах ARMv7-R и ARMv7-M имеются SDIV и UDIV, однако здесь они не рассматриваются.) На практике среда выполнения содержит программную реализацию деления, и при необходимости код просто вызывает её. Ниже показан пример со средой выполнения C в Windows:

```
01: 41 46         MOV             R1, R8
02: 30 46         MOV             R0, R6
03: 35 F0 9E FF   BL              __rt_udiv ; программная реализация udiv
```

Переходы и условное выполнение

Все рассмотренные до сих пор примеры выполнялись линейно. В большинстве программ присутствуют условия и циклы. На уровне языка ассемблера эти конструкции реализуются с помощью флагов условий, хранящихся в регистре состояния прикладной программы (APSR). APSR является псевдонимом CPSR и подобен регистру EFLAG в x86. На рисунке 2-5 показаны соответствующие флаги, описанные ниже:

• N (флаг отрицательного значения) - устанавливается, когда результат операции отрицателен (старший значащий бит результата равен 1).
• Z (флаг нуля) - устанавливается, когда результат операции равен нулю.
• C (флаг переноса) - устанавливается, когда результат операции над двумя беззнаковыми значениями вызывает переполнение.
• V (флаг переполнения) - устанавливается, когда результат операции над двумя знаковыми значениями вызывает переполнение.
• IT (биты If-then) - кодируют различные условия для команды Thumb IT. Они рассматриваются далее.

Биты N, Z, C и V идентичны битам SF, ZF, CF и OF регистра EFLAG в x86. Они используются для реализации условий и циклов языков высокого уровня, а также условного выполнения на уровне отдельных команд. Равенство описывается через эти флаги. В таблице 2-1 приведены распространённые отношения и соответствующие им флаги.

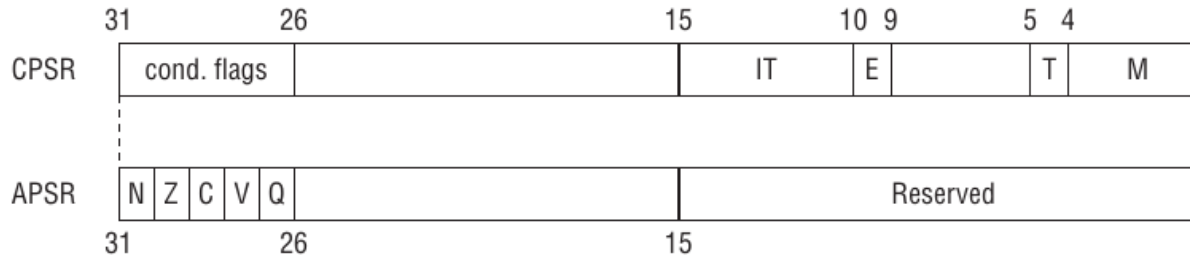

Рисунок 2-5. Поля регистров CPSR и APSR

Таблица 2-1. Коды условий и их значение

Суффикс/код	Значение	Флаги
EQ	Равно	`Z==1`
NE	Не равно	`Z==0`
MI	Минус, отрицательное значение	`N==1`
PL	Плюс, положительное значение или ноль	`N==0`
HI	Беззнаковое больше/выше	`C==1 and Z==0`

Суффикс/код	Значение	Флаги
LS	Беззнаковое меньше/ниже	`C==0 or Z==1`
GE	Знаковое больше или равно	`N==V`
LT	Знаковое меньше	`N!=V`
GT	Знаковое больше	`Z==0 and N==V`
LE	Знаковое меньше или равно	`Z==1 or N!=V`

Команды можно выполнять условно, добавляя в конец один из этих суффиксов. Например, BLT означает переход, если условие LT истинно. (Это аналог JL в x86.) По умолчанию команды не обновляют флаги условий, если не указан суффикс `S`; команды сравнения (CBZ, CMP, TST, CMN и TEQ) обновляют флаги автоматически, поскольку обычно используются перед командами перехода.

Вероятно, самая распространённая команда сравнения - CMP. Её синтаксис: `CMP Rn, X`, где Rn является регистром, а X может быть непосредственным значением, регистром или операцией барабанного сдвигателя. Семантика совпадает с x86: выполняется `Rn - X`, устанавливаются соответствующие флаги, а результат отбрасывается. Обычно после неё следует условный переход. Ниже приведён пример применения и псевдокод.

ARM:

```
01: B3 EB E7 7F   CMP.W     R3, R7, ASR #31
02: 05 DB         BLT       loc_less
03: 01 DC         BGT       loc_greater
04: BD 42         CMP       R5, R7
05: 02 D9         BLS       loc_less

06:             loc_greater
07: 07 3D         SUBS      R5, #7
08: 6E F1 00 0E   SBC.W     LR, LR, #0
09:             loc_less
10: A5 FB 08 12   UMULL.W   R1, R2, R5, R8
11: 87 FB 08 04   SMULL.W   R0, R4, R7, R8
12: 0E FB 08 23   MLA.W     R3, LR, R8, R2
```

Псевдокод на C:

```
if (r3 < r7) { goto loc_less; }
else if ( r3 > r7) { goto loc_greater; }
else if ( r5 < r7) { goto loc_less; }
```

Следующая по распространённости команда сравнения - TST; её синтаксис совпадает с CMP. Семантика идентична TEST в x86: выполняется `Rn & X`, устанавливаются соответствующие флаги, а результат отбрасывается. Обычно она служит для проверки значения на равенство другому значению или для проверки флагов. Как и после большинства команд сравнения, за ней обычно следует условный переход. Например:

```
01: AB 8A         LDRH     R3, [R5,#0x14]
02: 13 F0 02 0F   TST.W    R3, #2
03: 09 D0         BEQ      loc_10179DA
04: ...
05:             loc_10179BE
06: AA 8A         LDRH     R2, [R5,#0x14]
07: 12 F0 04 0F   TST.W    R2, #4
08: 02 D0         BEQ      loc_10179E8
```

В состоянии Thumb-2 есть две популярные команды сравнения: CBZ и CBNZ. Их синтаксис прост: `CBZ/CBNZ Rn, label`, где Rn - регистр, а `label` - смещение, к которому следует перейти при истинном условии. CBZ переходит к `label`, если регистр равен нулю. CBNZ действует так же, но проверяет условие неравенства нулю. Обычно эти команды используются, чтобы определить, равно ли число нулю или указатель NULL. Типичный пример:

ARM:

```
01: 10 F0 48 FF   BL          foo
                ; foo возвращает указатель в r0
02: 28 B1         CBZ         R0, loc_100BC8E
03: ...
04:             loc_100BC8E
05: 01 20         MOVS        R0, #1
06: 28 E0         B           locret_100BCE4
07: ...
08:             locret_100BCE4
09: BD E8 F8 89   POP.W       {R3-R8,R11,PC}
```

Псевдокод на C:

```
type *a;
a = foo(...);
if (a == NULL) { return 1; }
```

Другие команды сравнения - CMN и TEQ; они выполняют над операндами соответственно сложение и исключающее ИЛИ. Поскольку применяются они редко, здесь они не рассматриваются.

Как было показано, команду перехода B можно превратить в условный переход, добавив суффикс (BEQ, BLE, BLT, BLS и т. д.). В действительности таким же способом можно условно выполнять большинство команд ARM. Если условие не выполняется, команду можно считать не выполняющей никаких действий. Условное выполнение на уровне команд способно сократить количество переходов и тем самым ускорить выполнение. Например:

ARM:

```
01: 00 00 50 E3   CMP     R0, #0
02: 01 00 A0 03   MOVEQ   R0, #1
03: 68 00 D0 15   LDRNEB  R0, [R0,#0x68]
04: 1E FF 2F E1   BX      LR
```

Псевдокод на C:

```
unk_type *a = ...;
if (a == NULL) { return 1; }
else { return a->off_48; }
```

Благодаря команде LDR в строке 3 сразу понятно, что R0 является указателем. В строке 1 проверяется, равен ли R0 значению NULL. Если да (EQ), строка 2 устанавливает R0 в 1; в противном случае NEQ загружает в R0 значение по адресу R0+0x68 (строка 3), после чего выполняется возврат. Поскольку EQ и NEQ не могут быть истинны одновременно, будет выполнена только одна из этих команд. Обратите внимание: команд перехода здесь нет.

Состояние Thumb

В отличие от большинства команд ARM, команды Thumb (за исключением B) нельзя выполнять условно без команды IT (if-then). Эта команда, характерная для Thumb-2, позволяет условно выполнить до четырёх следующих за ней команд. Общий синтаксис: `ITxyz cc`, где `cc` - код условия для первой команды, а `x`, `y` и `z` описывают условия соответственно второй, третьей и четвёртой команд. Условия для команд после первой обозначаются одной из двух букв: T или E. T означает, что для выполнения условие должно совпадать с `cc`; E означает, что команда выполняется только при условии, обратном `cc`. Рассмотрим пример.

ARM:

```
01: 00 2B         CMP     R3, #0
          ; проверить и установить условие
02: 12 BF         ITEE NE
          ; начать блок IT
03: BC FA 8C F0   CLZNE.W R0, R12
          ; первая команда

04: B6 FA 86 F0   CLZEQ.W R0, R6
           ; вторая команда
05: 20 30         ADDEQ   R0, #0x20
           ; третья команда
```

Псевдокод на C:

```
if (R3 != 0) {
 R0 = countleadzeros(R12);
} else {

 R0 = countleadzeros(R6);
 R0 += 0x20
}
```

Строка 1 выполняет сравнение и устанавливает флаг условия. Строка 2 задаёт условия и начинает блок if-then. NE является условием выполнения первой команды; первая E (после IT) показывает, что условие выполнения второй команды обратно условию первой. (EQ обратно NE.) Вторая E задаёт то же самое для третьей команды. Строки 3-5 находятся внутри блока IT.

Благодаря своей гибкости команда IT позволяет уменьшить количество команд, необходимых для реализации коротких условных конструкций в состоянии Thumb.

Switch-Case

Операторы `switch-case` можно представить как множество объединённых операторов `if-else`. Поскольку проверяемое выражение и целевая метка известны во время компиляции, компиляторы обычно создают таблицу переходов для хранения адресов (ARM) или смещений (Thumb) каждого обработчика варианта. Определив индекс таблицы переходов, компилятор косвенно переходит к адресу назначения, загружая его в PC. В состоянии ARM это обычно выполняется командой LDR, где PC является регистром-приёмником и базовым регистром. Рассмотрим пример:

```
01:  ; R1 - вариант
02: 0B 00 51 E3   CMP      R1, #0xB   ; находится ли он в диапазоне?
03: 01 F1 9F 97   LDRLS    PC, [PC,R1,LSL#2] ; да, выбрать вариант,
                                     ; проиндексировав таблицу
04: 14 00 00 EA   B        loc_DD10  ; нет, выйти
05: 3C DD 00 00+  DCD loc_DD3C  ; начало таблицы переходов
06: 4C DD 00 00+  DCD loc_DD4C
07: 68 DD 00 00+  DCD loc_DD68
08: 8C DD 00 00+  DCD loc_DD8C
09: BC DD 00 00+  DCD loc_DDBC
10: F0 DD 00 00+  DCD loc_DDF0
```

```
11: 38 DE 00 00+  DCD loc_DE38
12: 38 DE 00 00+  DCD loc_DE38
13: EC DC 00 00+  DCD loc_DCEC  ; вариант/индекс 8
14: EC DC 00 00+  DCD loc_DCEC  ; вариант/индекс 9
15: 3C DD 00 00+  DCD loc_DD3C

16: 3C DD 00 00   DCD loc_DD3C
17:             loc_DCEC   ; обработчик вариантов 8, 9
18: 00 00 A0 E3   MOV      R0, #0
19: 08 10 41 E2   SUB      R1, R1, #8
20: 04 30 A0 E3   MOV      R3, #4
21: 14 00 82 E5   STR      R0, [R2,#0x14]
22: BC 31 C2 E1   STRH     R3, [R2,#0x1C]
23: 10 10 82 E5   STR      R1, [R2,#0x10]
```

В строке 2 проверяется, находится ли вариант в диапазоне; если нет, выполняется обработчик по умолчанию (строка 4). Строка 3 выполняется условно, если R1 находится в диапазоне; она переходит к обработчику варианта, индексируя таблицу переходов, и загружает адрес назначения в PC. Напомним, что в состоянии ARM PC находится на 8 байт дальше текущей команды, поэтому таблица переходов обычно хранится в 8 байтах от команды LDR.

В режиме Thumb применяется та же идея, но таблица переходов содержит смещения, а не адреса. Для поддержки переходов по таблицам со смещениями размером в байт или полуслово в ARM были добавлены команды TBB и TBH. Для TBB записи таблицы являются байтовыми значениями, а для TBH - полусловами. Чтобы получить окончательный адрес назначения перехода, запись таблицы нужно умножить на два и прибавить к PC. Ниже предыдущий пример представлен с использованием TBB:

```
01: 0101E600 0B 29         CMP     R1, #0xB ; находится ли он в диапазоне?
02: 0101E602 76 D8         BHI     loc_101E6F2 ; нет, выйти
03: 0101E604 04 26         MOVS    R6, #4
04: 0101E606 DF E8 01 F0   TBB.W   [PC,R1] ; перейти по смещению из таблицы
05: 0101E60A 06          jpt_101E606 DCB 6 ; начало таблицы переходов
06: 0101E60B 09            DCB 9
07: 0101E60C 0F            DCB 0xF
08: 0101E60D 18            DCB 0x18
09: 0101E60E 24            DCB 0x24
10: 0101E60F 32            DCB 0x32
11: 0101E610 45            DCB 0x45
12: 0101E611 45            DCB 0x45
13: 0101E612 6D            DCB 0x6D ; смещение для 8
14: 0101E613 6D            DCB 0x6D ; смещение для 9
15: 0101E614 06            DCB 6
16: 0101E615 06            DCB 6
17: ...
18: 0101E6E4             loc_101E6E4   ; обработчик вариантов 8, 9
19: 0101E6E4 B1 F1 08 03   SUBS.W  R3, R1, #8
20: 0101E6E8 00 20         MOVS    R0, #0
21: 0101E6EA 60 61         STR     R0, [R4,#0x14]
```

Поскольку код находится в состоянии Thumb, PC расположен на 4 байта дальше текущей команды; следовательно, для варианта 8 запись таблицы находится по адресу `0x0101E612` (`=0x0101E60A+8`) и равна `0x6d`, а обработчик расположен по адресу `0x101E6E4` (`=PC+(0x6d*2)`). Как и в предыдущем примере, таблица переходов обычно размещается после команды TBB/TBH. Обратите внимание: TBB/TBH используются только в состоянии Thumb.

Разное

В этом разделе кратко рассматриваются понятия, которые непосредственно не связаны с процессом реверс-инжиниринга. Однако на практике знать их важно, поскольку они расширяют общий кругозор. Чем больше знаний, тем лучше. При первом чтении этот раздел можно пропустить.

JIT-компиляция и самомодифицирующийся код

ARM поддерживает JIT-компиляцию (just-in-time, JIT) и самомодифицирующийся код (self-modifying code, SMC). JIT-код - это машинный код, динамически созданный JIT-компилятором. Например, языки Microsoft .NET компилируются в промежуточный язык (MSIL), который для выполнения ядром ЦП преобразуется в машинный код конкретной архитектуры (x86, x64, ARM и т. д.). SMC - код, создаваемый или изменяемый текущим потоком команд. Распространённый пример SMC - закодированный шелл-код, который декодируется и выполняется во время работы. И JIT, и SMC требуют записать в память новые данные, которые впоследствии выбираются для выполнения.

В ядре ARM существуют отдельные строки кэша для команд (i-cache) и данных (d-cache): команды выполняются из i-cache, а обращения к памяти проходят через d-cache. Согласованность этих строк кэша не гарантируется, то есть данные, записанные в один кэш, могут стать видимыми другому не сразу. Предположим, например, что i-cache содержит четыре команды из потока команд, а пользователь создаёт новые или изменённые команды в том же месте (тем самым обновляя d-cache). Поскольку кэши не согласованы, i-cache может не узнать о недавнем изменении и выполнить устаревшие команды, что способно привести к загадочным сбоям или неправильным результатам. При разработке JIT-систем или шелл-кода такая ситуация явно нежелательна. Решение состоит в том, чтобы явно заставить i-cache обновиться (это также называют сбросом кэша). В ARM это делается обновлением регистра сопроцессора управления системой (CP15):

```
01: 4F F0 00 00   MOV.W     R0, #0
02: 07 EE 15 0F   MCR       p15, 0, R0,c7,c5, 0
```

Большинство операционных систем предоставляет интерфейс для этой операции, поэтому писать её самостоятельно не требуется. В Linux используется `__clear_cache`, а в Windows - **`FlushInstructionCache`**.

Примитивы синхронизации

В ARM нет команды, подобной `cmpxchg` (сравнение и обмен) в x86; вместо неё используются две команды: LDREX и STREX. Они подобны LDR/STR, но перед загрузкой или сохранением получают исключительный доступ к адресу памяти. Вместе они обычно применяются для реализации встроенных операций сравнения и обмена. Например:

ARM:

```
01: 01 21         MOVS      R1, #1
02:             loc_100C4B0
03: 54 E8 00 2F   LDREX.W   R2, [R4]
04: 1A B9         CBNZ      R2, loc_100C4BE
05: 44 E8 00 13   STREX.W   R3, R1, [R4] ; результат находится в r3
06: 00 2B         CMP       R3, #0
07: F8 D1         BNE       loc_100C4B0
```

Псевдокод на C:

```
if (InterlockedCompareExchange(&r4, 1, 0) == 0) { do stuff; }
```

Строка 3 выполняет атомарную загрузку в R2 и сравнивает значение с 0; если оно равно нулю, то обменивается на ноль, а результат возвращается в R3. В действительности это реализация `InterlockedCompareExchange` в Windows.

Иногда встречается код с командами DMB, DSB и ISB. Это барьерные команды, которые перед выполнением последующих команд обеспечивают синхронизацию обращений к памяти и выборки команд. В некоторых случаях это необходимо, поскольку обращения к памяти и команды могут выполняться не по порядку (то есть ЦП способен выполнить команды в порядке, отличном от представленного в ассемблерном коде), а другие выполняющиеся потоки могут не увидеть обновлённый результат и, следовательно, получить несогласованное представление данных. Поэтому эти команды часто встречаются в коде, реализующем блокировки.

Системные службы и механизмы

При загрузке ядро ARM начинает выполнять код в состоянии ARM по адресу памяти `0x00000000` или `0xFFFF0000` в зависимости от параметра сопроцессора 15. Это определяется битом вектора (V) регистра управления системой (CP15, C1/C0). Если он равен 0, вектор исключения находится по адресу `0x00000000`; в противном случае - по адресу `0xFFFF0000`. Этот адрес обычно расположен во флеш-памяти (оперативная память ещё не инициализирована и использовать её нельзя), а находящееся там содержимое обычно называется векторами исключений. В ARM существует список предопределённых векторов, начинающийся по базовому адресу. Обработчик исключения RESET стоит в таблице первым, поэтому выполняется после события сброса. Поскольку это первый выполняемый код, обычно он начинает с базовой настройки оборудования и запускает процесс загрузки. Ниже приведён вектор исключений из реального устройства:

```
01: 00000000 1A 00 00 EA   B  vect_RESET
02: 00000004 12 00 00 EA   B  vect_UNDEFINED_INSTRUCTION

03: 00000008 12 00 00 EA   B  vect_SUPERVISOR_CALL ; (для SWI/SVC)
04: 0000000C 12 00 00 EA   B  vect_PREFETCHABORT
05: ...
06: 00000054              vect_UNDEFINED_INSTRUCTION
07: 00000054 FE FF FF EA   B  vect_UNDEFINED_INSTRUCTION
08: 00000058              vect_SUPERVISOR_CALL
09: 00000058 FE FF FF EA   B  vect_SUPERVISOR_CALL
10: 0000005C              vect_PREFETCHABORT
11: 0000005C FE FF FF EA   B  vect_PREFETCHABORT
12: ...
13: 00000070              vect_RESET
14: 00000070 1C F1 9F E5   LDR PC, =0x10000078
15:               ; код отображён по адресу 0x10000078
16:               ; начать выполнение там
17: ...
18: 10000078 18 01 9F E5   LDR R0, =0x2001
19: 1000007C 11 0F 0F EE   MCR p15, 0, R0,c15,c1, 0
20:               ; инициализирует регистр, зависящий от производителя
```

```
21: 10000080 00 00 A0 E1   NOP
22: 10000084 00 00 A0 E1   NOP
23: 10000088 00 00 A0 E1   NOP
24: 1000008C 78 00 A0 E3   MOV R0, #0x78
25: 10000090 10 0F 01 EE   MCR p15, 0, R0,c1,c0, 0
26:              ; инициализирует регистр управления системой
```

После инициализации оборудования код исключения сброса переходит к загрузчику, который обычно находится во флеш-памяти, на съёмном носителе (MMC, SD-карте и т. д.) либо в другом виде хранилища. Некоторые устройства используют U-Boot - популярный загрузчик с открытым исходным кодом. Загрузчик продолжает инициализацию оборудования, читает образ операционной системы из хранилища, отображает его в основную память и передаёт ему управление. После этого загружается операционная система, и система становится готова к работе.

Операционная система управляет аппаратными ресурсами и предоставляет пользователям службы. Поскольку пользовательский код (обычно работающий в режиме USR) имеет меньше привилегий, чем код ядра или операционной системы (обычно работающий в режиме SVC), для запроса службы ОС ему необходимо использовать определённый интерфейс. На практике такой интерфейс предоставляется через программное прерывание или специальную команду ловушки процессора, а служба обычно реализуется в виде системных вызовов. (Например, в 32-разрядной Linux x86 для системного вызова можно использовать прерывание `0x80` или специальную команду SYSENTER; в x64 это обеспечивает команда SYSCALL.) В ARM специальной команды системного вызова нет, поэтому системные вызовы реализуются посредством программного прерывания. При возникновении программного прерывания процессор переключается в режим супервизора для его обработки. Программные прерывания могут вызываться командой SWI/SVC. (Эти команды идентичны и отличаются лишь названием.)

Обе команды принимают непосредственное значение как параметр: некоторые операционные системы используют его как индекс таблицы системных вызовов, а другие не используют параметр, но требуют поместить номер системного вызова в регистр (например, Windows использует для этого R12). В некоторых системах Linux номер системного вызова помещается в R7, а аргументы передаются через R0-R2. Например:

Linux (Ubuntu):

```
01: 05 20 A0 E1   MOV    R2, R5    ; третий аргумент
02: 06 10 A0 E1   MOV    R1, R6    ; второй аргумент
03: 09 00 A0 E1   MOV    R0, R9    ; первый аргумент
04: 92 70 A0 E3   MOV    R7, #0x92
; номер системного вызова
05: 00 00 00 EF   SVC    0 ; выполнить системный вызов
06: 04 00 70 E3   CMN    R0, #4
; проверить возвращённое значение
07: 00 30 A0 13   MOVNE  R3, #0
; условное перемещение на основании возвращённого значения
```

Windows RT:

```
ZwCreateFile (в ntdll)
4F F0 53 0C   MOV.W    R12, #0x53
01 DF         SVC      1
70 47         BX       LR
             ; конец функции ZwCreateFile
```

SVC переходит в режим супервизора, копирует соответствующие пользовательские регистры в их собственную область, выполняет запрошенную функцию и после завершения возвращает управление. Как SVC узнаёт, куда возвращаться? Обычно управление возвращается к команде после SVC. Перед обработкой исключения режим SVC копирует адрес возврата в R14_svc - банковый регистр режима SVC. Банковыми называются регистры, имеющие значение только в контексте определённого режима процессора. Например, R13_svc и R14_svc являются банковыми регистрами режима SVC, поэтому их значения отличаются от R13-R14 в режиме USR.

Хотя для программной точки останова существует специальная команда BKPT, реализовать её можно несколькими способами. Первый - команда BKPT, которая вызывает обработчик исключения аварийного завершения предварительной выборки; затем обработчик может передать управление отладчику. Другой распространённый способ - вызвать обработчик исключения неопределённой команды с помощью неопределённой команды. В кодировке команд ARM имеется зарезервированный диапазон, который гарантированно является неопределённым.

Команды

Каждая команда в состоянии ARM кодирует арифметическое условие для поддержки условного выполнения. По умолчанию используется условие AL (выполнять всегда). Это условие кодируется в четырёх старших значащих битах кода операции (биты 28-31); AL определяется как `0b1110`, то есть `0xE`. Если внимательно посмотреть на фрагменты ассемблерного кода (в состоянии ARM), можно заметить, что байт-код обычно заканчивается последовательностью вида `0xE*`. Более того, при просмотре команд в шестнадцатеричном редакторе видно, что `0xE*` часто встречается через каждые четыре байта. Например:

```
FE FF FF EA FE FF FF EA FE FF FF EA FE FF FF EA
FE FF FF EA 1C F1 9F E5 00 00 A0 E1 18 01 9F E5
11 0F 0F EE 00 00 A0 E1 00 00 A0 E1 00 00 A0 E1
78 00 A0 E3 10 0F 01 EE 00 00 A0 E1 00 00 A0 E1
00 00 A0 E1 00 00 A0 E3 17 0F 08 EE 17 0F 07 EE
```

Почему важно знать эту закономерность? Потому что код ARM иногда встроен в ПЗУ или флеш-память и может не соответствовать определённому формату файла. В ходе реверс-инжиниринга порой предоставляется лишь необработанный дамп памяти почти без контекста, и тогда бывает полезно угадать архитектуру по кодам операций. Другая причина связана с эксплойтами. Шелл-код может быть встроен в эксплойт, доставляемый по сети или внутри документа; для анализа шелл-код необходимо извлечь из остального сетевого трафика. Иногда это просто и его границы очевидны, иногда - нет. Однако умение распознать закономерность позволяет быстро предположить начало и конец кода. Способность узнавать границы команд в кажущейся случайной массе данных важна. Возможно, позже вы оцените её по достоинству.

Практический разбор

Изучив все основные понятия, в этом разделе можно применить их, полностью декомпилировав неизвестную функцию. Эта функция охватывает многие понятия и приёмы из главы, поэтому прекрасно подходит для проверки знаний. По пути будут освоены и новые навыки, на которые в предыдущих разделах давались лишь намёки. Функция довольно длинная, поэтому ради экономии места и удобства чтения она представлена в виде графа. Тело функции показано на рисунке 2-6; все обсуждаемые в этом разделе номера строк кода относятся к данному рисунку.

Ниже приведён контекст, в котором она вызывается:

```
01: 17 9B         LDR             R3, [SP,#0x5c]
02: 16 9A         LDR             R2, [SP,#0x58]
03: 51 46         MOV             R1, R10
04: 20 46         MOV             R0, R4
05: FF F7 98 FF   BL              unk_function
```

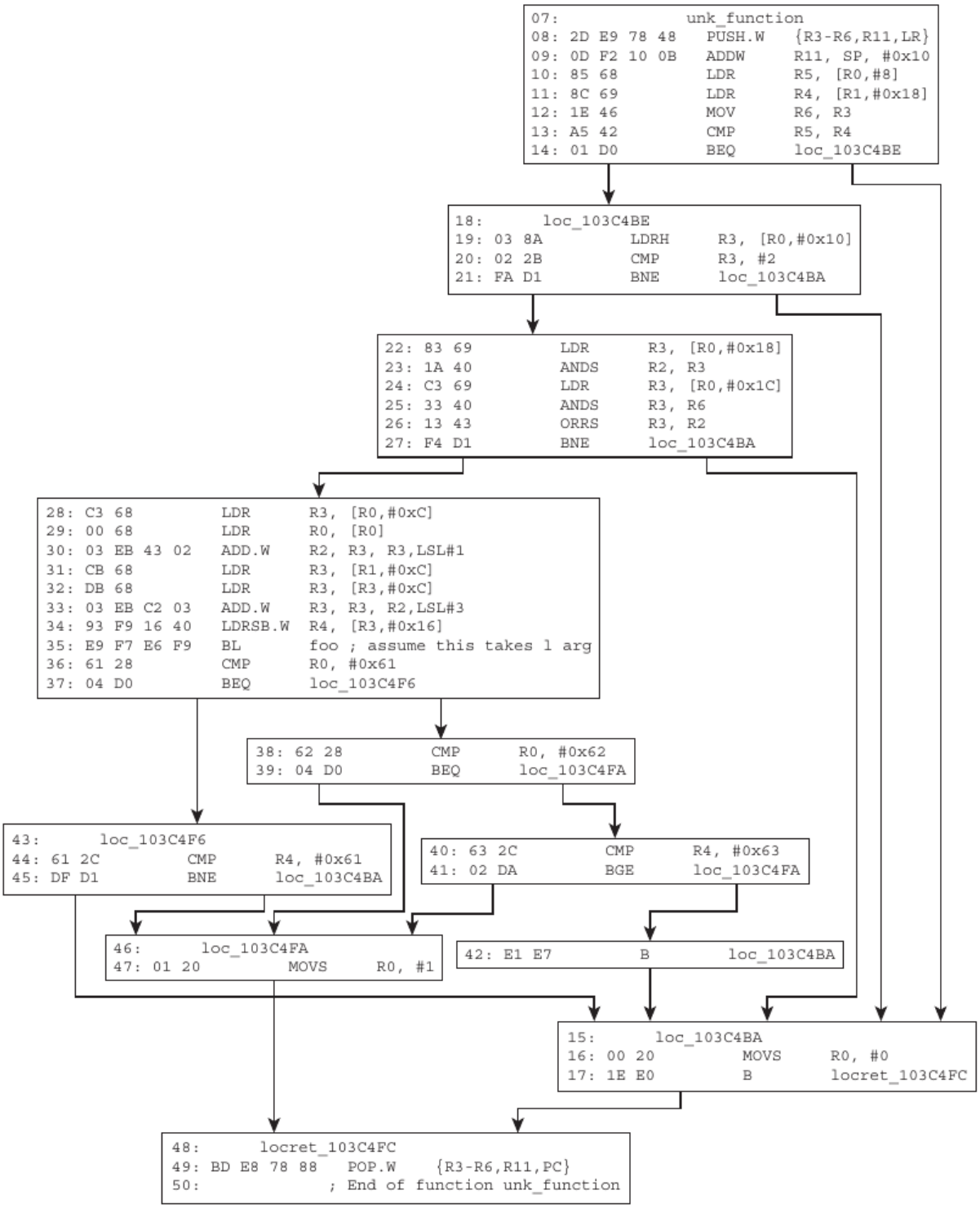

Рисунок 2-6. Граф потока управления функции `unk_function`

Текст листинга с рисунка 2-6:

```
07:             unk_function
08: 2D E9 78 48   PUSH.W   {R3-R6,R11,LR}
09: 0D F2 10 0B   ADDW     R11, SP, #0x10
10: 85 68         LDR      R5, [R0,#8]
11: 8C 69         LDR      R4, [R1,#0x18]
12: 1E 46         MOV      R6, R3
13: A5 42         CMP      R5, R4
14: 01 D0         BEQ      loc_103C4BE
15:             loc_103C4BA
16: 00 20         MOVS     R0, #0
17: 1E E0         B        locret_103C4FC
18:             loc_103C4BE
19: 03 8A         LDRH     R3, [R0,#0x10]
20: 02 2B         CMP      R3, #2
21: FA D1         BNE      loc_103C4BA
22: 83 69         LDR      R3, [R0,#0x18]
23: 1A 40         ANDS     R2, R3
24: C3 69         LDR      R3, [R0,#0x1C]
25: 33 40         ANDS     R3, R6
26: 13 43         ORRS     R3, R2
27: F4 D1         BNE      loc_103C4BA
28: C3 68         LDR      R3, [R0,#0xC]
29: 00 68         LDR      R0, [R0]
30: 03 EB 43 02   ADD.W    R2, R3, R3,LSL#1
31: CB 68         LDR      R3, [R1,#0xC]
32: DB 68         LDR      R3, [R3,#0xC]
33: 03 EB C2 03   ADD.W    R3, R3, R2,LSL#3
34: 93 F9 16 40   LDRSB.W  R4, [R3,#0x16]
35: E9 F7 E6 F9   BL       foo ; предположим, что функция принимает 1 аргумент
36: 61 28         CMP      R0, #0x61
37: 04 D0         BEQ      loc_103C4F6
38: 62 28         CMP      R0, #0x62
39: 04 D0         BEQ      loc_103C4FA
40: 63 2C         CMP      R4, #0x63
41: 02 DA         BGE      loc_103C4FA
42: E1 E7         B        loc_103C4BA
43:             loc_103C4F6
44: 61 2C         CMP      R4, #0x61
45: DF D1         BNE      loc_103C4BA
46:             loc_103C4FA
47: 01 20         MOVS     R0, #1
48:             locret_103C4FC
49: BD E8 78 88   POP.W    {R3-R6,R11,PC}
50:             ; конец функции unk_function
```

Приступая к неизвестной функции (или любому блоку кода), прежде всего нужно установить, что о ней известно наверняка. Ниже перечислены эти факты и основания для каждого из них:

• Код находится в состоянии Thumb, а набор команд - Thumb-2. Это известно потому, что: 1) в прологе и эпилоге (строки 1 и 49) используется конструкция PUSH/POP; 2) ширина команд составляет 16 или 32 бита; 3) для некоторых команд дизассемблер показывает суффикс `.W`, указывающий на 32-разрядную кодировку.
• Функция сохраняет R3-R6 и R11. Это известно потому, что они соответственно сохраняются и восстанавливаются в прологе (строка 1) и эпилоге (строка 49).

• Функция принимает не более четырёх аргументов (R0-R3) и возвращает логическое значение (R0). Это известно потому, что согласно двоичному интерфейсу приложений ARM (Application Binary Interface, ABI) первые четыре параметра передаются в R0-R3 (остальные помещаются в стек), а возвращаемое значение находится в R0. В данном случае говорится именно «не более четырёх», поскольку перед вызовом функции в строке 5 регистрам R0-R3 присваиваются некоторые значения и других команд записи в стек (для дополнительных аргументов) не видно. На этом этапе прототип функции выглядит так:

```
BOOL unk_function(int, int, int, int)
```

• Тип первых двух аргументов - «указатель на объект». Это известно потому, что R0 и R1 служат базовыми адресами в командах загрузки (строки 10-11). Скорее всего, это структуры, поскольку выполняются обращения по смещениям `0x10`, `0x18`, `0x1c` и т. д. (строки 10, 11, 19, 22, 24, 28 и прочие). Почти наверняка это не массивы, поскольку схема обращений и загрузки не последовательна. Без дополнительного контекста неизвестно, указывают ли R0 и R1 на один тип структуры или на два разных. Пока можно предположить, что это два разных типа. Обновлённый прототип выглядит так:

```
BOOL unk_function(struct1 *, struct2 *, int, int)
```

• `loc_103C4BA` - путь выхода с возвращением 0, `loc_103C4FA` - путь выхода с возвращением 1, а `locret_103C4FC` выполняет возврат из функции. Следовательно, переходы к этим адресам означают завершение работы функции.
• Третий и четвёртый аргументы имеют целочисленный тип. Это известно потому, что R2 и R3 используются в операциях AND/ORR (строки 23, 25 и 26). Хотя они действительно могут оказаться указателями, это маловероятно, если только указатели не кодировались или декодировались; даже если бы они были указателями, следовало бы ожидать их применения в операциях загрузки и сохранения, чего здесь нет.
• Хотя R11 устанавливается на 0x10 байт выше указателя стека, после этой команды он больше нигде не используется. Следовательно, его можно не учитывать.
• Функция `foo` (строка 35) принимает один аргумент. Из-за нехватки места её полное тело здесь не приводится. Для простоты будем считать это заданным условием.

Перечислив известные факты, теперь нужно логически вывести из них другие полезные сведения. Следующая важная задача - глубже исследовать две обнаруженные неизвестные структуры. Очевидно, полностью восстановить их компоновку невозможно, поскольку функция обращается лишь к некоторым элементам; тем не менее типы полей определить можно.

R0 имеет тип `struct1 *`. В строке 10 загружается поле по смещению `0x8`, а затем сравнивается с R4 (строка 13). R4 является полем по смещению `0x18` в структуре `struct2` (R1). Поскольку эти значения сравниваются, известно, что они имеют одинаковый тип. В строке 13 сравниваются эти два поля. Если они равны, выполнение переходит к `loc_103C4BE`; в противном случае возвращается 0 (строка 15). Поскольку выполняется сравнение на равенство, можно заключить, что оба поля являются целыми числами.

В строке 19 загружается ещё одно поле `struct1` и сравнивается с 2; если значения не равны, возвращается 0 (строка 21). Из команды LDRH (загрузка полуслова) можно заключить, что поле имеет тип `short`.

В строках 22-23 загружается ещё одно поле `struct1`, над которым выполняется AND с третьим аргументом (предположительно целым числом). В строках 25-27 нечто подобное выполняется с четвёртым аргументом. Из этих операций можно заключить, что поля по смещениям `0x18` и `0x1c` имеют целочисленный тип.

Полученные к настоящему моменту определения структур:

```
struct1
...
   +0x008 field08_i ; тот же тип, что у struct2.field18_i
...
   +0x010 field10_s ; short
...
   +0x018 field18_i ; int
   +0x01c field1c_i ; int

struct2
...
   +0x018 field18_i ; тот же тип, что у struct1.field08_i
```

> **Примечание.** В именах полей структуры можно выработать привычку указывать смещение и «тип». Например, суффикс `i` означает целое число (или некоторый универсальный 32-разрядный тип), `s` - `short` (16 бит), `c` - `char` (1 байт), а `p` - указатель некоторого типа. Это позволяет быстро вспоминать типы полей. Определив их истинное назначение, поля можно переименовать более осмысленно.

Имея эти типы, уже можно восстановить псевдокод всех строк с 1-й по 27-ю:

```
struct1 *arg1 = ...;
struct1 *arg2 = ...;
int arg3 = ...;
int arg4 = ...;

BOOL result = unk_function(arg1, arg2, arg3, arg4);
if (arg1->field08_i == arg2->field18_i) {
  if (arg1->field10_s != 2) return 0;
  if ( ((arg1->field18_i & arg3) |
        (arg1->field1c_i & arg4)
       ) != 0
     ) return 0;
...
} else {

    return 0;
  }
```

> **Примечание.** Использование операции AND над двумя соседними целочисленными полями выглядит немного подозрительно. Обычно это означает, что на самом деле перед нами 64-разрядные целые числа, разделенные между двумя регистрами или ячейками памяти. Это распространенный шаблон обращения к 64-разрядным константам на 32-разрядных архитектурах.

Внимательные читатели заметят, что строки 25-27 могут показаться несколько избыточными. ANDS устанавливает флаги условий, ORRS тут же перезаписывает их, а BNE использует флаг от ORRS; следовательно, флаги, установленные ANDS, в действительности не нужны. Компилятор порождает эту избыточность, поскольку оптимизирует плотность кода: команда AND будет иметь длину 4 байта, тогда как ANDS - всего 2 байта. Такой же оптимизации подвергаются MOV и MOVS. Этот шаблон часто встречается в коде, оптимизированном для Thumb.

Строка 28 загружает в R3 еще одно поле из `struct1`; строка 29 загружает в R0 значение по нулевому смещению той же структуры; строка 30 присваивает R2 значение `R3*3` (то есть `R3+(R3<<1)`). Строка 31 загружает в R3 поле из `struct2`, а затем обращается к другому полю, используя загруженное значение как базовый указатель. Это подразумевает, что по смещению `0xC` внутри `struct2` находится указатель на другую структуру. Строка 32 загружает в R3 поле этой новой структуры; строка 33 изменяет его на `R3+R2*8`; строка 34 использует полученное значение как базовый адрес и загружает в R4 знаковое значение `short` по смещению `0x16` еще одной структуры.

Прежде чем продолжить, обновим определения структур:

```
struct1
   +0x000 field00_i ; int
...
   +0x008 field08_i ; тот же тип, что у struct2.field18_i
   +0x00c field0c_i ; целое число
...
   +0x010 field10_s ; short
...
   +0x018 field18_i ; int
   +0x01c field1c_i ; int
...

struct2
...
   +0x00c field0c_p ; struct3 *
...
   +0x018 field18_i ; тот же тип, что у struct1.field08_i
...

struct3
...
   +0x00c field0c_p ; struct4 *
...

struct4 (size=0x18=24) // почему?

...
   +0x016 field16_c; char
   +0x017 end
```

О наличии массива можно было догадаться по множителю умножения/масштабирования (строки 30 и 33); двух массивов здесь нет, поскольку R2-R3 в строке 30 - не базовый адрес, а индекс. Кроме того, умножение базового адреса на 3 не имело бы смысла. В строке 33 базовым адресом массива служит R3, поскольку он индексируется с помощью R2. Мы заключили, что размер каждого элемента массива должен составлять `0x18` (24), поскольку после упрощения получилось `R2*3*8`, где R2 - индекс, а 24 - масштаб.

На рисунке 2-7 показаны взаимосвязи между четырьмя структурами.

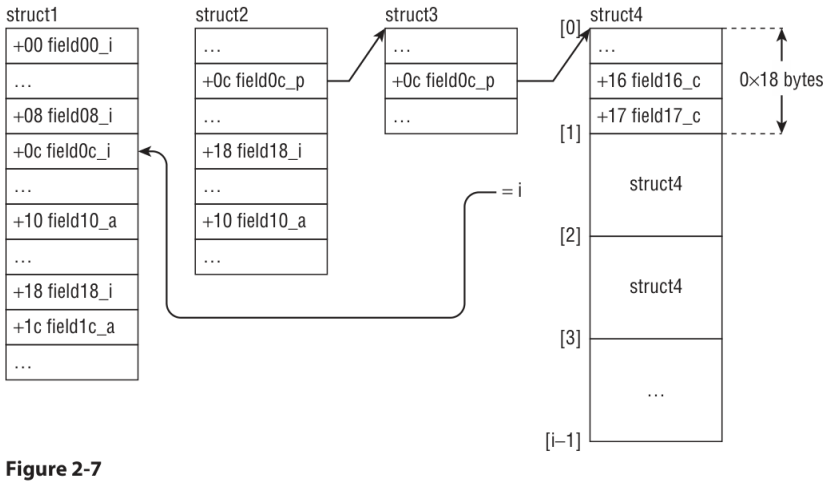

Figure 2-7

Рисунок 2-7. Взаимосвязи между четырьмя структурами

Ниже приведен псевдокод для строк 28-35:

```
r3 = arg1->field0c_i;
r2 = r3 + r3<<1
   = arg1->field0c_i*3;
r3 = arg2->field0c_p;
r3 = arg2->field0c_p->field0c_p;
r3 = arg2->field0c_p->field0c_p + r2*8
   = arg2->field0c_p->field0c_p + arg1->field0c_i*24;
   = arg2->field0c_p->field0c_p[arg1->field0c_i];
r4 = arg2->field0c_p->field0c_p[arg1->field0c_i].field16_c;
r0 = foo(arg1->field00_i);
```

Остальная часть функции просто сравнивает возвращаемое `foo` значение с R4. Теперь полный псевдокод выглядит так:

```
struct1 *arg1 = ...;
struct2 *arg2 = ...;

int arg3 = ...;
int arg4 = ...;

BOOL result = unk_function(arg1, arg2, arg3, arg4);

BOOL unk_function(struct1 *arg1, struct2 *arg2, int arg3, int arg4)
{
  char a;
  int b;
  if (arg1->field08_i == arg2->field18_i) {
    if (arg1->field10_s != 2) return 0;
    if ( ((arg1->field18_i & arg3) |
          (arg1->field1c_i & arg4)
         ) != 0
       ) return 0;
    b = foo(arg1->field00_i);
    a = arg2->field0c_p->field0c_p[arg1->field0c_i].field16_c;
    if (b == 0x61 && a != 0x61) {
      return 0;
    } else { return 1;}
    if (b == 0x62 && a >= 0x63) {
      return 1;
    } else { return 0;}
  } else {
   return 0;
  }
}
```

Хотя в этой функции использовалось несколько взаимосвязанных структур данных, полная компоновка которых неясна, мы все же смогли восстановить некоторые типы полей и их связи с другими полями. Мы также научились распознавать ширину и знаковость типа, рассматривая связанную с ним команду и код условия.

Дальнейшие шаги

В этой главе были изложены основополагающие навыки, необходимые для статического обратного анализа кода ARM. Мы намеренно не стали писать справочник по командам и опустили множество подробностей; чтобы совершенствовать свои навыки, вам придется выполнять упражнения, практиковаться и читать руководства ARM (эти занятия взаимосвязаны). Техническое справочное руководство может оказаться довольно сложным для восприятия, но знания, полученные из этой главы, значительно облегчат его понимание.

Следующим шагом должна стать покупка устройства ARM и эксперименты с ним. Существует множество устройств ARM, но, возможно, лучше всего для обучения подходят BeagleBoard и PandaBoard. Это отладочные платы, предназначенные для знакомства людей со встраиваемой разработкой на платформе ARM;

они относительно мощные, дешевые ($150-$170), хорошо документированные и обладают большим сообществом пользователей. (Возможно, вам встретится не так много людей, разбирающихся в ассемблере ARM, но это не страшно, потому что вы уже прочитали эту главу. Помощь обычно может понадобиться в вопросах, связанных со встроенными периферийными устройствами и с тем, как они программируются и управляются.) На эти платы можно установить Linux с полноценной средой разработки, поэтому проверить свои знания ARM очень просто.

Упражнения

Упражнения включены для того, чтобы вы хорошо усвоили понятия и повысили свою мотивацию. Некоторые упражнения были намеренно подобраны так, чтобы в них встречались команды, не рассмотренные в этой главе, и вы привыкали читать руководство (это очень важная привычка); контекст вызова также опущен, чтобы заставить вас больше думать. Каждая функция является самодостаточной, чтобы облегчить полную декомпиляцию; некоторые из них подобраны так, что, выполнив достаточное количество упражнений, вы сможете проверить свой ответ. Рекомендуется прямо на самих упражнениях писать комментарии и примечания, а также отмечать связи между ветвями и метками.

Для кода в каждом упражнении выполните по порядку следующие действия (когда это возможно):

• Определите, находится ли он в состоянии Thumb или ARM.

• Объясните семантику каждой команды. Если это команда LDR/STR, объясните также режим адресации.

• Определите типы (ширину и знаковость) всех возможных объектов. Для структур по возможности восстановите размер поля, его тип и понятное имя. Восстановить удастся не все поля структуры, поскольку функция может обращаться лишь к нескольким из них. Для каждого восстановленного типа объясните себе (или кому-либо еще), как вы его вывели.

• Восстановите прототип функции.

• Найдите пролог и эпилог функции.

• Объясните, что делает функция, а затем напишите для нее псевдокод.

• Декомпилируйте функцию обратно в C и дайте ей осмысленное имя.

1. На рисунке 2-8 показана функция, принимающая два аргумента. Сначала она может показаться довольно сложной, однако выполняемая ею операция встречается очень часто. Наберитесь терпения.

2. На рисунке 2-9 показана функция, найденная в таблице экспорта.

3. Ниже приведена простая функция:

```
01:             mystery3
02: 83 68         LDR             R3, [R0,#8]
03: 0B 60         STR             R3, [R1]
04: C3 68         LDR             R3, [R0,#0xC]
05: 00 20         MOVS            R0, #0
06: 4B 60         STR             R3, [R1,#4]
07: 70 47         BX              LR
08:             ; Конец функции mystery3
```

4. На рисунке 2-10 показана еще одна простая функция.

5. Функция на рисунке 2-11 тоже проста. Настоящие имена строк были удалены, чтобы вы не могли схитрить, отыскав их в Интернете.

6. В функции на рисунке 2-12 придется немного повозиться с битами.

7. На рисунке 2-13 показана распространенная процедура, но вы, возможно, еще не видели такой ее реализации.

8. На рисунке 2-14 `byteArray` - это массив из 256 символов, содержимое которого имеет вид `byteArray[] = {0, 1, ..., 0xff}`.

9. Что делает функция, показанная на рисунке 2-15?

10. На рисунке 2-16 показана функция из Windows RT. При необходимости прочитайте MSDN. Не обращайте внимания на защитные процедуры PUSH/POP для cookie.

11. На рисунке 2-17 функция `sub_101651C` принимает три аргумента и ничего не возвращает. Если вы выполните это упражнение, можете похлопать себя по плечу.

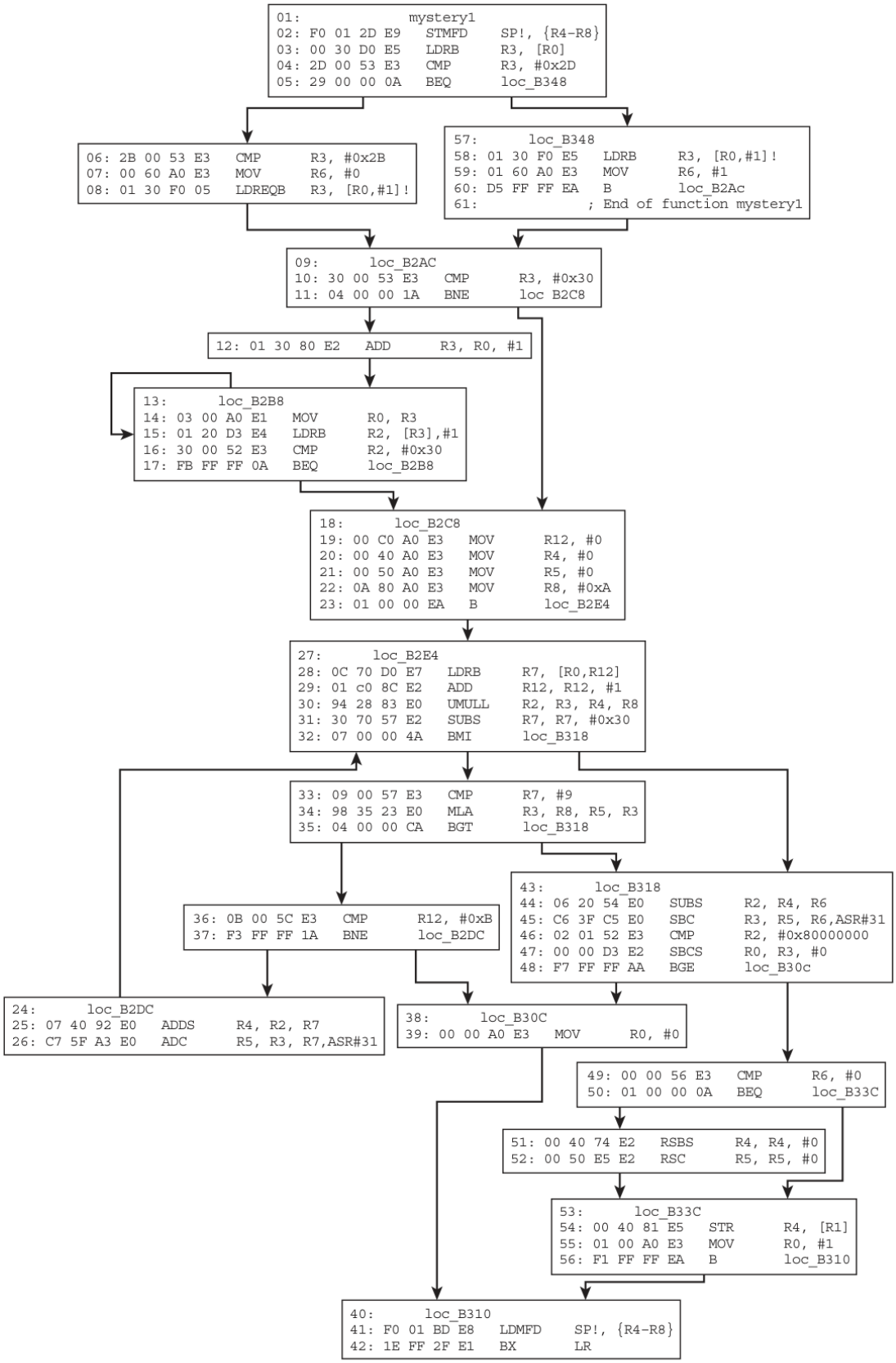

Figure 2-8

Рисунок 2-8. Граф потока управления функции mystery1

```
01:             mystery1
02: F0 01 2D E9   STMFD    SP!, {R4-R8}
03: 00 30 D0 E5   LDRB     R3, [R0]
04: 2D 00 53 E3   CMP      R3, #0x2D
05: 29 00 00 0A   BEQ      loc_B348
06: 2B 00 53 E3   CMP      R3, #0x2B
07: 00 60 A0 E3   MOV      R6, #0
08: 01 30 F0 05   LDREQB   R3, [R0,#1]!
09:             loc_B2AC
10: 30 00 53 E3   CMP      R3, #0x30
11: 04 00 00 1A   BNE      loc_B2C8
12: 01 30 80 E2   ADD      R3, R0, #1
13:             loc_B2B8
14: 03 00 A0 E1   MOV      R0, R3
15: 01 20 D3 E4   LDRB     R2, [R3],#1
16: 30 00 52 E3   CMP      R2, #0x30
17: FB FF FF 0A   BEQ      loc_B2B8
18:             loc_B2C8
19: 00 C0 A0 E3   MOV      R12, #0
20: 00 40 A0 E3   MOV      R4, #0
21: 00 50 A0 E3   MOV      R5, #0
22: 0A 80 A0 E3   MOV      R8, #0xA
23: 01 00 00 EA   B        loc_B2E4
24:             loc_B2DC
25: 07 40 92 E0   ADDS     R4, R2, R7
26: C7 5F A3 E0   ADC      R5, R3, R7,ASR#31
27:             loc_B2E4
28: 0C 70 D0 E7   LDRB     R7, [R0,R12]
29: 01 C0 8C E2   ADD      R12, R12, #1
30: 94 28 83 E0   UMULL    R2, R3, R4, R8
31: 30 70 57 E2   SUBS     R7, R7, #0x30
32: 07 00 00 4A   BMI      loc_B318
33: 09 00 57 E3   CMP      R7, #9
34: 98 35 23 E0   MLA      R3, R8, R5, R3
35: 04 00 00 CA   BGT      loc_B318
36: 0B 00 5C E3   CMP      R12, #0xB
37: F3 FF FF 1A   BNE      loc_B2DC
38:             loc_B30C
39: 00 00 A0 E3   MOV      R0, #0
40:             loc_B310
41: F0 01 BD E8   LDMFD    SP!, {R4-R8}
42: 1E FF 2F E1   BX       LR
43:             loc_B318
44: 06 20 54 E0   SUBS     R2, R4, R6
45: C6 3F C5 E0   SBC      R3, R5, R6,ASR#31
46: 02 01 52 E3   CMP      R2, #0x80000000
47: 00 00 D3 E2   SBCS     R0, R3, #0
48: F7 FF FF AA   BGE      loc_B30C
49: 00 00 56 E3   CMP      R6, #0
50: 01 00 00 0A   BEQ      loc_B33C
51: 00 40 74 E2   RSBS     R4, R4, #0
52: 00 50 E5 E2   RSC      R5, R5, #0
53:             loc_B33C
54: 00 40 81 E5   STR      R4, [R1]
55: 01 00 A0 E3   MOV      R0, #1
56: F1 FF FF EA   B        loc_B310
57:             loc_B348
58: 01 30 F0 E5   LDRB     R3, [R0,#1]!
59: 01 60 A0 E3   MOV      R6, #1
60: D5 FF FF EA   B        loc_B2Ac
61:             ; Конец функции mystery1
```

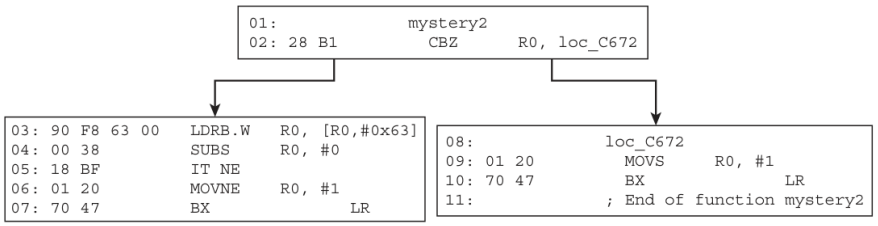

Figure 2-9

Рисунок 2-9. Граф потока управления функции mystery2

```
01:             mystery2
02: 28 B1         CBZ      R0, loc_C672
03: 90 F8 63 00   LDRB.W   R0, [R0,#0x63]
04: 00 38         SUBS     R0, #0
05: 18 BF         IT NE
06: 01 20         MOVNE    R0, #1
07: 70 47         BX       LR
08:             loc_C672
09: 01 20         MOVS     R0, #1
10: 70 47         BX       LR
11:             ; Конец функции mystery2
```

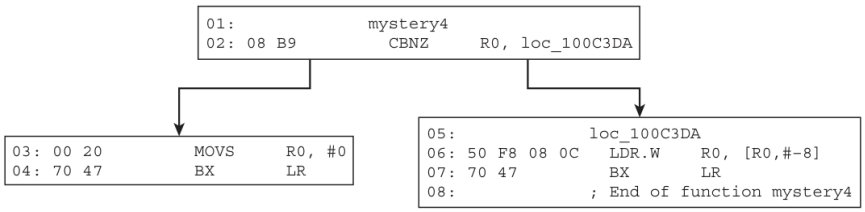

Figure 2-10

Рисунок 2-10. Граф потока управления функции mystery4

```
01:             mystery4
02: 08 B9         CBNZ     R0, loc_100C3DA
03: 00 20         MOVS     R0, #0
04: 70 47         BX       LR
05:             loc_100C3DA
06: 50 F8 08 0C   LDR.W    R0, [R0,#-8]
07: 70 47         BX       LR
08:             ; Конец функции mystery4
```

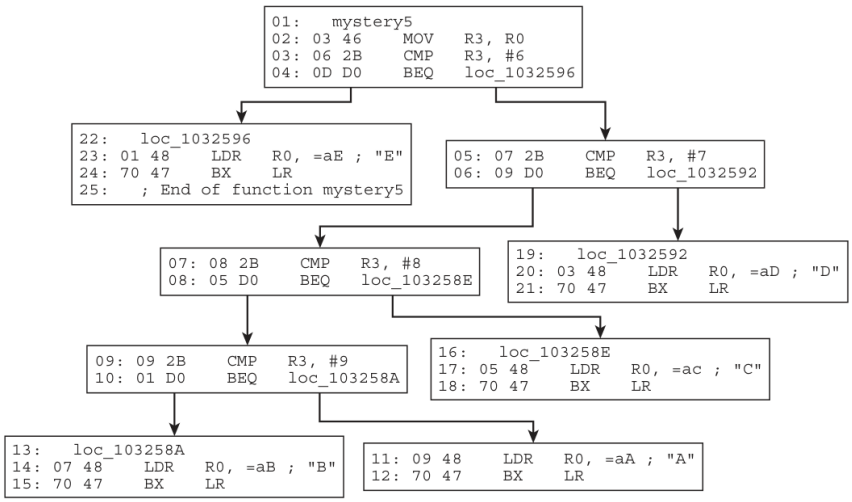

Figure 2-11

Рисунок 2-11. Граф потока управления функции mystery5

```
01:             mystery5
02: 03 46         MOV       R3, R0
03: 06 2B         CMP       R3, #6
04: 0D D0         BEQ       loc_1032596
05: 07 2B         CMP       R3, #7
06: 09 D0         BEQ       loc_1032592
07: 08 2B         CMP       R3, #8
08: 05 D0         BEQ       loc_103258E
09: 09 2B         CMP       R3, #9
10: 01 D0         BEQ       loc_103258A
11: 09 48         LDR       R0, =aA ; "A"
12: 70 47         BX        LR
13:             loc_103258A
14: 07 48         LDR       R0, =aB ; "B"
15: 70 47         BX        LR
16:             loc_103258E
17: 05 48         LDR       R0, =aC ; "C"
18: 70 47         BX        LR
19:             loc_1032592
20: 03 48         LDR       R0, =aD ; "D"
21: 70 47         BX        LR
22:             loc_1032596
23: 01 48         LDR       R0, =aE ; "E"
24: 70 47         BX        LR
25:             ; Конец функции mystery5
```

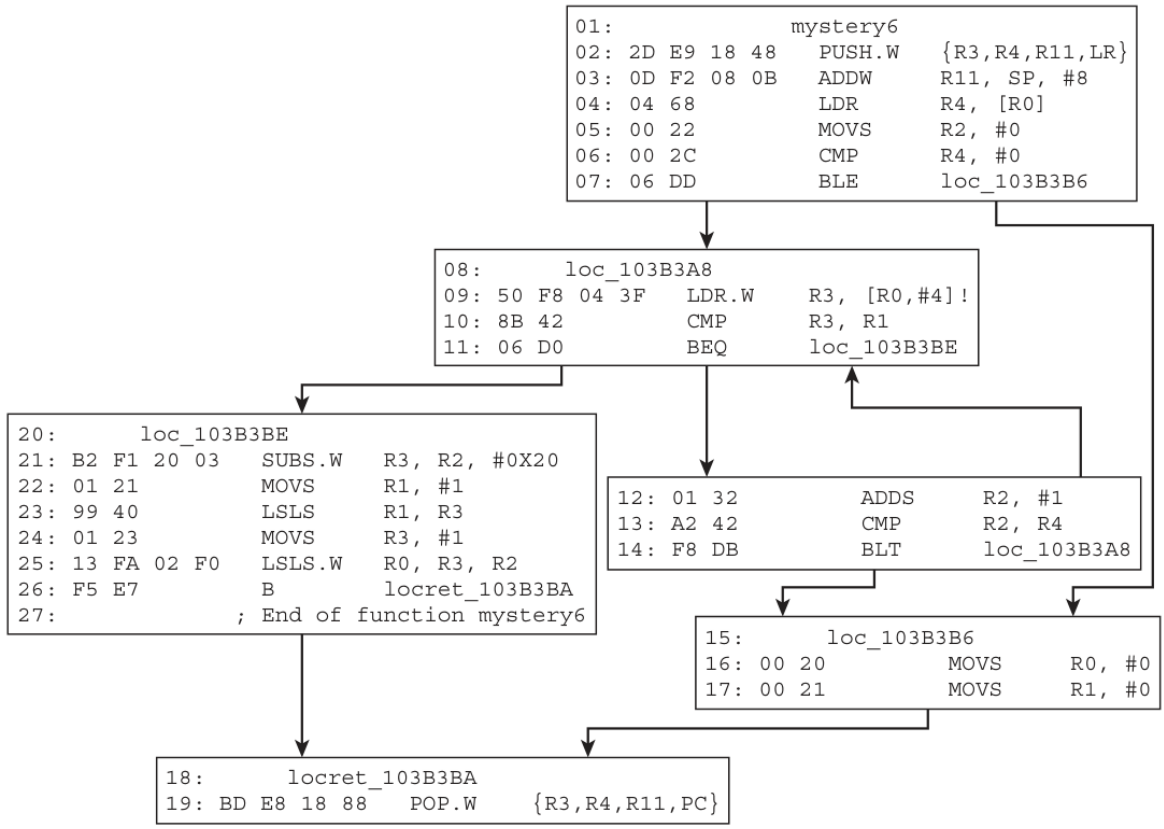

Figure 2-12

Рисунок 2-12. Граф потока управления функции mystery6

```
01:             mystery6
02: 2D E9 18 48   PUSH.W   {R3,R4,R11,LR}
03: 0D F2 08 0B   ADDW     R11, SP, #8
04: 04 68         LDR      R4, [R0]
05: 00 22         MOVS     R2, #0
06: 00 2C         CMP      R4, #0
07: 06 DD         BLE      loc_103B3B6
08:             loc_103B3A8
09: 50 F8 04 3F   LDR.W    R3, [R0,#4]!
10: 8B 42         CMP      R3, R1
11: 06 D0         BEQ      loc_103B3BE
12: 01 32         ADDS     R2, #1
13: A2 42         CMP      R2, R4
14: F8 DB         BLT      loc_103B3A8
15:             loc_103B3B6
16: 00 20         MOVS     R0, #0
17: 00 21         MOVS     R1, #0
18:             locret_103B3BA
19: BD E8 18 88   POP.W    {R3,R4,R11,PC}
20:             loc_103B3BE
21: B2 F1 20 03   SUBS.W   R3, R2, #0X20
22: 01 21         MOVS     R1, #1
23: 99 40         LSLS     R1, R3
24: 01 23         MOVS     R3, #1
25: 13 FA 02 F0   LSLS.W   R0, R3, R2
26: F5 E7         B        locret_103B3BA
27:             ; Конец функции mystery6
```

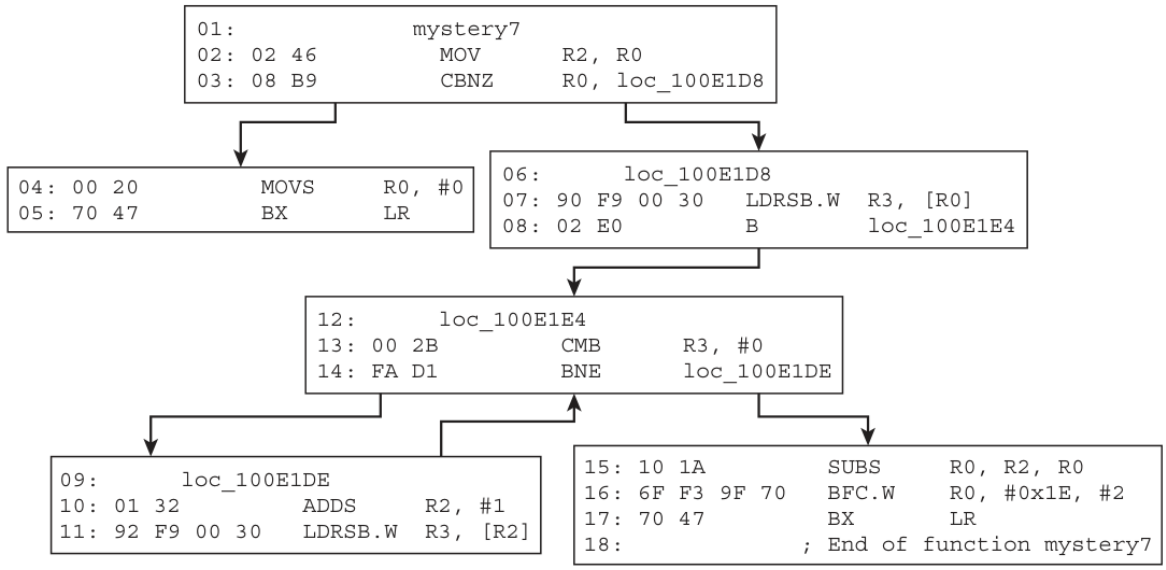

Figure 2-13

Рисунок 2-13. Граф потока управления функции mystery7

```
01:             mystery7
02: 02 46         MOV      R2, R0
03: 08 B9         CBNZ     R0, loc_100E1D8
04: 00 20         MOVS     R0, #0
05: 70 47         BX       LR
06:             loc_100E1D8
07: 90 F9 00 30   LDRSB.W  R3, [R0]
08: 02 E0         B        loc_100E1E4
09:             loc_100E1DE
10: 01 32         ADDS     R2, #1
11: 92 F9 00 30   LDRSB.W  R3, [R2]
12:             loc_100E1E4
13: 00 2B         CMB      R3, #0
14: FA D1         BNE      loc_100E1DE
15: 10 1A         SUBS     R0, R2, R0
16: 6F F3 9F 70   BFC.W    R0, #0x1E, #2
17: 70 47         BX       LR
18:             ; Конец функции mystery7
```

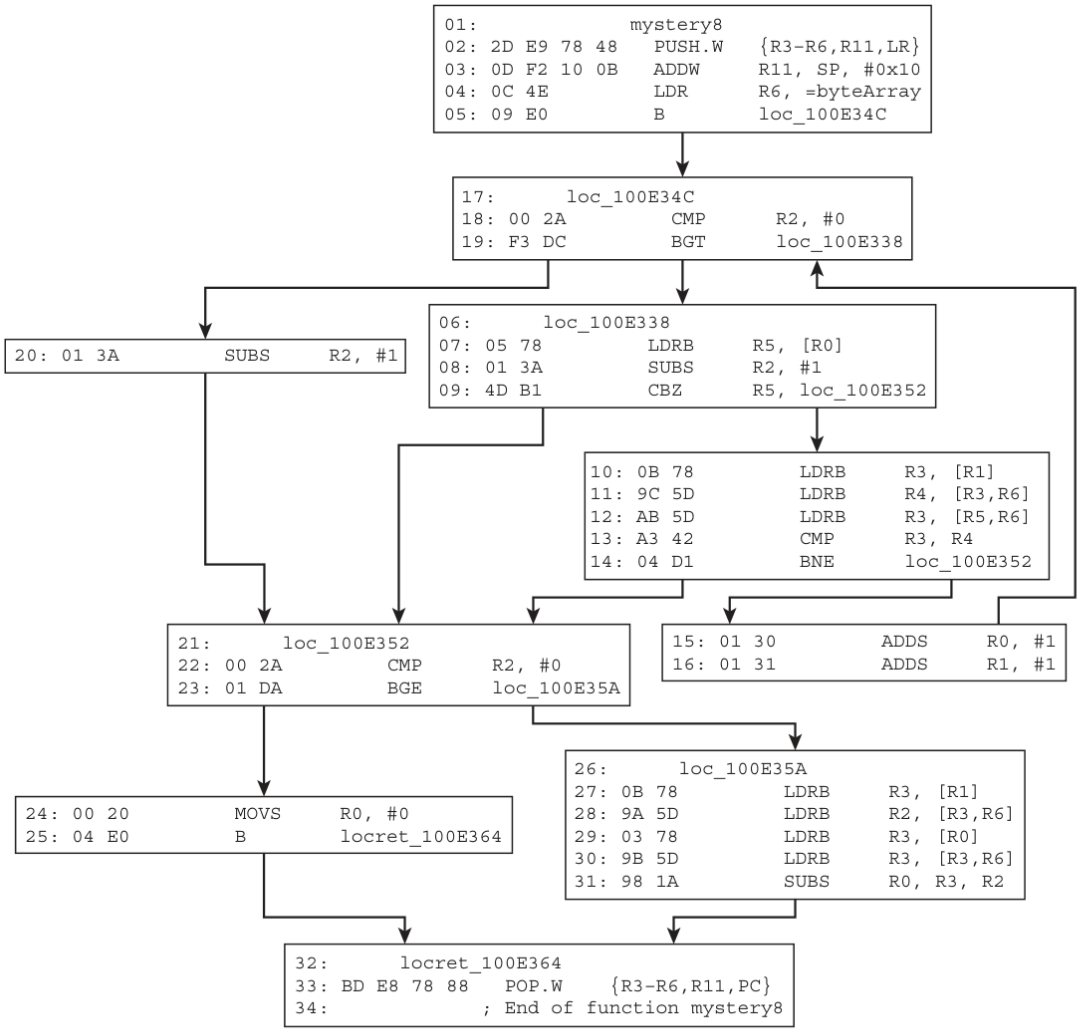

Figure 2-14

Рисунок 2-14. Граф потока управления функции mystery8

```
01:             mystery8
02: 2D E9 78 48   PUSH.W   {R3-R6,R11,LR}
03: 0D F2 10 0B   ADDW     R11, SP, #0x10
04: 0C 4E         LDR      R6, =byteArray
05: 09 E0         B        loc_100E34C
06:             loc_100E338
07: 05 78         LDRB     R5, [R0]
08: 01 3A         SUBS     R2, #1
09: 4D B1         CBZ      R5, loc_100E352
10: 0B 78         LDRB     R3, [R1]
11: 9C 5D         LDRB     R4, [R3,R6]
12: AB 5D         LDRB     R3, [R5,R6]
13: A3 42         CMP      R3, R4
14: 04 D1         BNE      loc_100E352
15: 01 30         ADDS     R0, #1
16: 01 31         ADDS     R1, #1
17:             loc_100E34C
18: 00 2A         CMP      R2, #0
19: F3 DC         BGT      loc_100E338
20: 01 3A         SUBS     R2, #1
21:             loc_100E352
22: 00 2A         CMP      R2, #0
23: 01 DA         BGE      loc_100E35A
24: 00 20         MOVS     R0, #0
25: 04 E0         B        locret_100E364
26:             loc_100E35A
27: 0B 78         LDRB     R3, [R1]
```

```
28: 9A 5D         LDRB     R2, [R3,R6]
29: 03 78         LDRB     R3, [R0]
30: 9B 5D         LDRB     R3, [R3,R6]
31: 98 1A         SUBS     R0, R3, R2
32:             locret_100E364
33: BD E8 78 88   POP.W    {R3-R6,R11,PC}
34:             ; Конец функции mystery8
```

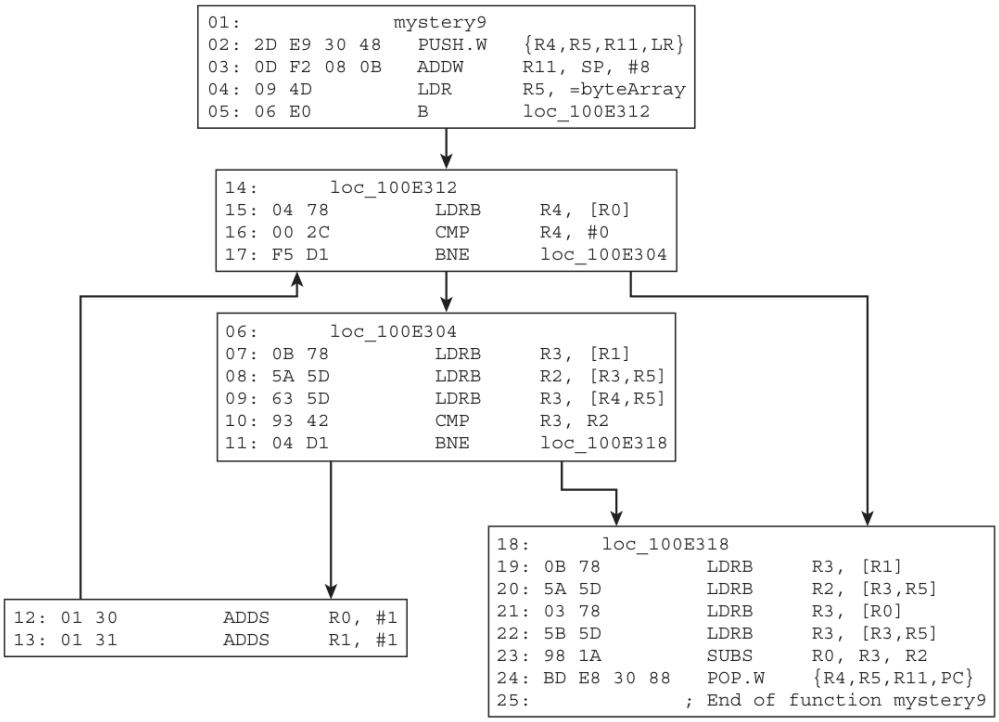

Figure 2-15

Рисунок 2-15. Граф потока управления функции mystery9

```
01:             mystery9
02: 2D E9 30 48   PUSH.W   {R4,R5,R11,LR}
03: 0D F2 08 0B   ADDW     R11, SP, #8
04: 09 4D         LDR      R5, =byteArray
05: 06 E0         B        loc_100E312
06:             loc_100E304
07: 0B 78         LDRB     R3, [R1]
08: 5A 5D         LDRB     R2, [R3,R5]
09: 63 5D         LDRB     R3, [R4,R5]
10: 93 42         CMP      R3, R2
11: 04 D1         BNE      loc_100E318
12: 01 30         ADDS     R0, #1
13: 01 31         ADDS     R1, #1
14:             loc_100E312
15: 04 78         LDRB     R4, [R0]
16: 00 2C         CMP      R4, #0
17: F5 D1         BNE      loc_100E304
18:             loc_100E318
19: 0B 78         LDRB     R3, [R1]
20: 5A 5D         LDRB     R2, [R3,R5]
21: 03 78         LDRB     R3, [R0]
22: 5B 5D         LDRB     R3, [R3,R5]
23: 98 1A         SUBS     R0, R3, R2
24: BD E8 30 88   POP.W    {R4,R5,R11,PC}
25:             ; Конец функции mystery9
```

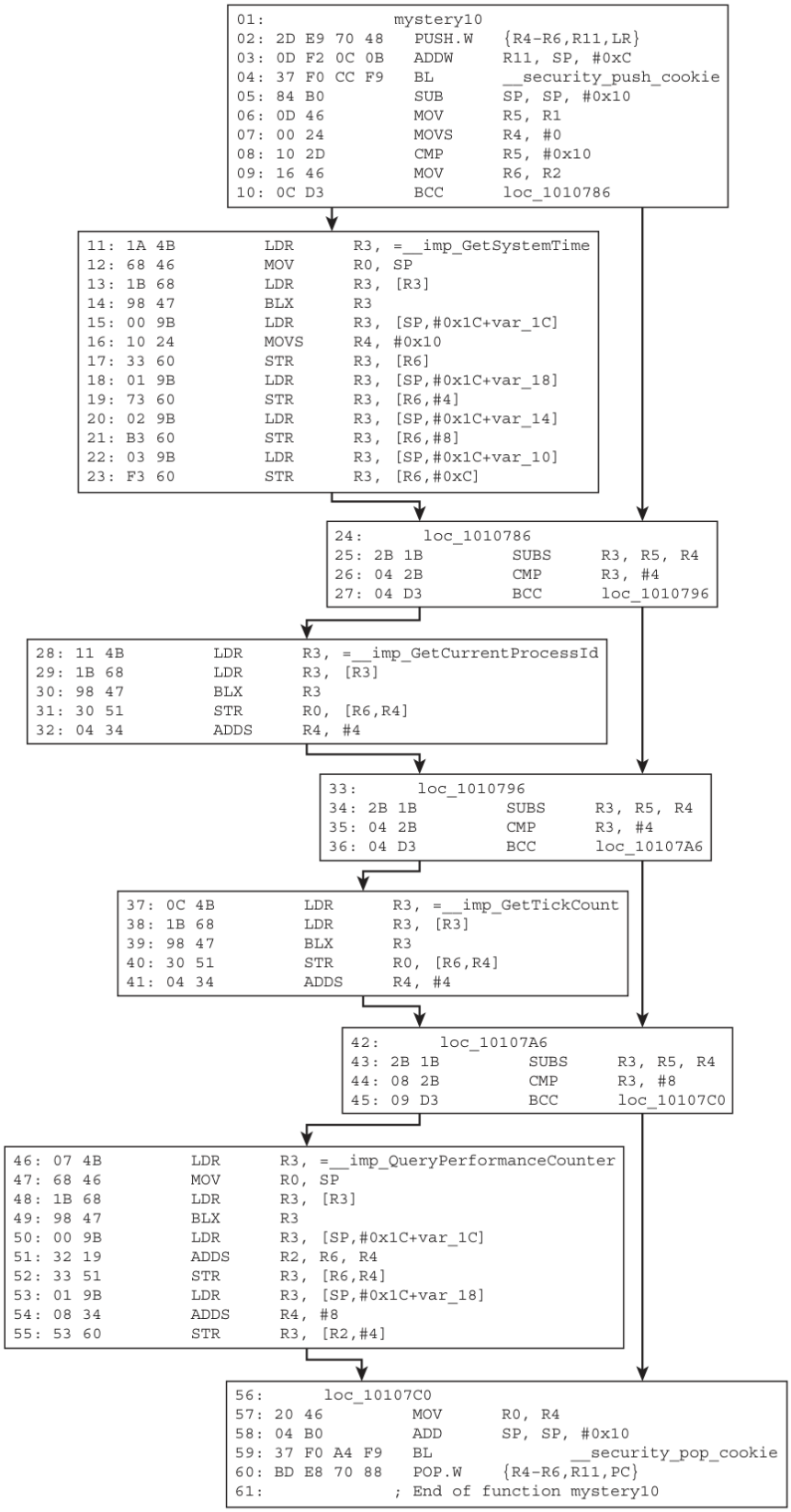

Figure 2-16

Рисунок 2-16. Граф потока управления функции mystery10

```
01:             mystery10
02: 2D E9 70 48   PUSH.W   {R4-R6,R11,LR}
03: 0D F2 0C 0B   ADDW     R11, SP, #0xC
04: 37 F0 CC F9   BL       __security_push_cookie
05: 84 B0         SUB      SP, SP, #0x10
06: 0D 46         MOV      R5, R1
07: 00 24         MOVS     R4, #0
08: 10 2D         CMP      R5, #0x10
09: 16 46         MOV      R6, R2
10: 0C D3         BCC      loc_1010786
11: 1A 4B         LDR      R3, =__imp_GetSystemTime
12: 68 46         MOV      R0, SP
13: 1B 68         LDR      R3, [R3]
14: 98 47         BLX      R3
15: 00 9B         LDR      R3, [SP,#0x1C+var_1C]
16: 10 24         MOVS     R4, #0x10
17: 33 60         STR      R3, [R6]
18: 01 9B         LDR      R3, [SP,#0x1C+var_18]
19: 73 60         STR      R3, [R6,#4]
20: 02 9B         LDR      R3, [SP,#0x1C+var_14]
21: B3 60         STR      R3, [R6,#8]
22: 03 9B         LDR      R3, [SP,#0x1C+var_10]
23: F3 60         STR      R3, [R6,#0xC]
24:             loc_1010786
25: 2B 1B         SUBS     R3, R5, R4
26: 04 2B         CMP      R3, #4
27: 04 D3         BCC      loc_1010796
28: 11 4B         LDR      R3, =__imp_GetCurrentProcessId
29: 1B 68         LDR      R3, [R3]
30: 98 47         BLX      R3
31: 30 51         STR      R0, [R6,R4]
32: 04 34         ADDS     R4, #4
33:             loc_1010796
34: 2B 1B         SUBS     R3, R5, R4
35: 04 2B         CMP      R3, #4
36: 04 D3         BCC      loc_10107A6
37: 0C 4B         LDR      R3, =__imp_GetTickCount
38: 1B 68         LDR      R3, [R3]
39: 98 47         BLX      R3
40: 30 51         STR      R0, [R6,R4]
41: 04 34         ADDS     R4, #4
42:             loc_10107A6
43: 2B 1B         SUBS     R3, R5, R4
44: 08 2B         CMP      R3, #8
45: 09 D3         BCC      loc_10107C0
46: 07 4B         LDR      R3, =__imp_QueryPerformanceCounter
47: 68 46         MOV      R0, SP
48: 1B 68         LDR      R3, [R3]
49: 98 47         BLX      R3
50: 00 9B         LDR      R3, [SP,#0x1C+var_1C]
51: 32 19         ADDS     R2, R6, R4
52: 33 51         STR      R3, [R6,R4]
53: 01 9B         LDR      R3, [SP,#0x1C+var_18]
54: 08 34         ADDS     R4, #8
55: 53 60         STR      R3, [R2,#4]
56:             loc_10107C0
57: 20 46         MOV      R0, R4
58: 04 B0         ADD      SP, SP, #0x10
59: 37 F0 A4 F9   BL       __security_pop_cookie
60: BD E8 70 88   POP.W    {R4-R6,R11,PC}
61:             ; Конец функции mystery10
```

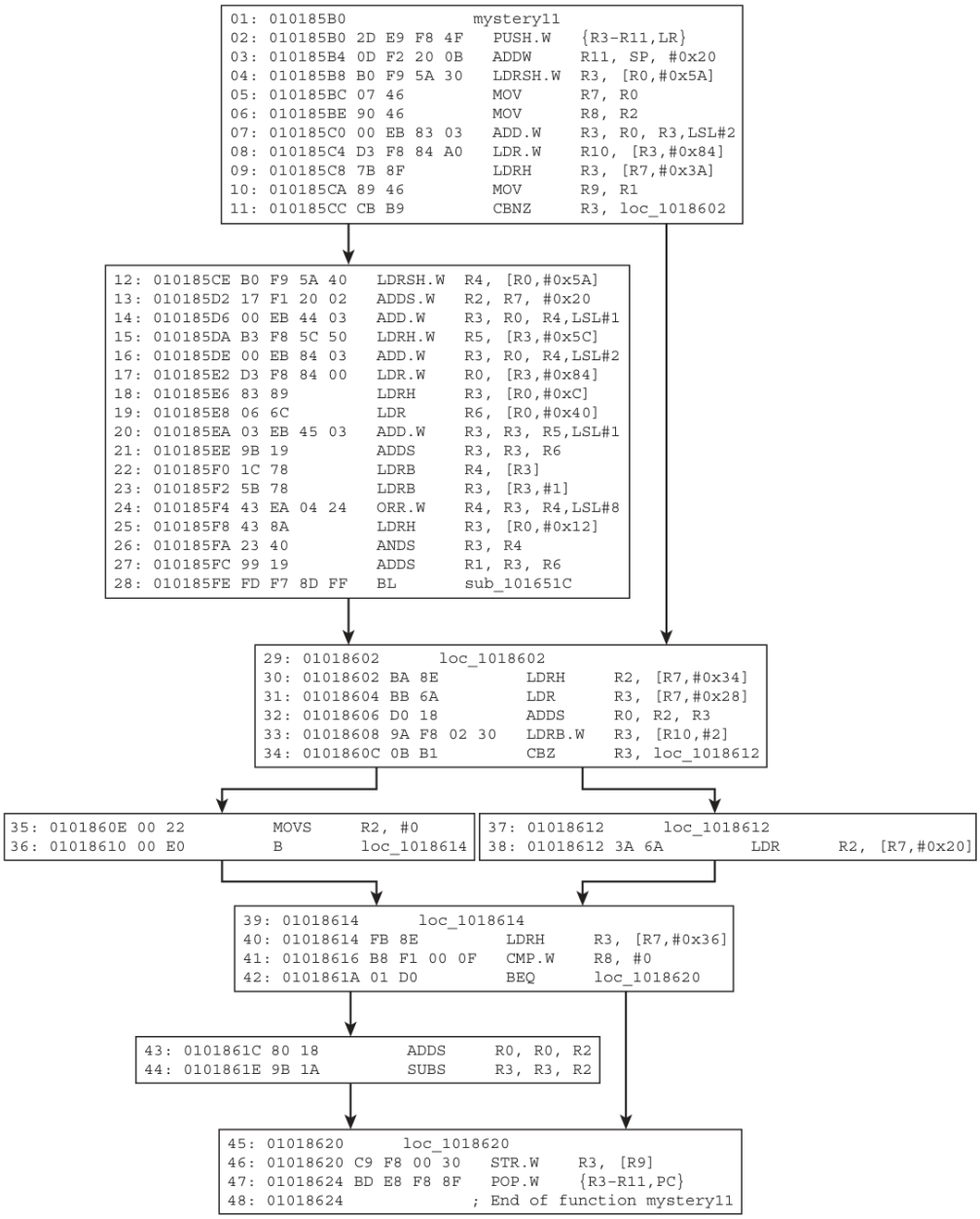

Figure 2-17

Рисунок 2-17. Граф потока управления функции mystery11

```
01: 010185B0             mystery11
02: 010185B0 2D E9 F8 4F   PUSH.W   {R3-R11,LR}
03: 010185B4 0D F2 20 0B   ADDW     R11, SP, #0x20
04: 010185B8 B0 F9 5A 30   LDRSH.W  R3, [R0,#0x5A]
05: 010185BC 07 46         MOV      R7, R0
06: 010185BE 90 46         MOV      R8, R2
07: 010185C0 00 EB 83 03   ADD.W    R3, R0, R3,LSL#2
08: 010185C4 D3 F8 84 A0   LDR.W    R10, [R3,#0x84]
09: 010185C8 7B 8F         LDRH     R3, [R7,#0x3A]
10: 010185CA 89 46         MOV      R9, R1
11: 010185CC CB B9         CBNZ     R3, loc_1018602
12: 010185CE B0 F9 5A 40   LDRSH.W  R4, [R0,#0x5A]
13: 010185D2 17 F1 20 02   ADDS.W   R2, R7, #0x20
14: 010185D6 00 EB 44 03   ADD.W    R3, R0, R4,LSL#1
15: 010185DA B3 F8 5C 50   LDRH.W   R5, [R3,#0x5C]
16: 010185DE 00 EB 84 03   ADD.W    R3, R0, R4,LSL#2
17: 010185E2 D3 F8 84 00   LDR.W    R0, [R3,#0x84]
```

```
18: 010185E6 83 89         LDRH     R3, [R0,#0xC]
19: 010185E8 06 6C         LDR      R6, [R0,#0x40]
20: 010185EA 03 EB 45 03   ADD.W    R3, R3, R5,LSL#1
21: 010185EE 9B 19         ADDS     R3, R3, R6
22: 010185F0 1C 78         LDRB     R4, [R3]
23: 010185F2 5B 78         LDRB     R3, [R3,#1]
24: 010185F4 43 EA 04 24   ORR.W    R4, R3, R4,LSL#8
25: 010185F8 43 8A         LDRH     R3, [R0,#0x12]
26: 010185FA 23 40         ANDS     R3, R4
27: 010185FC 99 19         ADDS     R1, R3, R6
28: 010185FE FD F7 8D FF   BL       sub_101651C
29: 01018602             loc_1018602
30: 01018602 BA 8E         LDRH     R2, [R7,#0x34]
31: 01018604 BB 6A         LDR      R3, [R7,#0x28]
32: 01018606 D0 18         ADDS     R0, R2, R3
33: 01018608 9A F8 02 30   LDRB.W   R3, [R10,#2]
34: 0101860C 0B B1         CBZ      R3, loc_1018612
35: 0101860E 00 22         MOVS     R2, #0
36: 01018610 00 E0         B        loc_1018614
37: 01018612             loc_1018612
38: 01018612 3A 6A         LDR      R2, [R7,#0x20]
39: 01018614             loc_1018614
40: 01018614 FB 8E         LDRH     R3, [R7,#0x36]
41: 01018616 B8 F1 00 0F   CMP.W    R8, #0
42: 0101861A 01 D0         BEQ      loc_1018620
43: 0101861C 80 18         ADDS     R0, R0, R2
44: 0101861E 9B 1A         SUBS     R3, R3, R2
45: 01018620             loc_1018620
46: 01018620 C9 F8 00 30   STR.W    R3, [R9]
47: 01018624 BD E8 F8 8F   POP.W    {R3-R11,PC}
48: 01018624             ; Конец функции mystery11
```

Глава 3. Ядро Windows

В этой главе рассматриваются принципы и методы, необходимые для анализа кода драйверов режима ядра, например руткитов, на платформе Windows. Поскольку драйверы взаимодействуют с ОС через четко определенные интерфейсы, аналитическую задачу можно разложить на следующие общие цели:

• Понять, как реализованы основные компоненты ОС.

• Понять структуру драйвера.

• Понять интерфейсы между пользователем и драйвером, а также между драйвером и ОС, и то, как Windows реализует эти интерфейсы.

• Понять, как определенные программные конструкции драйвера проявляются в двоичной форме.

• Систематически применять знания, полученные на предыдущих этапах, в общем процессе обратного анализа.

Если бы процесс обратного анализа драйверов Windows можно было представить как дискретную задачу, то 90% этой задачи составляло бы понимание работы Windows и 10% - понимание ассемблерного кода. Поэтому эта глава написана как введение в ядро Windows для специалистов по обратному анализу. Она начинается с обсуждения интерфейсов между пользовательским режимом и режимом ядра и их реализации. Затем рассматриваются связанные списки и их применение в Windows. Далее объясняются такие понятия, как потоки, процессы, память и прерывания, а также их использование в ядре и драйверах. После этого мы переходим к архитектуре драйвера режима ядра и программному интерфейсу между драйвером и ядром. В завершение эти понятия применяются к обратному анализу руткита.

Если не указано иное, все примеры в этой главе взяты из Windows 8 RTM.

Основы Windows

Начнем с обсуждения основных понятий ядра Windows, включая фундаментальные структуры данных и объекты ядра, имеющие отношение к программированию драйверов и обратному анализу.

Компоновка памяти

Подобно многим операционным системам, Windows делит виртуальное адресное пространство на две части: пространство ядра и пользовательское пространство. На x86 и ARM верхние 2 Гбайт зарезервированы для ядра, а нижние 2 Гбайт - для пользовательских процессов. Следовательно, виртуальные адреса от 0 до `0x7fffffff` находятся в пользовательском пространстве, а адреса `0x80000000` и выше - в пространстве ядра. На x64 действует тот же принцип, за исключением того, что пользовательское пространство простирается от 0 до `0x000007ff`ffffffff`, а пространство ядра начинается с `0xffff0800`00000000`. На рисунке 3-1 показана общая компоновка для x86 и x64. Пространство памяти ядра в основном одинаково во всех процессах. Однако выполняющиеся процессы имеют доступ только к своему пользовательскому адресному пространству; код режима ядра может обращаться к обоим пространствам. (Некоторые диапазоны адресов ядра, например находящиеся в сеансовом и гиперпространстве, различаются от процесса к процессу.) Этот факт важно помнить, поскольку мы вернемся к нему позже при обсуждении контекста выполнения. Страницы режима ядра и пользовательского режима различаются специальным битом в элементе таблицы страниц.

Когда поток процесса планируется к выполнению, ОС изменяет регистр, зависящий от конкретного процессора, так, чтобы он указывал на каталог страниц именно этого процесса. Благодаря этому все преобразования виртуальных адресов в физические относятся к этому процессу, а не к другим. Именно так ОС может поддерживать несколько процессов, каждый из которых пребывает в иллюзии, что владеет всем адресным пространством пользовательского режима. В архитектурах x86 и x64 регистром базы каталога страниц является CR3; на ARM это регистр базы таблицы трансляции (translation table base register, TTBR).

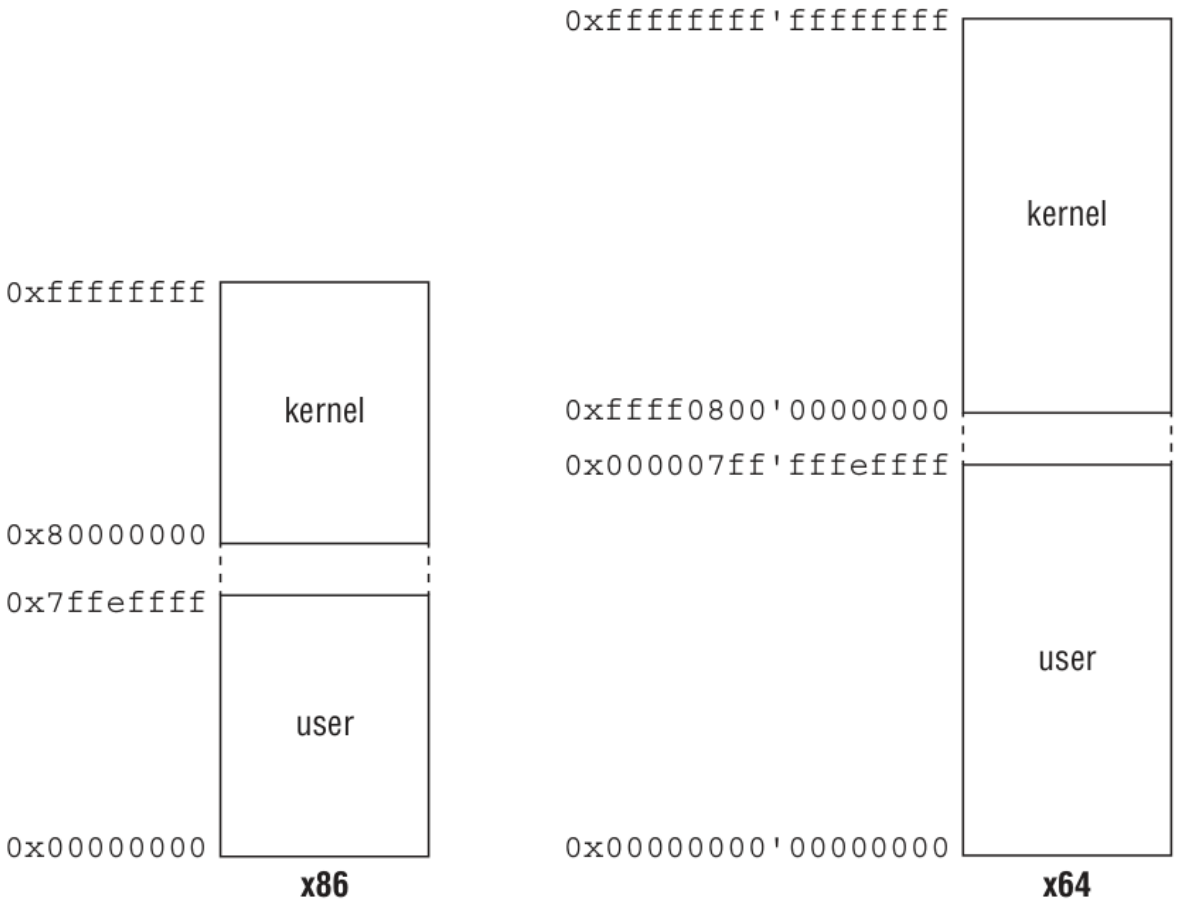

Рисунок 3-1. Общая компоновка адресного пространства на x86 и x64

Примечание. Это стандартное поведение можно изменить, указав параметр `/3GB` в параметрах загрузки. При `/3GB` пользовательское адресное пространство увеличивается до 3 Гбайт, а оставшийся 1 Гбайт отводится ядру.

Диапазоны пользовательских адресов и адресов ядра хранятся в двух символах ядра: `MmSystemRangeStart` (ядро) и `MmHighestUserAddress` (пользователь). Эти символы можно просмотреть с помощью отладчика ядра. Можно заметить, что на x86/ARM между пользовательским пространством и пространством ядра имеется промежуток размером 64 Кбайт. Эта область, обычно называемая областью без доступа (no-access region), существует для того, чтобы ядро случайно не пересекло границу адресов и не повредило память пользовательского режима. На x64 внимательный читатель может заметить, что `0xffff0800`00000000` является неканоническим адресом, а потому операционная система не может его использовать. В действительности этот адрес служит лишь разделителем между пользовательским пространством и пространством ядра. Первый пригодный для использования адрес в пространстве ядра - `0xffff8000`00000000`.

Инициализация процессора

При загрузке ядро выполняет некоторую базовую инициализацию каждого процессора. Большинство подробностей инициализации несущественны для повседневных задач обратного анализа, однако важно знать несколько основных структур.

Область управления процессором (processor control region, PCR) - это отдельная для каждого процессора структура, в которой хранятся критически важные сведения о ЦП и его состояние. Например, на x86 она содержит базовый адрес IDT и текущий IRQL. Внутри PCR находится другая структура данных, называемая блоком управления областью процессора (processor region control block, PRCB). Это отдельная для каждого процессора структура, содержащая сведения о процессоре: тип, модель и скорость ЦП, текущий выполняемый им поток, следующий поток для выполнения, очередь DPC для выполнения и так далее. Как и PCR, эта структура не документирована, однако ее определение все равно можно просмотреть с помощью отладчика ядра:

```
x64 (x86 аналогична)

PCR

   0: kd> dt nt!_KPCR
      +0x000 NtTib            : _NT_TIB
      +0x000 GdtBase          : Ptr64 _KGDTENTRY64
      +0x008 TssBase          : Ptr64 _KTSS64
      +0x010 UserRsp          : Uint8B
      +0x018 Self             : Ptr64 _KPCR
      +0x020 CurrentPrcb      : Ptr64 _KPRCB
   ...
      +0x180 Prcb             : _KPRCB

PRCB

   0: kd> dt nt!_KPRCB
      +0x000 MxCsr            : Uint4B
      +0x004 LegacyNumber     : UChar
      +0x005 ReservedMustBeZero : UChar
      +0x006 InterruptRequest : UChar
      +0x007 IdleHalt         : UChar
      +0x008 CurrentThread    : Ptr64 _KTHREAD
      +0x010 NextThread       : Ptr64 _KTHREAD
      +0x018 IdleThread       : Ptr64 _KTHREAD
   ...
      +0x040 ProcessorState   : _KPROCESSOR_STATE
      +0x5f0 CpuType          : Char
      +0x5f1 CpuID            : Char
      +0x5f2 CpuStep          : Uint2B
      +0x5f2 CpuStepping      : UChar
      +0x5f3 CpuModel         : UChar
      +0x5f4 MHz              : Uint4B
   ...
      +0x2d80 DpcData          : [2] _KDPC_DATA
      +0x2dc0 DpcStack         : Ptr64 Void
      +0x2dc8 MaximumDpcQueueDepth : Int4B
   ...

ARM

PCR

   0: kd> dt nt!_KPCR
      +0x000 NtTib            : _NT_TIB
      +0x000 TibPad0          : [2] Uint4B
      +0x008 Spare1           : Ptr32 Void
      +0x00c Self             : Ptr32 _KPCR
      +0x010 CurrentPrcb      : Ptr32 _KPRCB
   ...
```

PRCB

```
  0: kd> dt nt!_KPCR
     +0x000 NtTib            : _NT_TIB
     +0x000 TibPad0          : [2] Uint4B
     +0x008 Spare1           : Ptr32 Void
     +0x00c Self             : Ptr32 _KPCR
     +0x010 CurrentPrcb      : Ptr32 _KPRCB
  ...
  0: kd> dt nt!_KPRCB
     +0x000 LegacyNumber     : UChar
     +0x001 ReservedMustBeZero : UChar
     +0x002 IdleHalt         : UChar
     +0x004 CurrentThread    : Ptr32 _KTHREAD
     +0x008 NextThread       : Ptr32 _KTHREAD
     +0x00c IdleThread       : Ptr32 _KTHREAD
  ...
     +0x020 ProcessorState   : _KPROCESSOR_STATE
     +0x3c0 ProcessorModel   : Uint2B
     +0x3c2 ProcessorRevision : Uint2B
     +0x3c4 MHz              : Uint4B
  ...
     +0x690 DpcData          : [2] _KDPC_DATA
     +0x6b8 DpcStack         : Ptr32 Void
  ...
     +0x900 InterruptCount   : Uint4B
     +0x904 KernelTime       : Uint4B
     +0x908 UserTime         : Uint4B
     +0x90c DpcTime          : Uint4B
     +0x910 InterruptTime    : Uint4B
  ...
```

PCR текущего процессора всегда доступна из режима ядра через специальные регистры. Она хранится в сегменте FS (x86), сегменте GS (x64) или одном из регистров системного сопроцессора (ARM). Например, ядро Windows экспортирует две процедуры для получения текущих EPROCESS и ETHREAD: `PsGetCurrentProcess` и `PsGetCurrentThread`. Эти процедуры работают, запрашивая PCR/PRCB:

```
PsGetCurrentThread proc near
  mov     rax, gs:188h   ; gs:[0] - это PCR, смещение 0x180 - PRCB,
                         ; смещение 0x8 внутри PRCB - поле CurrentThread
  retn
PsGetCurrentThread endp

PsGetCurrentProcess proc near
  mov     rax, gs:188h    ; получить текущий поток (см. выше)
  mov     rax, [rax+0B8h] ; по смещению 0x70 внутри ETHREAD находится связанный
                          ; процесс (в действительности ETHREAD.ApcState.Process)
  retn
PsGetCurrentProcess endp
```

Системные вызовы

Операционная система управляет аппаратными ресурсами и предоставляет интерфейсы, через которые пользователи могут запрашивать эти ресурсы. Наиболее часто используемый интерфейс - системный вызов. Системный вызов обычно представляет собой функцию в ядре, обслуживающую пользовательские запросы ввода-вывода; он реализуется в ядре, поскольку управлять такими ресурсами может только код с высокими привилегиями. Например, когда текстовый процессор сохраняет файл на диск, ему сначала нужно запросить у ядра дескриптор файла, записать данные в файл, а затем зафиксировать содержимое файла на жестком диске; ОС предоставляет системные вызовы для получения дескриптора файла и записи в него байтов. Хотя эти операции кажутся простыми, для обслуживания запроса системные вызовы должны выполнять в ядре множество важных задач. Например, чтобы получить дескриптор файла, системный вызов должен взаимодействовать с файловой системой (чтобы определить, допустим ли путь), а затем запросить у диспетчера безопасности, достаточно ли у пользователя прав для доступа к файлу; чтобы записать байты в файл, ядру нужно выяснить, на каком томе жесткого диска находится файл, отправить запрос этому тому и упаковать данные в структуру, понятную нижележащему контроллеру жесткого диска. Все эти операции выполняются совершенно прозрачно для пользователя.

Подробности реализации системных вызовов Windows официально не документированы, поэтому их стоит исследовать из интеллектуальных и педагогических соображений. Хотя реализация различается между процессорами, сами понятия остаются неизменными. Сначала мы объясним эти понятия, а затем обсудим подробности реализации на x86, x64 и ARM.

Windows описывает и хранит сведения о системных вызовах с помощью двух структур данных: дескриптора таблицы служб и массива указателей на функции/смещений. Дескриптор таблицы служб - это структура, содержащая метаданные о системных вызовах, поддерживаемых ОС; ее определение официально не документировано, однако многие исследователи восстановили посредством обратного анализа следующие важные поля. (Эти поля также можно выяснить, проанализировав процедуры `KiSystemCall64` или `KiSystemService`.)

```
typedef struct _KSERVICE_TABLE_DESCRIPTOR
{
  PULONG Base; // массив адресов или смещений
  PULONG Count;
  ULONG Limit; // размер массива
  PUCHAR Number;
  ...
} KSERVICE_TABLE_DESCRIPTOR, *PKSERVICE_TABLE_DESCRIPTOR;
```

`Base` - указатель на массив указателей на функции или смещений (в зависимости от процессора); номер системного вызова является индексом в этом массиве. `Limit` - количество элементов в массиве. Ядро хранит два глобальных массива `KSERVICE_DESCRIPTOR_DESCRIPTOR`: `KeServiceDescriptorTable` и `KeServiceDescriptorTableShadow`. Первый содержит собственную таблицу системных вызовов; второй содержит те же данные, а также таблицу системных вызовов для потоков GUI. Кроме того, ядро хранит два глобальных указателя на массивы адресов/смещений: `KiServiceTable` указывает на таблицу системных вызовов, не относящихся к GUI, а `W32pServiceTable` - на таблицу GUI. На рисунке 3-2 показано, как эти структуры данных связаны друг с другом на x86.

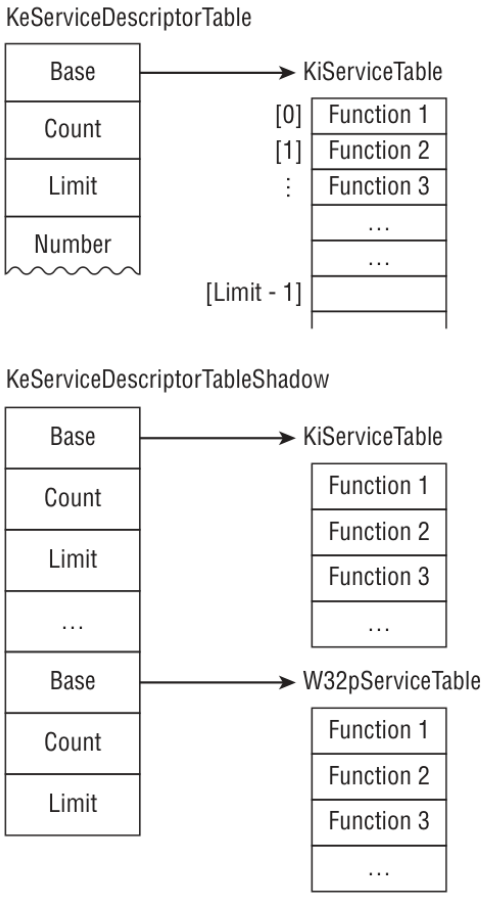

Figure 3-2

Рисунок 3-2. Связи таблиц системных вызовов на x86

На x86 поле `Base` представляет собой массив указателей на функции системных вызовов:

```
0: kd> dps nt!KeServiceDescriptorTable
81472400  813564d0 nt!KiServiceTable    ; база
81472404  00000000
81472408  000001ad
8147240c  81356b88 nt!KiArgumentTable
0: kd> dd nt!KiServiceTable
813564d0  81330901 812cf1e2 81581540 816090af
813564e0  815be478 814b048f 8164e434 8164e3cb
813564f0  812dfa09 814e303f 814a0830 81613a9f
81356500  814e5b65 815b9e3a 815e0c4e 8158ce33
...
0: kd> dps nt!KiServiceTable
813564d0  81330901 nt!NtWorkerFactoryWorkerReady
813564d4  812cf1e2 nt!NtYieldExecution
813564d8  81581540 nt!NtWriteVirtualMemory
813564dc  816090af nt!NtWriteRequestData

813564e0  815be478 nt!NtWriteFileGather
813564e4  814b048f nt!NtWriteFile
```

Однако на x64 и ARM это массив 32-разрядных целых чисел, в которых закодированы смещение системного вызова и количество аргументов, передаваемых через стек. Смещение содержится в старших 20 битах, а количество аргументов в стеке - в младших 4 битах. Для получения действительного адреса системного вызова смещение прибавляется к базе `KiServiceTable`. Например:

```
0: kd> dps nt!KeServiceDescriptorTable
fffff803`955cd900  fffff803`952ed200 nt!KiServiceTable   ; база
fffff803`955cd908  00000000`00000000
fffff803`955cd910  00000000`000001ad
fffff803`955cd918  fffff803`952edf6c nt!KiArgumentTable
0: kd> u ntdll!NtCreateFile
ntdll!NtCreateFile:
000007f8`34f23130 mov     r10,rcx
000007f8`34f23133 mov     eax,53h ; номер системного вызова
000007f8`34f23138 syscall
...
0: kd> x nt!KiServiceTable
fffff803`952ed200 nt!KiServiceTable (<no parameter info>)
0: kd> dd nt!KiServiceTable + (0x53*4) L1
fffff803`952ed34c  03ea2c07       ; закодированные смещение и число аргументов
0: kd> u nt!KiServiceTable + (0x03ea2c07>>4) ; получить смещение и прибавить его
Base
nt!NtCreateFile:
fffff803`956d74c0 sub     rsp,88h
fffff803`956d74c7 xor     eax,eax
fffff803`956d74c9 mov     qword ptr [rsp+78h],rax
fffff803`956d74ce mov     dword ptr [rsp+70h],20h
0: kd> ? 0x03ea2c07 & 0xf         ; число аргументов
Evaluate expression: 7 = 00000000`00000007
; NtCreateFile принимает 11 аргументов. Первые 4 передаются через регистры,
; а последние 7 - через стек
```

Как показано выше, каждый системный вызов идентифицируется номером, являющимся индексом в `KiServiceTable` или `W32pServiceTable`. На самом низком уровне API пользовательского режима раскладываются на один или несколько системных вызовов.

В общих чертах именно так работают системные вызовы в Windows. Подробности реализации различаются в зависимости от архитектуры процессора и платформы. Системные вызовы обычно реализуются посредством программных прерываний или команд, специфичных для конкретной архитектуры; подробности рассматриваются в следующих разделах.

Сбои, ловушки и прерывания

В рамках подготовки к следующим разделам нам необходимо ввести несколько основных терминов, чтобы объяснить, как периферийные устройства и программное обеспечение взаимодействуют с процессором. В современных вычислительных системах процессор обычно соединен с периферийными устройствами через шину данных, например PCI Express, FireWire или USB. Когда устройству требуется внимание процессора, оно вызывает прерывание, которое заставляет процессор приостановить текущую работу и обработать запрос устройства. Как процессор узнает, каким образом обработать запрос? На самом высоком уровне прерывание можно представить связанным с некоторым номером, который затем используется как индекс в массиве указателей на функции. Получив прерывание, процессор выполняет функцию по индексу, связанному с запросом, а затем возобновляет выполнение с того места, где оно находилось до возникновения прерывания. Такие прерывания называются аппаратными, поскольку их порождают аппаратные устройства. По своей природе они асинхронны.

Во время выполнения команды процессор может столкнуться с исключениями. Например, команда вызывает ошибку деления на ноль, обращается по недопустимому адресу или инициирует переход

между уровнями привилегий. Для целей нашего обсуждения исключения можно разделить на две категории: сбои и ловушки. Сбой - это исправимое исключение. Например, когда процессор выполняет команду, обращающуюся к допустимому адресу памяти, но данных нет в основной памяти (они были выгружены), возникает исключение страничного сбоя. Процессор обрабатывает его, сохраняя текущее состояние выполнения, вызывая обработчик страничного сбоя для исправления исключения (путем подкачки данных) и повторно выполняя ту же команду (которая больше не должна вызывать страничный сбой). Ловушка - это исключение, вызванное выполнением специальных видов команд. Например, на x64 команда SYSCALL заставляет процессор начать выполнение по адресу, указанному MSR; после завершения обработчика выполнение возобновляется с команды, непосредственно следующей за SYSCALL. Таким образом, основное различие между сбоем и ловушкой состоит в том, откуда возобновляется выполнение. Системные вызовы обычно реализуются посредством специальных исключений или команд-ловушек.

Прерывания

Архитектура Intel определяет таблицу дескрипторов прерываний (interrupt descriptor table, IDT) из 256 элементов; каждый элемент представляет собой структуру со сведениями, определяющими обработчик прерывания. Базовый адрес IDT хранится в специальном регистре IDTR. Каждому прерыванию соответствует индекс в этой таблице. Архитектурой зарезервированы предопределенные прерывания. Например, `0x0` предназначено для исключения деления, `0x3` - для программной точки останова, а `0xe` - для страничных сбоев. Прерывания 32-255 определяются пользователем.

На x86 каждый элемент таблицы IDT является 8-байтовой структурой следующего вида:

```
1: kd> dt nt!_KIDTENTRY
   +0x000 Offset           : Uint2B
   +0x002 Selector         : Uint2B
   +0x004 Access           : Uint2B
   +0x006 ExtendedOffset   : Uint2B
```

(На x64 структура элемента IDT в основном такая же, за исключением того, что адрес обработчика прерывания разделен между тремя полями. Увидеть ее можно, выведя структуру `nt!_KIDTENTRY64`. Также обратите внимание, что ширина IDTR составляет 48 бит и он разделен на две части: базовый адрес IDT и предел. WinDBG отображает только базовый адрес.)

Адрес обработчика прерывания разделен между полями `Offset` и `ExtendedOffset`. Ниже приведен пример декодирования IDT и дизассемблирования обработчика прерывания деления на ноль (`0x0`):

```
1: kd> r @idtr
idtr=8b409d50
1: kd> dt nt!_KIDTENTRY 8b409d50
   +0x000 Offset           : 0xa284
   +0x002 Selector         : 8
   +0x004 Access           : 0x8e00
   +0x006 ExtendedOffset   : 0x813c
1: kd> u 0x813ca284
nt!KiTrap00:
813ca284 push    0
813ca286 mov     word ptr [esp+2],0
813ca28d push    ebp
813ca28e push    ebx
```

На рисунке 3-3 показана IDT на x86.

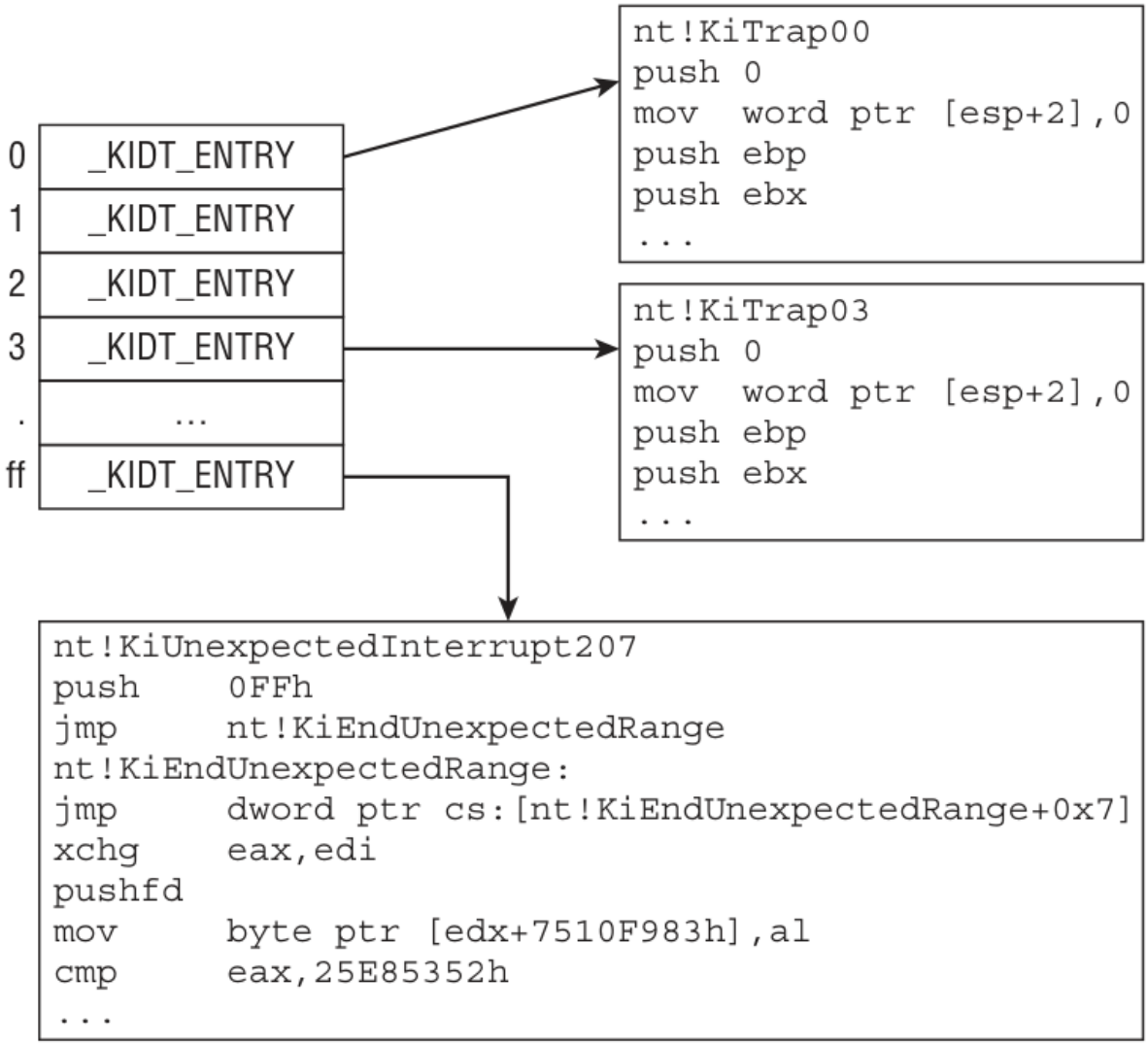

Figure 3-3

Рисунок 3-3. Таблица дескрипторов прерываний на x86

На процессорах, предшествующих Pentium 2, Windows использует прерывание `0x2e` для реализации системных вызовов. Программы пользовательского режима вызывают API в `kernel32.dll` (или `kernelbase.dll`), которые в конечном итоге сводятся к коротким заглушкам в `ntdll.dll`, инициирующим прерывание `0x2e`. В качестве иллюстрации рассмотрим следующий фрагмент процедуры API `kernelbase!CreateFileW` в Windows 7:

```
[внутри kernelbase!CreateFileW]
...
.text:0DCE9C87 mov ecx, [ebp+dwFlagsAndAttributes]
.text:0DCE9C8A push [ebp+lpSecurityAttributes]
.text:0DCE9C8D mov eax, [ebp+dwDesiredAccess]
.text:0DCE9C90 push [ebp+lpFileName]
.text:0DCE9C93 mov esi, ds:__imp__NtCreateFile@44
.text:0DCE9C99 push [ebp+var_4]
.text:0DCE9C9C and ecx, 7FA7h
.text:0DCE9CA2 push [ebp+dwShareMode]
.text:0DCE9CA5 mov [ebp+dwFlagsAndAttributes], ecx
.text:0DCE9CA8 push ecx
.text:0DCE9CA9 push ebx
.text:0DCE9CAA lea ecx, [ebp+var_20]
.text:0DCE9CAD push ecx
.text:0DCE9CAE or eax, 100080h
.text:0DCE9CB3 lea ecx, [ebp+var_64]
```

```
.text:0DCE9CB6 push ecx
.text:0DCE9CB7 push eax
.text:0DCE9CB8 mov [ebp+dwDesiredAccess], eax
.text:0DCE9CBB lea eax, [ebp+var_8]
.text:0DCE9CBE push eax
.text:0DCE9CBF call esi ; NtCreateFile(...)
```

Эта процедура выполняет некоторую предварительную проверку (здесь не показанную), а затем вызывает `ntdll!NtCreateFile`. Его реализация выглядит следующим образом:

```
[ntdll!NtCreateFile]
.text:77F04A10 _NtCreateFile@44 proc near
.text:77F04A10 mov eax, 42h ; номер системного вызова
.text:77F04A15 mov edx, 7FFE0300h ; KUSER_SHARED_DATA.SystemCall
; символом для 0x7ffe0300 является SharedUserData!SystemCallStub
.text:77F04A1A call dword ptr [edx] ; вызвать обработчик
.text:77F04A1C retn 2Ch ; вернуться к вызывающей стороне
.text:77F04A1C _NtCreateFile@44 endp
```

`NtCreateFile` присваивает EAX значение `0x42`, поскольку это номер системного вызова `NtCreateFile` в ядре. Затем она читает указатель по адресу `0x7ffe0300` и вызывает его. Чем примечателен адрес `0x7ffe0300`? На всех архитектурах существует отдельная для каждого процесса структура `KUSER_SHARED_DATA`, которая всегда отображается по адресу `0x7ffe0000`. Она содержит некоторые общие сведения о системе и поле `SystemCall`:

```
0:000> dt ntdll!_KUSER_SHARED_DATA
   +0x000 TickCountLowDeprecated : Uint4B
   +0x004 TickCountMultiplier : Uint4B
   +0x008 InterruptTime : _KSYSTEM_TIME
   +0x014 SystemTime : _KSYSTEM_TIME

   +0x020 TimeZoneBias : _KSYSTEM_TIME
   ...
   +0x2f8 TestRetInstruction : Uint8B
   +0x300 SystemCall : Uint4B ; обработчик системного вызова
   +0x304 SystemCallReturn : Uint4B
   ...
```

При дизассемблировании заглушки системного вызова можно увидеть следующее:

```
0:000> u poi(SharedUserData!SystemCallStub)
ntdll!KiIntSystemCall:
76e46500 lea edx,[esp+8]
76e46504 int 2Eh
76e46506 ret
76e46507 nop
```

Вывод элемента IDT с индексом `0x2e` показывает, что диспетчером системных вызовов является `KiSystemService`:

```
0: kd> !idt 0x2e
Dumping IDT: ...
2e: 8284b22e nt!KiSystemService
0: kd> u nt!KiSystemService
nt!KiSystemService:
8284b22e push 0
8284b230 push ebp
8284b231 push ebx
8284b232 push esi
8284b233 push edi
8284b234 push fs
8284b236 mov ebx,30h
...
```

Подробности работы диспетчера системных вызовов рассматриваются в следующем разделе.

Ловушки

В предыдущем разделе объяснялось, как системные вызовы реализуются с помощью встроенного механизма обработки прерываний. В этом разделе объясняется их реализация посредством команд-ловушек на x64, x86 и ARM.

Начиная с реализации на x64, рассмотрим заглушку системного вызова `ntdll!NtCreateFile`:

```
01: .text:00000001800030F0   public ZwCreateFile
02: .text:00000001800030F0 ZwCreateFile proc near
03: .text:00000001800030F0   mov     r10, rcx
04: .text:00000001800030F3   mov     eax, 53h
05: .text:00000001800030F8   syscall
06: .text:00000001800030FA   retn
07: .text:00000001800030FA ZwCreateFile endp
```

Строка 3 сохраняет первый аргумент в R10; это необходимо потому, что согласно семантике SYSCALL адрес возврата (строка 6) должен храниться в RCX. Строка 4 сохраняет номер системного вызова в EAX; после перехода SYSCALL в режим ядра это значение будет использовано как индекс в массиве `KiServiceTable`. В строке 5 выполняется SYSCALL, осуществляющая переход в режим ядра. Как это происходит? В документации SYSCALL указано, что в RIP будет загружено значение, определенное MSR `IA32_LSTAR` (`0xc0000082`), что можно увидеть в отладчике:

```
1: kd> rdmsr 0xC0000082
msr[c0000082] = fffff800`89e96dc0
1: kd> u fffff800`89e96dc0
nt!KiSystemCall64:
fffff800`89e96dc0 swapgs
fffff800`89e96dc3 mov     qword ptr gs:[10h],rsp
fffff800`89e96dcc mov     rsp,qword ptr gs:[1A8h]
fffff800`89e96dd5 push    2Bh
fffff800`89e96dd7 push    qword ptr gs:[10h]
fffff800`89e96ddf push    r11
```

Этот вывод отладчика ядра указывает, что SYSCALL всегда в конечном итоге начинает выполнение `KiSystemCall64` в ядре. В действительности `KiSystemCall64` - главный диспетчер системных вызовов в 64-разрядной Windows. Windows присваивает MSR IA32 LSTAR адрес `KiSystemCall64` на раннем этапе инициализации процессора (см. `KiInitializeBootStructures`). В первую очередь эта процедура отвечает за сохранение контекста пользовательского режима, подготовку стека ядра, копирование аргументов пользовательского режима в стек ядра, определение системного вызова в `KiServiceTable` (или `W32pServiceTable`) по индексу, переданному в EAX, вызов системного вызова и возврат в пользовательский режим. Как диспетчер системных вызовов узнает, куда возвращаться в пользовательском режиме? Напомним, что SYSCALL сохраняет адрес возврата в RCX. После завершения и возврата системного вызова диспетчер системных вызовов применяет команду SYSRET, которая присваивает RIP значение RCX, тем самым возвращая выполнение в пользовательский режим.

Хотя KiSystemCall64 поддерживает множество функций (профилирование системных вызовов, планирование в пользовательском режиме, отладку и т. д.), ее основная обязанность - диспетчеризация запросов системных вызовов. В предыдущем разделе мы утверждали, что каждое значение в массиве KiServiceTable кодирует смещение системного вызова и количество аргументов, передаваемых через стек. Это можно увидеть в следующем фрагменте кода KiSystemCall64:

```
01: KiSystemCall64 proc near
02:
03: var_110= byte ptr -110h
04:
05:   swapgs
06:   mov     gs:10h, rsp    ; KPCR->UserRsp
07:   mov     rsp, gs:1A8h   ; KPCR->KPRCB->RspBase
08:                          ; подготовить новый стек ядра

09:   push    2Bh
10:   push    qword ptr gs:10h ; KPCR->UserRsp
11:   push    r11
12:
13:   sti                      ; разрешить прерывания
14:   mov     [rbx+88h], rcx   ; KTHREAD->FirstArgument
15:   mov     [rbx+80h], eax   ; KTHREAD->SystemCallNumber
16: KiSystemServiceStart proc near
17:   mov     [rbx+90h], rsp   ; KTHREAD->TrapFrame
18:   mov     edi, eax         ; eax = номер системного вызова
19:   shr     edi, 7           ; определить таблицу системных вызовов
20:   and     edi, 20h
21:   and     eax, 0FFFh       ; индекс в таблице (вспомните кодирование
                              ; 64-разрядного системного вызова)
22: KiSystemServiceRepeat proc near
23:   lea     r10, KeServiceDescriptorTable
24:   lea     r11, KeServiceDescriptorTableShadow
25:   test    dword ptr [rbx+78h], 40h ; определить, является ли поток потоком GUI
26:   cmovnz  r10, r11         ; какую таблицу использовать?
27:   cmp     eax, [rdi+r10+10h] ; находится ли таблица системных вызовов в пределах
                                ; Limit этой таблицы?
28:                             ; то есть KSERVICE_TABLE_DESCRIPTOR.Limit
29:   jnb     case_invalidcallnumber
30:   mov     r10, [rdi+r10]     ; выбрать правильную таблицу
31:   movsxd  r11, dword ptr [r10+rax*4] ; получить смещение системного вызова
32:   mov     rax, r11
33:   sar     r11, 4
34:   add     r10, r11   ; прибавить к базе таблицы и получить VA системного вызова
35:   cmp     edi, 20h   ; edi определяет таблицу. Здесь он используется, чтобы
                         ; определить, относится ли она к GUI
36:   jnz     short case_nonguirequest
37:   mov     r11, [rbx+0F0h]
38:
39: KiSystemServiceCopyEnd proc near
40:   test    cs:dword_140356088, 40h
41:   jnz     case_loggingenabled
42:   call    r10        ; вызвать системный вызов
```

Пошаговое изучение KiSystemCall64 может оказаться познавательным занятием и оставляется в качестве упражнения.

На x86 Windows реализует системные вызовы с помощью команды SYSENTER. Механизм похож на механизм SYSCALL на процессорах x64. Прежде чем переходить к реализации, рассмотрим заглушку системного вызова `ntdll!NtQueryInformationProcess`:

```
01: _ZwQueryInformationProcess@20 proc near
02:   mov     eax, 0B0h          ; номер системного вызова
03:   call    sub_6A214FCD       ; заглушка
04:   retn    14h    ; очистить стек и вернуться. NtQueryInformationProcess
                     ; принимает 5 параметров, и они передаются через стек
                     ; SYSENTER вернется сюда (см. следующий пример)

06: _ZwQueryInformationProcess@20 endp
07:
08: sub_6A214FCD proc near
09:   mov     edx, esp
10:   sysenter
11:   retn
12: sub_6A214FCD endp
```

`ntdll!NtCreateFile` устанавливает номер системного вызова и вызывает другую процедуру, которая сохраняет указатель стека в EDX, после чего выполняется команда SYSENTER. В документации Intel указано, что SYSENTER присваивает EIP значение, хранящееся в MSR `0x176`:

```
0: kd> rdmsr 0x176
msr[176] = 00000000`80f7d1d0
0: kd> u 00000000`80f7d1d0
nt!KiFastCallEntry:
80f7d1d0 mov     ecx,23h
80f7d1d5 push    30h
80f7d1d7 pop     fs
80f7d1d9 mov     ds,cx
80f7d1db mov     es,cx
80f7d1dd mov     ecx,dword ptr fs:[40h]
```

Вывод отладчика показывает, что при выполнении команды SYSENTER происходит переход в режим ядра и начинается выполнение `KiFastCallEntry`. `KiFastCallEntry` - главный диспетчер системных вызовов в 32-разрядной Windows при использовании SYSENTER (рассматривайте его как аналог `KiSystemCall64` на x64). Одна необычная особенность SYSENTER заключается в том, что, в отличие от SYSCALL, она не сохраняет адрес возврата в регистре. Как после завершения системного вызова ядро узнает, куда вернуться? Ответ состоит из двух частей. Снова воспользуемся `NtQueryInformationProcess` в качестве примера: до вызова SYSENTER для входа в режим ядра последовательность вызовов сначала выглядит так:

```
    kernel32!GetLogicalDrives ->
ntdll!NtQueryInformationProcess ->
                              stub -> SYSENTER
```

Это означает, что до выполнения SYSENTER адрес возврата уже подготовлен в стеке. Непосредственно перед SYSENTER `KiFastSystemCall` сохраняет указатель стека в EDX. Во-вторых, после SYSENTER код переходит к `KiFastCallEntry`, которая сохраняет этот указатель стека. По завершении системного вызова диспетчер системных вызовов выполняет команду SYSEXIT. По определению SYSEXIT присваивает EIP значение EDX, а ESP - значение ECX; на практике перед входом в ядро ядро присваивает EDX адрес `ntdll!KiSystemCallRet`, а ECX - указатель стека. Это можно увидеть в действии, установив точку останова на команде SYSEXIT внутри `KiSystemCallExit2`, а затем просмотрев оттуда стек:

```
1: kd> r
eax=00000000 ebx=00000000 ecx=029af304 edx=77586954 esi=029af3c0 edi=029afa04
eip=815d0458 esp=a08f7c8c ebp=029af3a8 iopl=0         nv up ei ng nz na pe cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000287

nt!KiSystemCallExit2+0x18:
815d0458 sysexit
1: kd> dps @ecx L5   # SYSEXIT присвоит ESP значение ECX (обратите внимание на адрес возврата)
029af304  77584fca ntdll!NtQueryInformationProcess+0xa   # адрес возврата
029af308  775a9628 ntdll!RtlDispatchException+0x7c
029af30c  ffffffff
029af310  00000022
029af314  029af348
1: kd> u 77584fca
ntdll!NtQueryInformationProcess+0xa:
77584fca ret     14h            # это строка 4 предыдущего фрагмента
1: kd> u @edx                   # SYSEXIT присвоит EIP значение EDX
ntdll!KiFastSystemCallRet:
77586954 ret                    # возврат к 77584fca
```

После выполнения `KiFastSystemCallRet` (содержащей всего одну команду: RET) происходит возврат в `NtQueryInformationProcess`.

Полезно сравнить реализации SYSENTER в Windows 7 и 8. Вам будет предложено сделать это в одном из упражнений.

Windows на ARM реализует системные вызовы с помощью команды SVC. В более старой документации SVC может называться SWI, но это один и тот же код операции. Напомним, что у ARM нет IDT, как у x86/x64, однако ее таблица векторов исключений обеспечивает схожую функциональность:

```
.text:004D0E00 KiArmExceptionVectors
.text:004D0E00   LDR.W           PC, =0xFFFFFFFF
.text:004D0E04   LDR.W           PC, =(KiUndefinedInstructionException+1)
.text:004D0E08   LDR.W           PC, =(KiSWIException+1)
.text:004D0E0C   LDR.W           PC, =(KiPrefetchAbortException+1)
.text:004D0E10   LDR.W           PC, =(KiDataAbortException+1)
.text:004D0E14   LDR.W           PC, =0xFFFFFFFF
.text:004D0E18   LDR.W           PC, =(KiInterruptException+1)
.text:004D0E1C   LDR.W           PC, =(KiFIQException+1)
```

При каждом выполнении команды SVC процессор переключается в режим супервизора и вызывает `KiSWIException` для обработки исключения. Эту функцию можно рассматривать как ARM-аналог `KiSystemCall64` на x64. Чтобы понять весь процесс системного вызова на ARM, снова рассмотрим функцию пользовательского режима `ntdll!NtQueryInformationProcess`:

```
01: NtQueryInformationProcess
02:   MOV.W           R12, #0x17           ; NtQueryInformationProcess
03:   SVC             1
04:   BX              LR
```

Сначала номер системного вызова помещается в R12, после чего выполняется SVC. При выполнении SVC происходит переход в обработчик `KiSWIException`:

```
01: KiSWIException
02: trapframe= -0x1A0
03:   SUB             SP, SP, #0x1A0
04:   STRD.W          R0, R1, [SP,#0x1A0+trapframe._R0]

05:   STRD.W          R2, R3, [SP,#0x1A0+trapframe._R2]
06:   STR.W           R12, [SP,#0x1A0+trapframe._R12]
07:   STR.W           R11, [SP,#0x1A0+trapframe._R11]
08:   ORR.W           LR, LR, #1
09:   MOVS            R0, #0x30
10:   MRC             p15, 0, R12,c13,c0, 3 ; получить текущий поток
11:   STRD.W          LR, R0, [SP,#0x1A0+trapframe._Pc] ; LR - адрес возврата
                                                        ; после команды SVC. Он
                                                        ; сохраняется здесь, чтобы
                                                        ; система знала, куда
                                                        ; вернуться после завершения
                                                        ; системного вызова
12:   LDRB.W          R1, [R12,#_ETHREAD.Tcb.Header.___u0.__s3.DebugActive]
13:   MOVS            R3, #2
14:   STR             R3, [SP,#0x1A0+trapframe.ExceptionActive]
15:   ADD.W           R11, SP, #0x1A0+trapframe._R11
16:   CMP             R1, #0
17:   BNE             case_DebugMode
18: loc_4D00D0
19:   MRC             p15, 0, R0,c1,c0, 2
20:   MOVS            R1, #0
21:   TST.W           R0, #0xF00000
22:   BEQ             loc_4D00F2
23:   ADD             R3, SP, #0x1A0+var_C8
24:   VMRS            R2, FPSCR
25:   ADD             R1, SP, #sizeof(_KTRAP_FRAME)
26:   STR             R2, [SP,#0x1A0+var_114]
27:   VSTMIA          R3, {D8-D15}
28:   BIC.W           R2, R2, #0x370000
29:   VMSR            FPSCR, R2
30:
31: loc_4D00F2
32:   STR             R1, [SP,#0x1A0+trapframe.VfpState]
33:   LDR             R0, [SP,#0x1A0+trapframe._R12] ; получить сохраненный номер
                                                     ; системного вызова из строки 6
34:   LDR             R1, [SP,#0x1A0+trapframe._R0]
35:   MOV             R2, SP
36:   CPS.W           #0x1F
37:   STR.W           SP, [R2,#0x1A0+trapframe._Sp]
38:   STR.W           LR, [R2,#0x1A0+trapframe._Lr]
39:   CPSIE.W         I, #0x13
40:   STRD.W          R0, R1, [R12,#_ETHREAD.Tcb.SystemCallNumber]
                                                    ; записать номер системного
                                                    ; вызова в поток
41:   MRC             p15, 0, R0,c13,c0, 4
42:   BFC.W           R0, #0, #0xC
43:   LDR.W           R1, [R0,#0x594]
44:   MOV             R2, #0x5CF300
45:   MOV             R12, #KiTrapFrameLog
46:   CMP             R1, #4

47:   BCS             loc_4D0178
48:
49:
50: loc_4D0178
51:   MRC             p15, 0, R12,c13,c0, 3
52:   LDR.W           R0, [R12,#_ETHREAD.Tcb.SystemCallNumber]
53:   BL              KiSystemService        ; диспетчеризовать системный вызов
54:   B               KiSystemServiceExit    ; вернуться в пользовательский режим
```

Эта функция выполняет множество действий, но главное заключается в том, что она создает кадр ловушки (`nt!_KTRAP_FRAME`) для сохранения некоторых регистров, сохраняет адрес возврата в пользовательский режим (SVC автоматически помещает адрес возврата в LR), сохраняет номер системного вызова в объекте текущего потока и диспетчеризует системный вызов (с помощью того же механизма, что и на x64). Возврат в пользовательский режим выполняется через `KiSystemServiceExit`:

```
01: KiSystemServiceExit
02: ...
03:   BIC.W           R0, R0, #1
04:   MOV             R3, SP
05:   ADD             SP, SP, #0x1A0
06:   CPS.W           #0x1F
07:   LDR.W           SP, [R3,#_KTRAP_FRAME._Sp]
08:   LDRD.W          LR, R11, [R3,#_KTRAP_FRAME._Lr]
09:   CPS.W           #0x12
10:   STRD.W          R0, R1, [SP]
11:   LDR             R0, [R3,#_KTRAP_FRAME._R0]
12:   MOVS            R1, #0
13:   MOVS            R2, #0
14:   MOVS            R3, #0
15:   MOV             R12, R1
16:   RFEFD.W         SP       ; вернуться в пользовательский режим
```

Уровень запроса прерывания

Ядро Windows использует абстрактное понятие, называемое уровнем запроса прерывания (interrupt request level, IRQL), для управления прерываемостью системы. Прерывания можно разделить на две общие категории: программные и аппаратные. Программные прерывания - синхронные события, инициируемые условиями в выполняющемся коде (деление на 0, выполнение команды INT, страничный сбой и т. д.); аппаратные прерывания - асинхронные события, инициируемые устройствами, подключенными к ЦП. Аппаратные прерывания асинхронны, поскольку могут возникнуть в любой момент; обычно они используются для сообщения процессору об операциях ввода-вывода. Подробности работы аппаратных прерываний зависят от оборудования, поэтому их скрывает уровень аппаратных абстракций (hardware abstraction layer, HAL) Windows.

Если говорить конкретно, IRQL - это просто число (определяемое типом `KIRQL`, который в действительности является `UCHAR`), назначенное процессору. Windows связывает IRQL с прерыванием и определяет порядок его обработки. Точное число, соответствующее каждому IRQL, может различаться от платформы к платформе, поэтому мы будем ссылаться на них только по именам. Общее правило состоит в том, что прерывания на IRQL X маскируют все прерывания с уровнем ниже X. После обработки прерывания ядро понижает IRQL, чтобы иметь возможность выполнять другие задачи. Поскольку IRQL является отдельным для каждого процессора значением, несколько процессоров могут одновременно работать на разных IRQL.

Существует несколько различных IRQL, но важнее всего запомнить следующие:

• `PASSIVE_LEVEL` (0) - самый низкий IRQL в системе. На этом IRQL выполняется весь код пользовательского режима и большая часть кода ядра.

• `APC_LEVEL` (1) - IRQL, на котором выполняются асинхронные вызовы процедур (asynchronous procedure calls, APC). (См. раздел «Асинхронные вызовы процедур».)

• `DISPATCH_LEVEL` (2) - самый высокий программный IRQL в системе. На этом IRQL работают диспетчер потоков и отложенные вызовы процедур (deferred procedure calls, DPC). (См. раздел «Отложенные вызовы процедур».) Код на этом IRQL не может ожидать.

> **Примечание.** IRQL выше `DISPATCH_LEVEL` обычно связаны с настоящими аппаратными прерываниями или чрезвычайно низкоуровневыми механизмами синхронизации. Например, `IPI_LEVEL` используется для обмена данными между процессорами.

Хотя IRQL может показаться свойством планирования потоков, это не так. IRQL - свойство процессора, тогда как приоритет потока - свойство потока.

Поскольку IRQL является программной абстракцией приоритета прерываний, нижележащая реализация напрямую связана с аппаратным обеспечением. Например, на x86/x64 локальный контроллер прерываний (local interrupt controller, LAPIC) в процессоре имеет программируемый регистр приоритета задач (task priority register, TPR) и доступный только для чтения регистр приоритета процессора (processor priority register, PPR). TPR определяет приоритет прерывания; PPR представляет текущий приоритет прерывания. Процессор доставляет только те прерывания, приоритет которых выше PPR. На практике, когда Windows требуется изменить приоритет прерывания, она вызывает функции ядра `KeRaiseIrql`/`KeLowerIrql`, которые программируют TPR локального APIC. Это можно увидеть в определении для x64 (на x64 CR8 - теневой регистр, обеспечивающий быстрый доступ к TPR LAPIC; системы x86 для задания TPR должны программировать LAPIC):

`KeRaiseIrql`

```
01: KzRaiseIrql proc near
02:   mov     rax, cr8
03:   movzx   ecx, cl
04:   mov     cr8, rcx
05:   retn
06: KzRaiseIrql endp
```

`KeLowerIrql`

```
01: KzLowerIrql proc near
02:   movzx   eax, cl
03:   mov     cr8, rax
04:   retn
05: KzLowerIrql endp
```

Изложенные выше понятия объясняют, почему код, выполняющийся на высоком IRQL, не может быть вытеснен кодом с более низким IRQL.

Пул памяти

Подобно приложениям пользовательского режима, код режима ядра может выделять память во время выполнения. Ее общее название - память пула; ее можно рассматривать как аналог кучи в пользовательском режиме. Память пула обычно делится на два типа: выгружаемый пул и невыгружаемый пул. Память выгружаемого пула может быть в любой момент выгружена диспетчером памяти. Когда код режима ядра обращается к выгруженному буферу, он инициирует исключение страничного сбоя, из-за которого диспетчер памяти подкачивает этот буфер с диска. Память невыгружаемого пула никогда не может быть выгружена; иными словами, обращение к такой памяти никогда не вызывает страничный сбой.

Это различие важно, поскольку оно имеет последствия для кода, выполняющегося на высоких IRQL. Предположим, поток ядра в данный момент выполняется на `DISPATCH_LEVEL` и обращается к памяти, которая была выгружена и должна быть обработана обработчиком страничного сбоя; поскольку обработчику страничного сбоя (см. `MmAccessFault`) требуется отправить запрос для загрузки страницы с диска, а диспетчер потоков работает на `DISPATCH_LEVEL`, обработчик не может устранить исключение, в результате чего возникает bugcheck. Это одна из причин, по которым код,

выполняющийся на DISPATCH_LEVEL, должен находиться только в памяти невыгружаемого пула и обращаться только к ней.

Память пула выделяется и освобождается семействами функций ExAllocatePool* и ExFreePool*. По умолчанию память невыгружаемого пула (тип NonPagedPool) отображается с разрешениями на чтение, запись и выполнение на x86/x64, но без разрешения на выполнение на ARM; в Windows 8 можно запросить неисполняемую память невыгружаемого пула, указав тип пула NonPagedPoolNX. Память выгружаемого пула отображается с разрешениями на чтение, запись и выполнение на x86, но без разрешения на выполнение на x64/ARM.

Списки дескрипторов памяти

Список дескрипторов памяти (memory descriptor list, MDL) - структура данных, используемая для описания набора физических страниц, отображаемых виртуальным адресом. Каждый элемент MDL описывает один непрерывный буфер, а несколько элементов можно связать друг с другом. После построения MDL для существующего буфера физические страницы можно закрепить в памяти (то есть они не будут использованы повторно) и отобразить по другому виртуальному адресу.

Чтобы MDL были полезны, их необходимо инициализировать, проверить и закрепить, а затем отобразить. Для лучшего понимания этого понятия рассмотрим несколько практических применений MDL.

Предположим, драйверу нужно отобразить некоторую память из пространства ядра в адресное пространство пользовательского режима процесса или наоборот. Для этого он сначала инициализирует MDL, описывающий буфер памяти (IoAllocateMdl), проверяет, что текущий поток имеет доступ к этим страницам, и закрепляет их (MmProbeAndLockPages), а затем отображает эти страницы в память (MmMapLockedPagesSpecifyCache) данного процесса.

Другой сценарий возникает, когда драйверу требуется записать данные на страницы, доступные только для чтения (например, страницы раздела кода). Один из способов сделать это - использовать MDL. Драйвер инициализирует MDL, закрепляет его, а затем отображает по другому виртуальному адресу с разрешением на запись. В этом сценарии с помощью MDL драйвер может реализовать в режиме ядра функцию, подобную VirtualProtect.

Процессы и потоки

Поток определяется двумя структурами данных ядра: ETHREAD и KTHREAD. Структура ETHREAD содержит служебные сведения о потоке (то есть идентификатор потока, связанный процесс, включена или отключена отладка и т. д.). Структура KTHREAD хранит сведения для планирования диспетчером потоков, например данные о стеке потока, процессоре, на котором следует выполняться, состоянии готовности к оповещению и так далее. ETHREAD содержит KTHREAD.

Планировщик Windows работает с потоками.

Процесс содержит по меньшей мере один поток и определяется двумя структурами данных ядра: EPROCESS и KPROCESS. Структура EPROCESS хранит основные сведения о процессе (то есть идентификатор процесса, маркер безопасности, список потоков и т. д.). Структура KPROCESS хранит сведения для планирования процесса (то есть таблицу каталога страниц, предпочтительный процессор, системное/пользовательское время и т. д.). EPROCESS содержит KPROCESS. Как и ETHREAD с KTHREAD, эти структуры данных также непрозрачны, и обращаться к ним следует только посредством документированных процедур ядра. Однако их поля можно просмотреть с помощью отладчика ядра следующим образом:

Процессы

```
kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x2c8 ProcessLock      : _EX_PUSH_LOCK
   +0x2d0 CreateTime       : _LARGE_INTEGER
   +0x2d8 RundownProtect   : _EX_RUNDOWN_REF
   +0x2e0 UniqueProcessId  : Ptr64 Void
   +0x2e8 ActiveProcessLinks : _LIST_ENTRY
   +0x2f8 Flags2           : Uint4B
   +0x2f8 JobNotReallyActive : Pos 0, 1 Bit
   +0x2f8 AccountingFolded : Pos 1, 1 Bit
   +0x2f8 NewProcessReported : Pos 2, 1 Bit
...

   +0x3d0 InheritedFromUniqueProcessId : Ptr64 Void
   +0x3d8 LdtInformation   : Ptr64 Void
   +0x3e0 CreatorProcess   : Ptr64 _EPROCESS
   +0x3e0 ConsoleHostProcess : Uint8B
   +0x3e8 Peb              : Ptr64 _PEB
   +0x3f0 Session          : Ptr64 Void
...
0: kd> dt nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 ProfileListHead  : _LIST_ENTRY
   +0x028 DirectoryTableBase : Uint8B
   +0x030 ThreadListHead   : _LIST_ENTRY
   +0x040 ProcessLock      : Uint4B
...
   +0x0f0 ReadyListHead    : _LIST_ENTRY
   +0x100 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x108 ActiveProcessors : _KAFFINITY_EX
...
```

Потоки

```
0: kd> dt nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x348 CreateTime       : _LARGE_INTEGER
   +0x350 ExitTime         : _LARGE_INTEGER
...
   +0x380 ActiveTimerListLock : Uint8B
   +0x388 ActiveTimerListHead : _LIST_ENTRY
   +0x398 Cid              : _CLIENT_ID
...
0: kd> dt nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : Ptr64 Void
   +0x020 QuantumTarget    : Uint8B
   +0x028 InitialStack     : Ptr64 Void
   +0x030 StackLimit       : Ptr64 Void
   +0x038 StackBase        : Ptr64 Void
   +0x040 ThreadLock       : Uint8B
...
   +0x0d8 WaitListEntry    : _LIST_ENTRY
   +0x0d8 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x0e8 Queue            : Ptr64 _KQUEUE
   +0x0f0 Teb              : Ptr64 Void
```

> **Примечание.** Хотя мы говорим, что обращаться к этим структурам следует только посредством документированных процедур ядра, реальные руткиты изменяют частично документированные или полностью недокументированные поля этих структур для достижения своих целей. Например, один из способов скрыть процесс - удалить его из поля `ActiveProcessLinks` структуры `EPROCESS`. Однако поскольку эти структуры непрозрачны и не документированы, смещения полей могут (и действительно будут) меняться от выпуска к выпуску.

Существуют также аналогичные структуры данных пользовательского режима, в которых хранятся сведения о процессах и потоках. Для процессов это блок окружения процесса (process environment block, PEB/`ntdll!_PEB`), хранящий такие основные сведения, как базовый адрес загрузки, загруженные модули, кучи процесса и так далее. Для потоков это блок окружения потока (thread environment block, TEB/`ntdll!_TEB`), хранящий данные планирования потока и сведения о связанном процессе. Код пользовательского режима всегда может обращаться к TEB через сегмент FS (x86), сегмент GS (x64) или сопроцессор 15 (ARM). Эти объекты часто встречаются в системном коде, поэтому ниже перечислены способы доступа к ним:

`Текущий поток (режим ядра)`

x86

```
  mov     eax, large fs:124h
```

x64

```
  mov     rax, gs:188h
```

ARM

```
  MRC             p15, 0, R3,c13,c0, 3
  BICS.W          R0, R3, #0x3F
```

`TEB (пользовательский режим)`

x86

```
  mov     edx, large fs:18h
```

x64

```
  mov     rax, gs:30h
```

ARM

```
  MRC             p15, 0, R4,c13,c0, 2
```

Контекст выполнения

Каждый выполняющийся поток имеет контекст выполнения. Контекст выполнения содержит адресное пространство, маркер безопасности и другие важные свойства выполняющегося потока. В любой момент времени в Windows работают сотни потоков в разных контекстах выполнения. С точки зрения ядра можно определить три общих контекста выполнения:

• Контекст потока - контекст определенного потока (или, как правило, контекст запрашивающего потока, если поток пользовательского режима запрашивает обслуживание у ядра).

• Системный контекст - контекст потока, выполняющегося в процессе `System`.

• Произвольный контекст - контекст того потока, который выполнялся до того, как планировщик взял управление на себя.

Напомним, что каждый процесс имеет собственное адресное пространство. В режиме ядра важно знать, в каком контексте выполняется ваш код, поскольку это определяет адресное пространство, в котором вы находитесь, и принадлежащие вам привилегии безопасности. Не существует набора правил, позволяющего точно определить контекст выполнения в конкретном сценарии, однако могут помочь следующие общие рекомендации:

• При загрузке драйвера его точка входа (`DriverEntry`) выполняется в системном контексте.

• Когда приложение пользовательского режима отправляет запрос (IOCTL) драйверу, обработчик IOCTL драйвера работает в контексте потока (то есть в контексте потока пользовательского режима, инициировавшего запрос).

• APC выполняются в контексте потока (то есть в контексте потока, в очередь которого была помещена APC).

• DPC и таймеры выполняются в произвольном контексте.

• Рабочие элементы выполняются в системном контексте.

• Системные потоки выполняются в системном контексте, если параметр `ProcessHandle` равен `NULL` (обычный случай).

Например, точка входа драйвера имеет доступ только к адресному пространству процесса `System`, а потому не может обратиться к пространству какого-либо другого процесса, не вызвав нарушения доступа. Если поток режима ядра хочет изменить свой контекст выполнения на контекст другого процесса, он может использовать документированный API `KeStackAttachProcess`. Это полезно, когда драйверу необходимо читать или записывать память определенного процесса.

Примитивы синхронизации ядра

Ядро предоставляет распространенные примитивы синхронизации для использования другими компонентами. Самые распространенные из них - события, спин-блокировки, мьютексы, блокировки ресурсов и таймеры. В этом разделе объясняется их интерфейс и обсуждается их применение.

Объекты событий используются для обозначения состояния операции. Например, когда в системе остается мало памяти невыгружаемого пула, ядро может уведомить драйвер с помощью событий. Событие может находиться в одном из двух состояний: сигнальном или несигнальном. Значение сигнального и несигнального состояния зависит от сценария использования. Внутренне событие является объектом, определенным структурой `KEVENT` и инициализированным API `KeInitializeEvent`. После инициализации события поток может ожидать его с помощью `KeWaitForSingleObject` или `KeWaitForMultipleObjects`. События часто используются в драйверах, чтобы уведомить другие потоки о завершении обработки или выполнении определенного условия.

Таймеры используются для обозначения того, что прошел определенный интервал времени. Например, при каждом наступлении нового столетия ядро выполняет некоторый код для обновления времени; нижележащим механизмом для этого служат таймеры. Внутренне объекты таймеров определяются структурой `KTIMER` и инициализируются процедурой `KeInitializeTimer/Ex`. При инициализации таймеров можно указать необязательную процедуру DPC, которая будет выполнена по истечении времени. По определению, каждый процессор имеет собственный таймер очереди; в частности, поле `TimerTable` в PRCB представляет собой список таймеров данного процессора. Таймеры часто используются для периодического выполнения действий или их выполнения в определенное время. И таймеры, и DPC подробнее рассматриваются далее в этой главе.

Мьютексы используются для монопольного доступа к общему ресурсу. Например, если два потока одновременно изменяют общий связанный список без мьютекса, они могут повредить список; решение заключается в том, чтобы обращаться к связанному списку только при удержании мьютекса. Хотя основная семантика мьютекса не меняется, ядро Windows предлагает два разных вида мьютексов: защищенный мьютекс и быстрый мьютекс. Защищенные мьютексы быстрее быстрых мьютексов, но доступны только в Windows 2003 и более новых версиях. Внутренне мьютекс определяется структурой `FAST_MUTEX` или `GUARDED_MUTEX` и инициализируется функцией `ExInitialize{Fast,Guarded}Mutex`. После инициализации их можно захватывать и освобождать с помощью различных API; дополнительные сведения см. в документации Windows Driver Kit.

Спин-блокировки также используются для монопольного доступа к общему ресурсу. Хотя концептуально они похожи на мьютексы, они применяются для защиты общих ресурсов, к которым обращаются на `DISPATCH_LEVEL` или более высоком IRQL. Например, ядро захватывает спин-блокировку перед изменением критически важных глобальных структур данных, таких как список активных процессов; это необходимо потому, что в многопроцессорной системе несколько потоков могут одновременно обращаться к списку и изменять его. Внутренне спин-блокировки определяются структурой `KSPIN_LOCK` и инициализируются функцией `KeInitializeSpinLock`. После инициализации их можно захватывать и освобождать с помощью различных документированных API; дополнительные сведения см. в документации WDK. Обратите внимание, что код, удерживающий спин-блокировку, выполняется на `DISPATCH_LEVEL` или выше; следовательно, выполняемый код и память, к которой он обращается, всегда должны находиться в физической памяти.

Списки

Связанные списки - фундаментальные строительные блоки динамических структур данных в ядре и драйверах. На основе списков построены многие важные структуры данных ядра (например, относящиеся к процессам и потокам). В действительности списки используются настолько часто, что WDK предоставляет набор функций для их создания и обработки универсальным способом. Хотя концептуально списки просты и не имеют прямого отношения к пониманию понятий ядра или практике обратного анализа, они вводятся здесь по двум важным причинам. Во-первых, они используются практически в каждой структуре данных ядра Windows, рассматриваемой в этой главе. Ядро часто работает с элементами различных списков (например, списка загруженных модулей, списка активных процессов, списка ожидающих потоков и т. д.), содержащихся в этих структурах, поэтому важно понимать механику таких операций. Во-вторых, хотя функции, работающие со списками, например `InsertHeadList`, `InsertTailList`, `RemoveHeadList`, `RemoveEntryList` и т. д., представлены в заголовках WDK в исходном виде, компилятор всегда встраивает их, поэтому в реальных двоичных файлах они никогда не появляются как «функции» на уровне ассемблера; иными словами, они никогда не встречаются как цель вызова или перехода. Следовательно, нужно понимать подробности их реализации и шаблоны использования, чтобы распознавать их на уровне ассемблера.

Подробности реализации

WDK предоставляет функции, поддерживающие три типа списков:

• Односвязный список - список, элементы которого связаны одним указателем (`Next`).

• Упорядоченный односвязный список - односвязный список с поддержкой атомарных операций. Например, можно удалить из списка первый элемент, не беспокоясь о захвате блокировки.

• Кольцевой двусвязный список - список, элементы которого связаны двумя указателями: один указывает на следующий элемент (`Flink`), а другой - на предыдущий (`Blink`).

С точки зрения использования на уровне исходного кода все три типа концептуально одинаковы. В этой главе рассматриваются только двусвязные списки, поскольку они встречаются чаще всего. В одном из упражнений вам будет предложено изучить документацию WDK по операциям со списками и написать драйвер, использующий все три типа списков.

Реализация построена на основе одной структуры:

```
typedef struct _LIST_ENTRY {
   struct _LIST_ENTRY *Flink;
   struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

`LIST_ENTRY` может представлять заголовок списка или элемент списка. Заголовок списка представляет «голову» списка и обычно не хранит никаких данных, кроме самой структуры `LIST_ENTRY`; всем функциям списков требуется указатель на заголовок списка. Элемент списка - это настоящий элемент, хранящий данные; на практике он представляет собой структуру `LIST_ENTRY`, встроенную в более крупную структуру.

Перед использованием списки необходимо инициализировать с помощью `InitializeListHead`. Эта функция просто задает полям `Flink` и `Blink` указатели на заголовок списка. Ее код приведен ниже и проиллюстрирован на рисунке 3-4:

```
VOID InitializeListHead(PLIST_ENTRY ListHead) {
    ListHead->Flink = ListHead->Blink = ListHead;
    return;
}
```

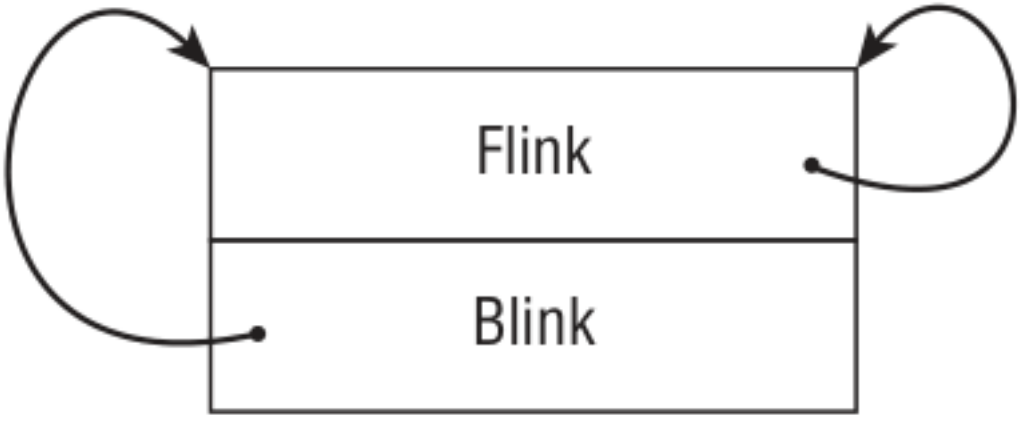

Figure 3-4

Рисунок 3-4. Инициализированный заголовок двусвязного списка

В ассемблерной форме это преобразуется в три команды: одна получает `ListHead`, а две заполняют указатели `Flink` и `Blink`. Рассмотрим, как `InitializeListHead` проявляется в ассемблерном коде x86, x64 и ARM:

x86

```
lea     eax, [esi+2Ch]
mov     [eax+4], eax
mov     [eax], eax
```

x64

```
lea     r11, [rbx+48h]
mov     [r11+8], r11
mov     [r11], r11
```

ARM

```
ADDS.W          R3, R4, #0x2C
STR             R3, [R3,#4]
STR             R3, [R3]
```

Во всех трех случаях в операциях только для записи используются один и тот же указатель и регистр. Еще одно важное наблюдение: записи выполняются по смещениям +0 и +4/8 относительно базового регистра; эти смещения соответствуют указателям `Flink` и `Blink` в структуре. Всякий раз, когда вы видите такой шаблон кода, следует подумать о списках.

После инициализации списка элементы можно вставлять в начало или конец. Как упоминалось ранее, элемент списка - это просто `LIST_ENTRY` внутри более крупной структуры; например, структура ядра `KDPC` (рассматриваемая далее в этой главе) имеет поле `DpcListEntry`:

Определение C

```
typedef struct _KDPC {
    UCHAR Type;
    UCHAR Importance;
    volatile USHORT Number;
    LIST_ENTRY DpcListEntry;
    PKDEFERRED_ROUTINE DeferredRoutine;
    PVOID DeferredContext;
    PVOID SystemArgument1;
    PVOID SystemArgument2;
    __volatile PVOID DpcData;
} KDPC, *PKDPC, *PRKDPC;
```

x64

```
0: kd> dt nt!_KDPC
   +0x000 Type             : UChar
   +0x001 Importance       : UChar
   +0x002 Number           : Uint2B
   +0x008 DpcListEntry     : _LIST_ENTRY
   +0x018 DeferredRoutine  : Ptr64     void
   +0x020 DeferredContext  : Ptr64 Void
   +0x028 SystemArgument1  : Ptr64 Void
   +0x030 SystemArgument2  : Ptr64 Void
   +0x038 DpcData          : Ptr64 Void
```

Предположим, имеется список с одним элементом `KDPC`, как показано на рисунке 3-5.

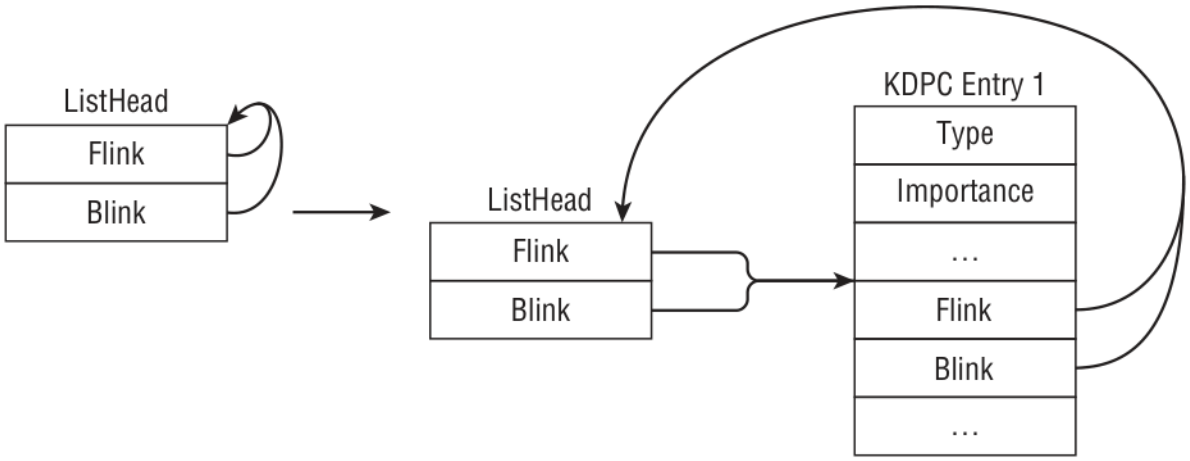

Figure 3-5

Рисунок 3-5. Список с одним элементом KDPC

Вставка выполняется с помощью `InsertHeadList` и `InsertTailList`. Рассмотрим вставку элемента в начало, показанную на рисунке 3-6.

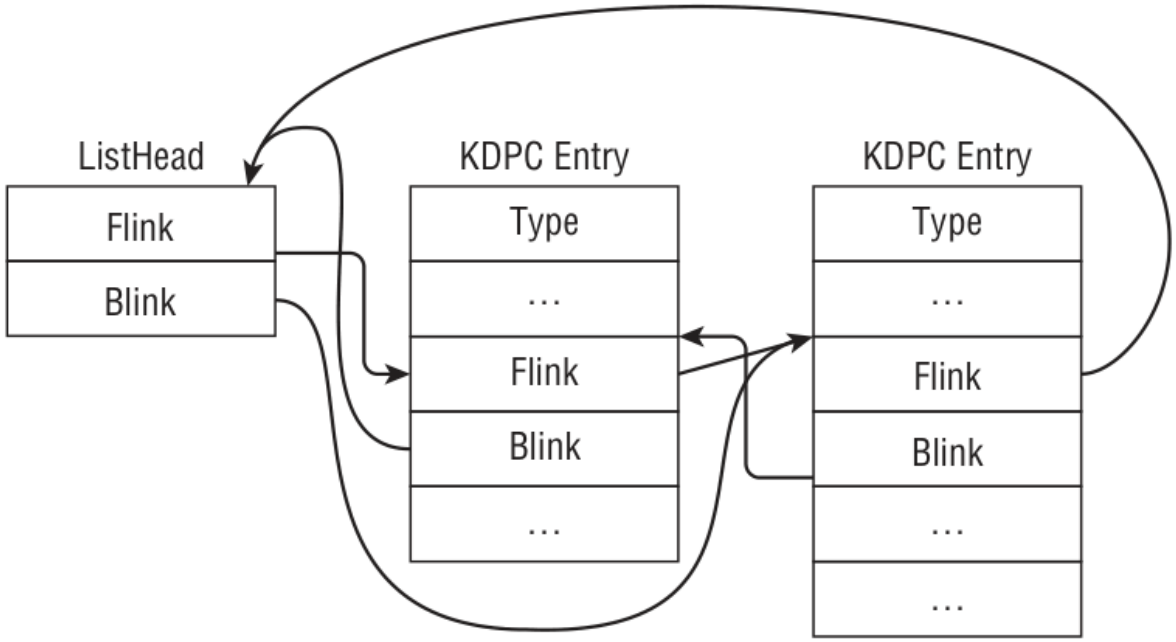

Figure 3-6

Рисунок 3-6. Вставка элемента KDPC в начало списка

Ниже показаны исходный код этих процедур и то, как они могут проявляться в ассемблерной форме:

> **Примечание.** Эти фрагменты взяты из функции ядра `KeInsertQueueDpc` в Windows 8; для ясности из них удалено несколько строк. Здесь важно увидеть, как новый элемент вставляется в список. Планирование команд может изменить порядок некоторых команд, но в основном они останутся теми же.

`InsertHeadList`

`C`

```
VOID InsertHeadList(PLIST_ENTRY ListHead, PLIST_ENTRY Entry) {
    PLIST_ENTRY Flink;
    Flink = ListHead->Flink;
    Entry->Flink = Flink;
    Entry->Blink = ListHead;
    Flink->Blink = Entry;
    ListHead->Flink = Entry;
    return;
}
```

ARM

```
LDR     R1, [R5]
STR     R5, [R2,#4]
STR     R1, [R2]
STR     R2, [R1,#4]
STR     R2, [R5]
```

x86

```
mov     edx, [ebx]
mov     [ecx], edx
mov     [ecx+4], ebx
mov     [edx+4], ecx
mov     [ebx], ecx
```

x64

```
mov     rcx, [rdi]
mov     [rax+8], rdi
mov     [rax], rcx
mov     [rcx+8], rax
mov     [rdi], rax
```

InsertTailList

C

```
VOID InsertTailList(PLIST_ENTRY ListHead, PLIST_ENTRY Entry) {
    PLIST_ENTRY Blink;
    Blink = ListHead->Blink;
    Entry->Flink = ListHead;
    Entry->Blink = Blink;
    Blink->Flink = Entry;
    ListHead->Blink = Entry;
    return;
}
```

ARM

```
LDR     R1, [R5,#4]
STR     R5, [R2]
STR     R1, [R2,#4]
STR     R2, [R1]
STR     R2, [R5,#4]
```

x86

```
mov     ecx, [ebx+4]
mov     [eax], ebx
mov     [eax+4], ecx
mov     [ecx], eax
mov     [ebx+4], eax
```

x64

```
mov     rcx, [rdi+8]
mov     [rax], rdi
mov     [rax+8], rcx
mov     [rcx], rax
mov     [rdi+8], rax
```

В приведенных выше фрагментах R5/EBX/RDI указывают на `ListHead`, а R2/ECX/RAX - на `Entry`.

Удаление выполняется с помощью `RemoveHeadList`, `RemoveTailList` и `RemoveEntryList`. Этим процедурам обычно предшествует функция `IsListEmpty`, которая просто проверяет, указывает ли поле `Flink` заголовка списка на сам заголовок:

`IsListEmpty`

C

```
BOOLEAN IsListEmpty(PLIST_ENTRY ListHead) {
    return (BOOLEAN)(ListHead->Flink == ListHead);
}
```

ARM

```
LDR R2, [R4]
CMP R2, R4
```

x86

```
mov eax, [esi]
cmp eax, esi
```

x64

```
mov rax, [rbx]
cmp rax, rbx
```

`RemoveHeadList`

C

```
PLIST_ENTRY RemoveHeadList(PLIST_ENTRY ListHead) {
    PLIST_ENTRY Flink;

    PLIST_ENTRY Entry;
    Entry = ListHead->Flink;
    Flink = Entry->Flink;
    ListHead->Flink = Flink;
    Flink->Blink = ListHead;
    return Entry;
}
```

ARM

```
LDR R2, [R4]
LDR R1, [R2]
STR R1, [R4]
STR R4, [R1,#4]
```

x86

```
mov eax, [esi]
mov ecx, [eax]
mov [esi], ecx
mov [ecx+4], esi
```

x64

```
mov rax, [rbx]
mov rcx, [rax]
mov [rbx], rcx
mov [rcx+8], rbx
```

`RemoveTailList`

C

```
PLIST_ENTRY RemoveTailList(PLIST_ENTRY ListHead) {
    PLIST_ENTRY Blink;
    PLIST_ENTRY Entry;
    Entry = ListHead->Blink;
    Blink = Entry->Blink;
    ListHead->Blink = Blink;
    Blink->Flink = ListHead;
    return Entry;
}
```

ARM

```
LDR  R6, [R5,#4]
LDR  R2, [R6,#4]
STR  R2, [R5,#4]
STR  R5, [R2]
```

x86

```
mov  ebx, [edi+4]
mov  eax, [ebx+4]
mov  [edi+4], eax
mov  [eax], edi
```

x64

```
mov  rsi, [rdi+8]
mov  rax, [rsi+8]
mov  [rdi+8], rax
mov  [rax], rdi
```

RemoveEntryList

C

```
BOOLEAN RemoveEntryList(PLIST_ENTRY Entry){
    PLIST_ENTRY Blink;
    PLIST_ENTRY Flink;
    Flink = Entry->Flink;
    Blink = Entry->Blink;
    Blink->Flink = Flink;
    Flink->Blink = Blink;
    return (BOOLEAN)(Flink == Blink);
}
```

ARM

```
LDR  R1,[R0]
LDR  R2,[R0,#4]
STR  R1,[R2]
STR  R2,[R1,#4]
```

x86

```
mov  edx, [ecx]
mov  eax, [ecx+4]
mov  [eax], edx
mov  [edx+4], eax
```

x64

```
mov  rdx, [rcx]
mov  rax, [rcx+8]
mov  [rax], rdx
mov  [rdx+8], rax
```

Обратите внимание, что все функции обработки списков работают исключительно со структурой LIST_ENTRY. Чтобы выполнять полезные действия с элементом списка, код должен манипулировать фактическими данными элемента. Как программы обращаются к полям в элементе списка? Это делается с помощью макроса CONTAINING_RECORD:

```
#define CONTAINING_RECORD(address, type, field) ((type *)( \
                                          (PCHAR)(address) - \
                                          (ULONG_PTR)(&((type *)0)->field)))
```

CONTAINING_RECORD возвращает базовый адрес структуры следующим способом: вычисляет смещение поля в структуре, приводя указатель структуры к 0, а затем вычитает это смещение из действительного адреса поля. На практике этот макрос обычно принимает адрес поля LIST_ENTRY в элементе списка, тип элемента списка и имя этого поля. Например, предположим, что у вас есть список элементов KDPC (см. приведенное ранее определение) и вам нужна функция для доступа к полю DeferredRoutine; код будет выглядеть следующим образом:

```
PKDEFERRED_ROUTINE ReadEntryDeferredRoutine (PLIST_ENTRY entry) {
       PKDPC p;
       p = CONTAINING_RECORD(entry, KDPC, DpcListEntry);
       return p->DeferredRoutine;
}
```

Этот макрос часто используется сразу после вызова одной из процедур удаления из списка или во время перечисления элементов списка.

Пошаговый разбор

Обсудив понятия и подробности реализации функций обработки списков в режиме ядра, теперь применим их к анализу образца C. Этот пошаговый разбор преследует три цели:

• Показать один из распространенных способов применения списков в настоящем драйвере/рутките.

• Продемонстрировать неопределенности, с которыми на практике сталкивается специалист по обратному анализу.

• Обсудить проблемы недокументированных структур и жестко заданных смещений.

Этот драйвер выполняет множество действий, но нас интересуют только две функции: sub_11553 и sub_115DA. Рассмотрим следующий фрагмент из sub_115DA:

```
01: .text:000115FF   mov     eax, dword_1436C
02: .text:00011604   mov     edi, ds:wcsncpy
03: .text:0001160A   mov     ebx, [eax]
04: .text:0001160C   mov     esi, ebx
05: .text:0001160E loop_begin:
06: .text:0001160E   cmp     dword ptr [esi+20h], 0
07: .text:00011612   jz      short failed
08: .text:00011614   push    dword ptr [esi+28h]
09: .text:00011617   call    ds:MmIsAddressValid
10: .text:0001161D   test    al, al
11: .text:0001161F   jz      short failed
12: .text:00011621   mov     eax, [esi+28h]
13: .text:00011624   test    eax, eax
14: .text:00011626   jz      short failed
15: .text:00011628   movzx   ecx, word ptr [esi+24h]
16: .text:0001162C   shr     ecx, 1
17: .text:0001162E   push    ecx                             ; size_t
18: .text:0001162F   push    eax                             ; wchar_t *
19: .text:00011630   lea     eax, [ebp+var_208]
20: .text:00011636   push    eax                             ; wchar_t *
21: .text:00011637   call    edi ; wcsncpy
22: .text:00011639   lea     eax, [ebp+var_208]
23: .text:0001163F   push    eax                             ; wchar_t *
```

```
24: .text:00011640   call    ds:_wcslwr
25: .text:00011646   lea     eax, [ebp+var_208]

26: .text:0001164C   push    offset aKrnl                ; "krnl"
27: .text:00011651   push    eax                         ; wchar_t *
28: .text:00011652   call    ds:wcsstr
29: .text:00011658   add     esp, 18h
30: .text:0001165B   test    eax, eax
31: .text:0001165D   jnz     short matched_krnl
32: .text:0001165F   mov     esi, [esi]
33: .text:00011661   cmp     esi, ebx
34: .text:00011663   jz      short loop_end
35: .text:00011665   jmp     short loop_begin
36: .text:00011667 matched_krnl:
37: .text:00011667   lea     eax, [ebp+var_208]
38: .text:0001166D   push    '\'                         ; wchar_t
39: .text:0001166F   push    eax                         ; wchar_t *
40: .text:00011670   call    ds:wcsrchr
41: .text:00011676   pop     ecx
42: .text:00011677   test    eax, eax
```

Строки 1-4 читают указатель из глобальной переменной `dword_1436C` и сохраняют его в EBX и ESI. Тело цикла обращается по смещениям `0x20` и `0x28` относительно этого указателя; следовательно, можно заключить, что это указатель на структуру размером не менее `0x2c` байт. В конце цикла из структуры читается еще один указатель и сравнивается с исходным указателем (сохраненным в строке 3). Обратите внимание, что указатель читается по смещению 0. Поэтому на данном этапе можно предположить, что этот цикл перебирает список, в котором указатель «следующий» находится по смещению 0. Можно ли утверждать, что эта структура содержит поле `LIST_ENTRY` по смещению 0? Нет, сейчас для этого недостаточно конкретных данных. Выясним, откуда берется глобальная переменная `dword_1436C`.

`sub_11553` использует соглашение о вызовах STDCALL и принимает два параметра: указатель на `DRIVER_OBJECT` и указатель на глобальную переменную `dword_1436C`. Она содержит следующий интересный фрагмент кода:

```
01: .text:00011578   mov     eax, 0FFDFF034h
02: .text:0001157D   mov     eax, [eax]
03: .text:0001157F   mov     eax, [eax+70h]
04: ...
05: .text:0001159E   mov     ecx, [ebp+arg_4]  ; указатель на глобальную переменную
06: .text:000115A1   mov     [ecx], eax
```

Строка 2 читает указатель по жестко заданному адресу `0xFFDFF034`. В Windows XP по адресу `0xFFDFF000` находится структура блока управления процессором (рассматриваемая далее в этой главе), а по смещению `0x34` - указатель `KdVersionBlock`. Строки 3-6 читают значение указателя по смещению `0x70` внутри `KdVersionBlock` и записывают его обратно в глобальную переменную; мы знаем, что это указатель, поскольку он используется для перебора элементов списка в `sub_115DA`. Чтобы определить точный тип элемента списка, нужно выяснить, что находится по смещению `0x70` структуры `KdVersionBlock`. Поскольку это недокументированная структура, зависящая от конкретной ОС, придется либо провести обратный анализ ядра Windows XP, либо поискать в Интернете, не выяснил ли это уже кто-то другой. Результаты показывают, что в Windows XP по смещению `0x70` структуры `KdVersionBlock` находится указатель на заголовок глобального списка `PsLoadedModuleList`. Каждый элемент этого списка имеет тип `KLDR_DATA_TABLE_ENTRY` и хранит сведения о загруженных в данный момент модулях ядра (имя, базовый адрес, размер и т. д.); первое поле этой структуры имеет тип `LIST_ENTRY`. Это согласуется с нашим предыдущим выводом о том, что по смещению 0 находится указатель «следующий» (точнее, `Flink`).

> **Примечание.** Структура `KLDR_DATA_TABLE_ENTRY` не документирована, но очень похожа на `LDR_DATA_TABLE_ENTRY`, которая присутствует в открытых символах. В Windows XP поля `FullDllName` и `BaseDllName` находятся по тем же смещениям (`0x24` и `0x2c`).

Если предположить, что найденные в Интернете сведения верны, эти две функции можно кратко описать следующим образом:

• sub_11553 читает указатель KdVersionBlock из блока управления процессором и получает из него указатель на PsLoadedModuleList; этот указатель сохраняется в глобальной переменной. PsLoadedModuleList - заголовок списка, элементы которого имеют тип KLDR_DATA_TABLE_ENTRY. Этой функции будет дано понятное имя GetLoadedModuleList.

• sub_115DA использует указатель на заголовок списка, чтобы перебрать все элементы в поисках имени модуля с подстрокой "krnl". Код ищет подстроку "krnl", поскольку автор разыскивает имя образа ядра NT (обычно "ntoskrnl.exe"). Этой функции будет дано понятное имя GetKernelName.

Их можно кратко перевести обратно на C:

```
typedef struct _KLDR_DATA_TABLE_ENTRY {
    LIST_ENTRY ListEntry;
    ...
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ...
} KLDR_DATA_TABLE_ENTRY, *PKLDR_DATA_TABLE_ENTRY;

BOOL GetLoadedModuleList(PDRIVER_OBJECT drvobj, PLIST_ENTRY g_modlist)
{
    ...
    g_modlist = (PCR->KdVersionBlock) + 0x70
    ...
}

BOOL GetKernelName()
{
    WCHAR fname[...];
    PKLDR_DATA_TABLE_ENTRY entry;
    PLIST_ENTRY p = g_modlist->Flink;
    while (p != g_modlist)
    {

        entry = CONTAINING_RECORD(p, KLDR_DATA_TABLE_ENTRY, ListEntry);
        ...
        wcsncpy(fname, entry->FullDllName.Buffer, entry->FullDllName
        .Length * 2);
        ...
        if (wcsstr(fname, L"krnl") != NULL) { ... }
        p = p->Flink;
     }
     ...
}
```

Хотя может показаться, что этот драйвер работает в определенной версии Windows, с ним связано несколько проблем. Во-первых, он предполагает, что PCR всегда находится по адресу 0xFFDFF000, а KdVersionBlock - по смещению 0x34; эти предположения неверны для Windows Vista и более новых версий. Во-вторых, драйвер предполагает, что KdVersionBlock всегда содержит допустимое значение; это неверно, поскольку значение действительно только для PCR первого процессора. Следовательно, если бы этот код выполнялся в многопроцессорной системе и поток оказался запланирован на другом процессоре, код завершился бы сбоем. В-третьих, он предполагает наличие UNICODE_STRING по смещению 0x24 структуры KLDR_DATA_TABLE_ENTRY (которая сама не документирована); это может быть верно не всегда, поскольку Microsoft может добавлять или удалять поля из определения структуры, вызывая изменение смещения. В-четвертых, этот код наверняка не сработает в ядре x64, поскольку там все смещения отличаются. Наконец, список загруженных модулей может измениться (например, из-за выгрузки драйверов), пока драйвер перебирает список; в результате он может получить устаревшие результаты или вызвать нарушение доступа, обратившись к модулю, которого больше нет. Обратите также внимание, что при переборе глобального списка драйвер не использует никакого механизма блокировки.

Анализируя больше руткитов режима ядра или сторонних драйверов, вы будете часто встречать код, написанный с такими преждевременными предположениями.

Для данного конкретного образца видно, что разработчик всего лишь хочет получить имя и базовый адрес образа ядра. Этого легко можно было достичь с помощью документированного API ядра `AuxKlibQueryModuleInformation`. (См. также упражнение по `AuxKlibQueryModuleInformation`.)

В заключение кратко обсудим ход мыслей при анализе этих двух функций. Как удалось перейти от кажущихся случайными значений вроде `0xFFDF034`, `0x70` и `0x28` к PCR, `KdVersionBlock`, `PsLoadedModuleList`, `KLDR_DATA_TABLE_ENTRY` и так далее? Правда в том, что у нас уже имелись знания ядра и опыт анализа драйверов режима ядра, поэтому мы инстинктивно подумали об этих структурах. Например, мы начали с цикла, обрабатывающего каждый элемент списка в поисках подстроки `"krnl"`; мы сразу предположили, что выполняется поиск имени образа ядра. Смещения строки и длины (`0x24` и `0x28`) указали нам на `UNICODE_STRING`; благодаря знаниям ядра мы предположили, что это структура `KLDR_DATA_TABLE_ENTRY`, и подтвердили это с помощью открытых символов. Далее мы знаем, что `PsLoadedModuleList` - глобальный заголовок списка загруженных модулей. Поскольку `PsLoadedModuleList` не является экспортируемым символом, мы знаем, что драйвер должен получить его из другой структуры. Двигаясь в обратном направлении, мы видим жестко заданный адрес памяти `0xFFDF034` и сразу думаем о PCR. Проверим это в отладчике:

```
0: kd> dt nt!_KPCR 0xffdff000
   +0x000 NtTib            : _NT_TIB
   +0x01c SelfPcr          : 0xffdff000 _KPCR
   +0x020 Prcb             : 0xffdff120 _KPRCB
   +0x024 Irql             : 0 ''
   +0x028 IRR              : 0
   +0x02c IrrActive        : 0
   +0x030 IDR              : 0xffffffff
   +0x034 KdVersionBlock   : 0x8054d2b8 Void
...
```

Из опыта мы знаем, что `KdVersionBlock` - указатель на большую структуру, в которой хранятся интересные сведения, например базовый адрес ядра и заголовки списков. На этом этапе у нас есть все сведения и структуры данных, необходимые для понимания кода.

Как видите, в основе анализа лежит систематический мыслительный процесс; однако он требует значительных фоновых знаний об операционной системе и опыта. Поначалу у вас может не быть всех знаний и интуиции, необходимых для быстрого понимания драйверов режима ядра. Не бойтесь! Эта книга пытается заложить прочный фундамент, объясняя все основные понятия и структуры данных ядра. Имея прочный фундамент и много практикуясь (см. упражнения), со временем вы сможете делать это с большой легкостью. Помните: фундаментальные знания + интуиция + опыт + терпение = навыки.

Упражнения

1. В Windows 8 x64 следующие функции ядра содержат по меньшей мере одно встроенное вхождение `InitializeListHead`:

• `CcAllocateInitializeMbcb`

• `CmpInitCallbacks`

• `ExCreateCallback`

• `ExpInitSystemPhase0`

• `ExpInitSystemPhase1`

• `ExpTimerInitialization`

- InitBootProcessor
- IoCreateDevice
- IoInitializeIrp
- KeInitThread
- KeInitializeMutex
- KeInitializeProcess
- KeInitializeTimerEx
- KeInitializeTimerTable
- KiInitializeProcessor
- KiInitializeThread
- MiInitializeLoadedModuleList
- MiInitializePrefetchHead
- PspAllocateProcess
- PspAllocateThread

Найдите, где в эти процедуры встроена InitializeListHead.

2. Повторите предыдущее упражнение для InsertHeadList в следующих процедурах:

- CcSetVacbInFreeList
- CmpDoSort
- ExBurnMemory
- ExFreePoolWithTag
- IoPageRead
- IovpCallDriver1
- KeInitThread
- KiInsertQueueApc
- KeInsertQueueDpc
- KiQueueReadyThread
- MiInsertInSystemSpace
- MiUpdateWsle
- ObpInsertCallbackByAltitude

3. Повторите предыдущее упражнение для InsertTailList в следующих процедурах:

- AlpcpCreateClientPort
- AlpcpCreateSection
- AlpcpCreateView
- AuthzBasepAddSecurityAttributeToLists
- CcFlushCachePriv
- CcInitializeCacheManager
- CcInsertVacbArray

- CcSetFileSizesEx
- CmRenameKey
- ExAllocatePoolWithTag
- ExFreePoolWithTag
- ExQueueWorkItem
- ExRegisterCallback
- ExpSetTimer
- IoSetIoCompletionEx2
- KeInsertQueueDpc
- KeStartThread
- KiAddThreadToScbQueue
- KiInsertQueueApc
- KiQueueReadyThread
- MiInsertNewProcess
- PnpRequestDeviceAction
- PspInsertProcess
- PspInsertThread

4. Повторите предыдущее упражнение для RemoveHeadList в следующих процедурах:

- AlpcpFlushResourcesPort
- CcDeleteMbcb
- CcGetVacbMiss
- CmpLazyCommitWorker
- ExAllocatePoolWithTag
- FsRtlNotifyCompleteIrpList
- IopInitializeBootDrivers
- KiProcessDisconnectList
- PnpDeviceCompletionQueueGetCompletedRequest
- RtlDestroyAtomTable
- RtlEmptyAtomTable
- RtlpFreeAllAtom

5. Повторите предыдущее упражнение для RemoveTailList в следующих процедурах:

- BootApplicationPersistentDataProcess
- CmpCallCallBacks
- CmpDelayCloseWorker
- ObpCallPostOperationCallbacks
- RaspAddCacheEntry

6. Повторите предыдущее упражнение для `RemoveEntryList` в следующих процедурах:

- `AlpcSectionDeleteProcedure`
- `AlpcpDeletePort`
- `AlpcpUnregisterCompletionListDatabase`
- `AuthzBasepRemoveSecurityAttributeFromLists`
- `CcDeleteBcbs`
- `CcFindNextWorkQueueEntry`
- `CcLazyWriteScan`
- `CcSetFileSizesEx`
- `CmShutdownSystem`
- `CmUnRegisterCallback`
- `CmpCallCallBacks`
- `CmpPostApc`
- `ExFreePoolWithTag`
- `ExQueueWorkItem`
- `ExTimerRundown`
- `ExpDeleteTimer`
- `ExpSetTimer`
- `IoDeleteDevice`
- `IoUnregisterFsRegistrationChange`
- `IopfCompleteRequest`
- `KeDeregisterBugCheckCallback`
- `KeDeregisterObjectNotification`
- `KeRegisterObjectNotification`
- `KeRemoveQueueApc`
- `KeRemoveQueueDpc`
- `KiCancelTimer`
- `KeTerminateThread`
- `KiDeliverApc`
- `KiExecuteAllDpcs`
- `KiExpireTimerTable`
- `KiFindReadyThread`
- `KiFlushQueueApc`
- `KiInsertTimerTable`
- `KiProcessExpiredTimerList`
- `MiDeleteVirtualAddresses`
- `NtNotifyChangeMultipleKeys`

• `ObRegisterCallbacks`

• `ObUnRegisterCallbacks`

7. Повторите предыдущие упражнения в Windows 8 x86/ARM и Windows 7 x86/x64. Какие различия (если они были) вы обнаружили?

8. Если вы выполнили упражнения для `InsertHeadList`, `InsertTailList`, `RemoveHeadList`, `RemoveTailList` и `RemoveEntryList` в Windows 8, вы должны были заметить конструкцию кода, общую для всех этих функций. Эта конструкция также должна позволить легко обнаруживать встроенные процедуры вставки в список и удаления из списка. Объясните эту конструкцию кода и причину ее существования. Подсказка: эта конструкция существует только в Windows 8, а для ее понимания потребуется изучить IDT.

9. В пошаговом разборе мы упоминали, что драйвер может перечислить все загруженные модули с помощью документированного API `AuxKlibQueryModuleInformation`. Гарантирует ли этот API, что возвращенный список модулей всегда актуален? Объясните свой ответ. Затем проведите обратный анализ `AuxKlibQueryModuleInformation` в Windows 8 и объясните, как он работает. Как он обрабатывает случай, когда несколько потоков запрашивают доступ к списку загруженных модулей? Примечание: внутренняя функция, обрабатывающая этот (и другие) запросы, довольно велика, поэтому вам потребуется терпение. В качестве альтернативы можно воспользоваться отладчиком, чтобы проследить интересующий код.

10. Объясните работу следующих функций: `KeInsertQueueDpc`, `KiRetireDpcList`, `KiExecuteDpc` и `KiExecuteAllDpcs`. Если хотите превзойти ожидания, декомпилируйте эти функции из ассемблерного кода x86 и x64 и объясните различия.

Асинхронное и ситуативное выполнение

В течение жизненного цикла драйвер может создавать системные потоки, регистрировать функции обратного вызова для определенных событий, ставить функцию в очередь для выполнения в будущем и так далее. В этом разделе рассматриваются различные механизмы, с помощью которых драйвер может реализовать такие формы асинхронного и ситуативного выполнения. К этим механизмам относятся системные потоки, рабочие элементы, APC, DPC, таймеры, а также функции обратного вызова процессов и потоков.

Системные потоки

Обычная программа пользовательского режима может иметь несколько потоков, обрабатывающих различные запросы. Аналогичным образом драйвер может создавать несколько потоков для обработки запросов от ядра или пользователя. Эти потоки можно создавать с помощью API `PsCreateSystemThread`:

```
NTSTATUS PsCreateSystemThread(
  _Out_      PHANDLE ThreadHandle,
  _In_       ULONG DesiredAccess,
  _In_opt_   POBJECT_ATTRIBUTES ObjectAttributes,
  _In_opt_   HANDLE ProcessHandle,
  _Out_opt_  PCLIENT_ID ClientId,
  _In_       PKSTART_ROUTINE StartRoutine,
  _In_opt_   PVOID StartContext
);
```

Если этот API вызывается с параметром `ProcessHandle`, равным `NULL`, он создает новый поток в процессе `System` и задает ему начальную процедуру `StartRoutine`. Способ использования системных потоков зависит от требований драйвера. Например, драйвер может решить создать поток во время инициализации для обработки последующих запросов ввода-вывода или ожидания некоторых событий. Один конкретный пример - создание ядром системного потока для обработки DPC (см. также функцию `KiStartDpcThread`).

Упражнения

1. Прочитав несколько интернет-форумов, вы замечаете, что некоторые люди утверждают, будто `PsCreateSystemThread` создает поток в контексте вызывающего процесса. Иными словами, они предполагают, что если вызвать `PsCreateSystemThread` в обработчике IOCTL, новый поток окажется в контексте запросившего приложение пользовательского режима. Оцените правильность этого утверждения, написав драйвер, который вызывает `PsCreateSystemThread` в обработчике IOCTL. Затем поэкспериментируйте с ненулевым `ProcessHandle` и определите, отличается ли контекст.

2. Найдите как можно больше перекрестных ссылок на вызовы `PsCreateSystemThread` в образе ядра. Определите, передает ли какой-либо из них ненулевой

параметр `ProcessHandle`. Объясните назначение этих процедур. Повторите упражнение для как можно большего количества функций.

Рабочие элементы

Рабочие элементы похожи на системные потоки, за исключением того, что для них не создаются физические объекты потоков. Рабочий элемент - это просто объект в очереди, обрабатываемой пулом системных потоков. Если говорить конкретно, рабочий элемент представляет собой структуру следующего вида:

```
0: kd> dt nt!_IO_WORKITEM
   +0x000 WorkItem         : _WORK_QUEUE_ITEM
   +0x020 Routine          : Ptr64     void
   +0x028 IoObject         : Ptr64 Void
   +0x030 Context          : Ptr64 Void
   +0x038 Type             : Uint4B
   +0x03c ActivityId       : _GUID
0: kd> dt nt!_WORK_QUEUE_ITEM
   +0x000 List             : _LIST_ENTRY
   +0x010 WorkerRoutine    : Ptr64     void
   +0x018 Parameter        : Ptr64 Void
```

Обратите внимание, что ее поле `WorkItem` в действительности является элементом списка, содержащим рабочую процедуру и параметр. Позже этот элемент будет вставлен в очередь. Драйвер вызывает функцию `IoAllocateWorkItem`, чтобы получить указатель на `IO_WORKITEM`, выделенный в невыгружаемом пуле. Затем драйвер инициализирует рабочий элемент и ставит его в очередь, вызывая `IoQueueWorkItem`:

```
PIO_WORKITEM IoAllocateWorkItem(
  _In_  PDEVICE_OBJECT DeviceObject
);

VOID IoQueueWorkItem(
  _In_      PIO_WORKITEM IoWorkItem,
  _In_      PIO_WORKITEM_ROUTINE WorkerRoutine,
  _In_      WORK_QUEUE_TYPE QueueType,
  _In_opt_  PVOID Context
);
```

В ходе инициализации просто заполняются рабочая процедура, параметр/контекст и приоритет/тип очереди:

```
IO_WORKITEM_ROUTINE WorkItem;

VOID WorkItem(
  _In_      PDEVICE_OBJECT DeviceObject,
  _In_opt_  PVOID Context
)
{ ... }

typedef enum _WORK_QUEUE_TYPE {
  CriticalWorkQueue       = 0,
  DelayedWorkQueue        = 1,
  HyperCriticalWorkQueue  = 2,
  MaximumWorkQueue        = 3
} WORK_QUEUE_TYPE;
```

Куда он помещается в очередь? Как объяснялось ранее, с каждым процессором связана KPRCB, содержащая поле `ParentNode`, которое является указателем на структуру `KNODE`; при инициализации процессора этот указатель указывает на структуру `ENODE`, хранящую очередь рабочих элементов:

`Очередь рабочих элементов`

```
0: kd> dt nt!_KPRCB
...
   +0x5338 ParentNode       : Ptr64 _KNODE
0: kd> dt nt!_KNODE
   +0x000 DeepIdleSet      : Uint8B
   +0x040 ProximityId      : Uint4B
   +0x044 NodeNumber       : Uint2B
0: kd> dt nt!_ENODE
   +0x000 Ncb              : _KNODE
   +0x0c0 ExWorkerQueues   : [7] _EX_WORK_QUEUE
   +0x2f0 ExpThreadSetManagerEvent : _KEVENT
   +0x308 ExpWorkerThreadBalanceManagerPtr : Ptr64 _ETHREAD
   +0x310 ExpWorkerSeed    : Uint4B
   +0x314 ExWorkerFullInit : Pos 0, 1 Bit
   +0x314 ExWorkerStructInit : Pos 1, 1 Bit
   +0x314 ExWorkerFlags    : Uint4B
0: kd> dt nt!_EX_WORK_QUEUE
   +0x000 WorkerQueue      : _KQUEUE
   +0x040 WorkItemsProcessed : Uint4B
   +0x044 WorkItemsProcessedLastPass : Uint4B
   +0x048 ThreadCount      : Int4B
   +0x04c TryFailed        : UChar
```

`ExQueueWorkItemEx`

```
ExQueueWorkItemEx proc near
...
mov     rax, gs:20h
mov     r8, [rax+5338h]       ; enode
movzx   eax, word ptr [r8+44h]
mov     ecx, eax
lea     rax, [rax+rax*2]
shl     rax, 6
add     rax, rbp
...
mov     edx, r9d   ; тип очереди
mov     rcx, r11   ; переданный workitem
call    ExpQueueWorkItemNode
```

В действительности у каждого процессора имеется несколько очередей для хранения рабочих элементов, а системный поток извлекает из очереди по одному элементу за раз для выполнения. Этим системным потоком, отвечающим за извлечение из очереди, является `ExpWorkerThread`.

Как объяснялось ранее, рабочие элементы легковесны, поскольку не требуют создания новых объектов потоков. Кроме того, они обладают двумя важными свойствами:

• Они выполняются в контексте процесса `System`. Причина в том, что `ExpWorkerThread` работает в процессе `System`.

• Они выполняются на `PASSIVE_LEVEL`.

Благодаря легковесной природе рабочих элементов распространенным шаблоном программирования драйверов является постановка рабочих элементов в очередь внутри DPC.

Упражнения

1. Объясните, как нам удалось определить, что `ExpWorkerThread` - системный поток, отвечающий за извлечение рабочих элементов из очереди и их выполнение. Подсказка: самый быстрый способ - написать драйвер.

2. Исследуйте `IoAllocateWorkItem`, `IoInitializeWorkItem`, `IoQueueWorkItem`, `IopQueueWorkItemProlog` и `ExQueueWorkItem` и объясните, как они работают.

3. Рабочие элементы и системные потоки (то есть созданные `PsCreateSystemThread`) по функциональности в основном одинаковы, поэтому объясните, почему DPC часто ставят рабочие элементы в очередь для обработки запросов, но никогда не вызывают `PsCreateSystemThread`.

4. Напишите драйвер, перечисляющий все рабочие элементы в системе, и объясните проблемы, которые вам пришлось преодолеть.

Асинхронные вызовы процедур

Асинхронные вызовы процедур (asynchronous procedure calls, APC) используются для реализации многих важных операций, например завершения асинхронного ввода-вывода, приостановки потока и завершения процесса. К сожалению, с точки зрения ядра они не документированы. Официальная документация по разработке драйверов содержит лишь короткий раздел, признающий существование APC и наличие различных их типов. Однако для обычных задач обратного анализа понимать все нижележащие подробности не требуется. В этом разделе объясняется, что такое APC и как они обычно применяются.

Основы APC

Вообще говоря, APC - это функции, выполняющиеся в контексте определенного потока. Их можно разделить на два типа: режима ядра и пользовательского режима. APC режима ядра могут быть обычными или специальными; обычные выполняются на PASSIVE_LEVEL, а специальные - на APC_LEVEL (оба вида выполняются в режиме ядра). Пользовательские APC выполняются на PASSIVE_LEVEL в пользовательском режиме, когда поток находится в состоянии готовности к оповещению. Поскольку APC работают в контексте потока, они всегда связаны с объектом ETHREAD.

Если говорить конкретно, APC определяется структурой KAPC:

```
1: kd> dt nt!_KAPC
   +0x000 Type             : UChar
   +0x001 SpareByte0       : UChar
   +0x002 Size             : UChar
   +0x003 SpareByte1       : UChar
   +0x004 SpareLong0       : Uint4B
   +0x008 Thread           : Ptr32 _KTHREAD
   +0x00c ApcListEntry     : _LIST_ENTRY
   +0x014 KernelRoutine    : Ptr32     void
   +0x018 RundownRoutine   : Ptr32     void
   +0x01c NormalRoutine    : Ptr32     void
   +0x014 Reserved         : [3] Ptr32 Void
   +0x020 NormalContext    : Ptr32 Void
   +0x024 SystemArgument1  : Ptr32 Void
   +0x028 SystemArgument2  : Ptr32 Void
   +0x02c ApcStateIndex    : Char
   +0x02d ApcMode          : Char
   +0x02e Inserted         : UChar
```

Эта структура инициализируется API KeInitializeApc:

KeInitializeApc

```
NTKERNELAPI VOID KeInitializeApc(
    PKAPC Apc,
    PKTHREAD Thread,
    KAPC_ENVIRONMENT Environment,
    PKKERNEL_ROUTINE KernelRoutine,
    PKRUNDOWN_ROUTINE RundownRoutine,
    PKNORMAL_ROUTINE NormalRoutine,
    KPROCESSOR_MODE ProcessorMode,
    PVOID NormalContext
    );

NTKERNELAPI BOOLEAN KeInsertQueueApc(
    PRKAPC Apc,
    PVOID SystemArgument1,
    PVOID SystemArgument2,
    KPRIORITY Increment
    );
```

Прототипы функций обратного вызова

```
typedef VOID (*PKKERNEL_ROUTINE)(
    PKAPC Apc,
    PKNORMAL_ROUTINE *NormalRoutine,
    PVOID *NormalContext,
    PVOID *SystemArgument1,
    PVOID *SystemArgument2
    );

typedef VOID (*PKRUNDOWN_ROUTINE)(
    PKAPC Apc
    );

typedef VOID (*PKNORMAL_ROUTINE)(
    PVOID NormalContext,
    PVOID SystemArgument1,
```

```
    PVOID SystemArgument2
    );

typedef enum _KAPC_ENVIRONMENT {
    OriginalApcEnvironment,
    AttachedApcEnvironment,
    CurrentApcEnvironment,
    InsertApcEnvironment
} KAPC_ENVIRONMENT, *PKAPC_ENVIRONMENT;
```

> **Примечание.** Это определение взято со страницы forum.sysinternals.com/howto-capture-kernel-stack-traces_topic19356.html. Хотя мы не можем гарантировать его правильность, известно, что в экспериментах оно работало.

`Apc` - выделенный вызывающей стороной буфер типа `KAPC`. На практике он обычно выделяется в невыгружаемом пуле функцией `ExAllocatePool` и освобождается в процедуре ядра или обычной процедуре. `Thread` - поток, в очередь которого следует поместить этот APC. `Environment` определяет среду, в которой выполняется APC; например, `OriginalApcEnvironment` означает, что APC будет работать в контексте процесса потока (если тот не присоединится к другому процессу). `KernelRoutine` - функция, выполняемая на `APC_LEVEL` в режиме ядра; `RundownRoutine` - функция, выполняемая при завершении потока; `NormalRoutine` - функция, выполняемая на `PASSIVE_LEVEL` в режиме `ProcessorMode`. Пользовательскими APC являются те, у которых задана `NormalRoutine`, а `ProcessorMode` имеет значение `UserMode`. `NormalContext` - параметр, передаваемый `NormalRoutine`.

После инициализации APC ставится в очередь с помощью API `KeInsertQueueApc`. `Apc` - APC, инициализированный `KeInitializeApc`. `SystemArgument1` и `SystemArgument2` - необязательные аргументы, которые можно передать процедурам ядра и обычным процедурам. `Increment` - величина увеличения приоритета времени выполнения; она аналогична параметру `PriorityBoost` функции `IoCompleteRequest`. Куда ставится APC?

Напомним, что APC всегда связаны с потоком. Структура `KTHREAD` имеет две очереди APC:

```
0: kd> dt nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : Ptr64 Void
   +0x020 QuantumTarget    : Uint8B
...
   +0x090 TrapFrame        : Ptr64 _KTRAP_FRAME
   +0x098 ApcState         : _KAPC_STATE
   +0x098 ApcStateFill     : [43] UChar
   +0x0c3 Priority         : Char
   +0x288 SchedulerApc     : _KAPC
...
   +0x2e0 SuspendEvent     : _KEVENT
0: kd> dt nt!_KAPC_STATE
   +0x000 ApcListHead      : [2] _LIST_ENTRY
   +0x020 Process          : Ptr64 _KPROCESS
   +0x028 KernelApcInProgress : UChar
   +0x029 KernelApcPending : UChar
   +0x02a UserApcPending   : UChar
```

Поле `ApcState` содержит массив из двух очередей, в которых соответственно хранятся APC режима ядра и пользовательского режима.

Реализация приостановки потока с помощью APC

Когда программа хочет приостановить поток, ядро ставит в очередь этого потока APC режима ядра. Этот APC приостановки является полем `SchedulerApc` структуры `KTHREAD`; он инициализируется в `KeInitThread`, причем обычной процедурой служит `KiSchedulerApc`. `KiSchedulerApc` просто удерживает событие `SuspendEvent` потока. Когда программа хочет возобновить поток, `KeResumeThread` освобождает это событие.

Если вы не проводите обратный анализ ядра Windows или руткитов режима ядра, то вряд ли встретите код, использующий APC. Главная причина состоит в том, что они не документированы, а потому редко применяются в коммерческих драйверах. Однако APC часто используются в руткитах, поскольку предоставляют удобный способ внедрить код из режима ядра в пользовательский режим. Руткиты достигают этого, помещая пользовательский APC в очередь потока того процесса, куда они хотят внедрить код.

Упражнения

1. Напишите драйвер, использующий APC как режима ядра, так и пользовательского режима.

2. Напишите драйвер, перечисляющий все APC пользовательского режима и режима ядра для всех потоков процесса. Подсказка: при перечислении необходимо учитывать уровень IRQL.
3. Функция ядра `KeSuspendThread` отвечает за приостановку потока. Ранее вы узнали, что в Windows 8 в приостановке потока участвуют APC. Объясните, как работает эта функция и как APC используются для реализации этой возможности в Windows 7. Чем она отличается от Windows 8?

4. APC также используются при завершении процесса. Объект `KTHREAD` имеет флаг `ApcQueueable`, определяющий, можно ли ставить APC в его очередь. Что происходит, когда вы запрещаете постановку APC в очередь потока? Поэкспериментируйте с этим, запустив `notepad.exe`, а затем вручную запретив постановку APC в очередь одного из его потоков (сделайте это с помощью отладчика ядра).

5. Объясните, что делают следующие функции:

• `KiInsertQueueApc`

• `PsExitSpecialApc`

• `PspExitApcRundown`

• `PspExitNormalApc`

• `PspQueueApcSpecialApc`

• `KiDeliverApc`

6. Объясните, как работает функция `KeEnumerateQueueApc`, а затем восстановите ее прототип. Примечание: эта функция доступна только в Windows 8.

7. Объясните, как ядро диспетчеризует APC. Напишите драйвер, использующий различные виды APC, и просмотрите стек во время их выполнения. Примечание: тот же метод мы использовали, чтобы выяснить, как ядро диспетчеризует рабочие элементы.

Отложенные вызовы процедур

Отложенные вызовы процедур (deferred procedure calls, DPC) - это процедуры, выполняемые на DISPATCH_LEVEL в произвольном контексте потока на определенном процессоре. Аппаратные драйверы используют их для обработки прерываний, поступающих от устройства. Обычный шаблон использования состоит в том, что процедура обслуживания прерывания (interrupt service routine, ISR) ставит DPC в очередь, а DPC, в свою очередь, ставит в очередь рабочий элемент для выполнения обработки.

Аппаратные драйверы делают это потому, что ISR обычно работает на высоких IRQL (выше DISPATCH_LEVEL), и если она выполняется слишком долго, общая производительность системы может снизиться. Поэтому ISR обычно ставит DPC в очередь и немедленно возвращает управление, чтобы система могла обрабатывать другие прерывания. Программные драйверы могут использовать DPC для быстрого выполнения коротких задач.

Внутренне DPC определяется структурой KDPC:

```
0: kd> dt nt!_KDPC
   +0x000 Type             : UChar
   +0x001 Importance       : UChar

   +0x002 Number           : Uint2B
   +0x008 DpcListEntry     : _LIST_ENTRY
   +0x018 DeferredRoutine  : Ptr64     void
   +0x020 DeferredContext  : Ptr64 Void
   +0x028 SystemArgument1  : Ptr64 Void
   +0x030 SystemArgument2  : Ptr64 Void
   +0x038 DpcData          : Ptr64 Void
```

Семантика каждого поля такова:

• Type - тип объекта. Он обозначает тип объекта ядра (например, процесс, поток, таймер, DPC, событие и т. д.). Напомним, что объекты ядра определяются перечислением nt!_KOBJECTS. В данном случае мы имеем дело с DPC, которые бывают двух типов: обычные и потоковые.

• Importance - важность DPC. Она определяет положение элемента DPC в очереди DPC. См. также KeSetImportanceDpc.

• Number - номер процессора, в очередь которого следует поставить и на котором следует выполнить DPC. См. также KeSetTargetProcessorDpc.

• DpcListEntry - LIST_ENTRY элемента DPC. Внутренние операции вставки DPC в очередь DPC и удаления из нее работают с этим полем. См. KeInsertQueueDpc.

• DeferredRoutine - функция, связанная с данным DPC. Она будет выполнена в произвольном контексте потока и на DISPATCH_LEVEL. Она определяется следующим образом:

```
KDEFERRED_ROUTINE CustomDpc;
VOID CustomDpc(
  _In_      struct _KDPC *Dpc,
  _In_opt_  PVOID DeferredContext,
  _In_opt_  PVOID SystemArgument1,
  _In_opt_  PVOID SystemArgument2
)
{ ... }
```

• DeferredContext - параметр, передаваемый функции DPC.

• SystemArgument1 - пользовательские данные для хранения в DPC.

• SystemArgument2 - пользовательские данные для хранения в DPC.

• DpcData - указатель на структуру KDPC_DATA:

```
0: kd> dt nt!_KDPC_DATA
   +0x000 DpcListHead      : _LIST_ENTRY
   +0x010 DpcLock          : Uint8B

  +0x018 DpcQueueDepth    : Int4B
  +0x01c DpcCount         : Uint4B
```

Как видите, в ней хранятся учетные сведения о DPC. Данные хранятся в поле DpcData структуры KPRCB, связанной с DPC. DpcListHead - головной элемент очереди DPC (он задается при инициализации KPRCB), а DpcLock - спин-блокировка, защищающая эту структуру; при каждой постановке DPC в очередь значения DpcCount и DpcQueueDepth увеличиваются на единицу. См. также KeInsertQueueDpc. Анализ KeInsertQueueDpc в ассемблере может быть поучительным; обратите внимание на обращение к KPRCB и вставку в начало/конец списка.

Шаблон использования DPC в коде прост: инициализировать объект KDPC с помощью KeInitializeDpc и поставить его в очередь с помощью KeInsertQueueDpc. Когда IRQL процессора опускается до DISPATCH_LEVEL, ядро обрабатывает все DPC в этой очереди.

Как упоминалось ранее, каждое ядро ЦП поддерживает собственную очередь DPC. Эта очередь отслеживается отдельной для каждого ядра структурой KPRCB:

```
0: kd> dt nt!_KPRCB
   +0x000 MxCsr             : Uint4B
   +0x004 LegacyNumber      : UChar
   +0x005 ReservedMustBeZero : UChar
   +0x006 InterruptRequest : UChar
   ...
   +0x2d80 DpcData          : [2] _KDPC_DATA
   +0x2dc0 DpcStack         : Ptr64 Void
   +0x2dc8 MaximumDpcQueueDepth : Int4B
   +0x2dcc DpcRequestRate   : Uint4B
   +0x2dd0 MinimumDpcRate   : Uint4B
   +0x2dd4 DpcLastCount     : Uint4B
   +0x2dd8 ThreadDpcEnable  : UChar
   +0x2dd9 QuantumEnd       : UChar
   +0x2dda DpcRoutineActive : UChar
0: kd> dt nt!_KDPC_DATA
   +0x000 DpcListHead      : _LIST_ENTRY
   +0x010 DpcLock          : Uint8B
   +0x018 DpcQueueDepth    : Int4B
   +0x01c DpcCount         : Uint4B
```

Два примечательных поля - DpcData и DpcStack. DpcData - массив структур KDPC_DATA, каждый элемент которого отслеживает очередь DPC; первый элемент отслеживает обычные DPC, а второй - потоковые DPC. Функция KeInsertQueueDpc просто вставляет DPC в одну из этих двух очередей. Связь показана на рисунке 3-7.

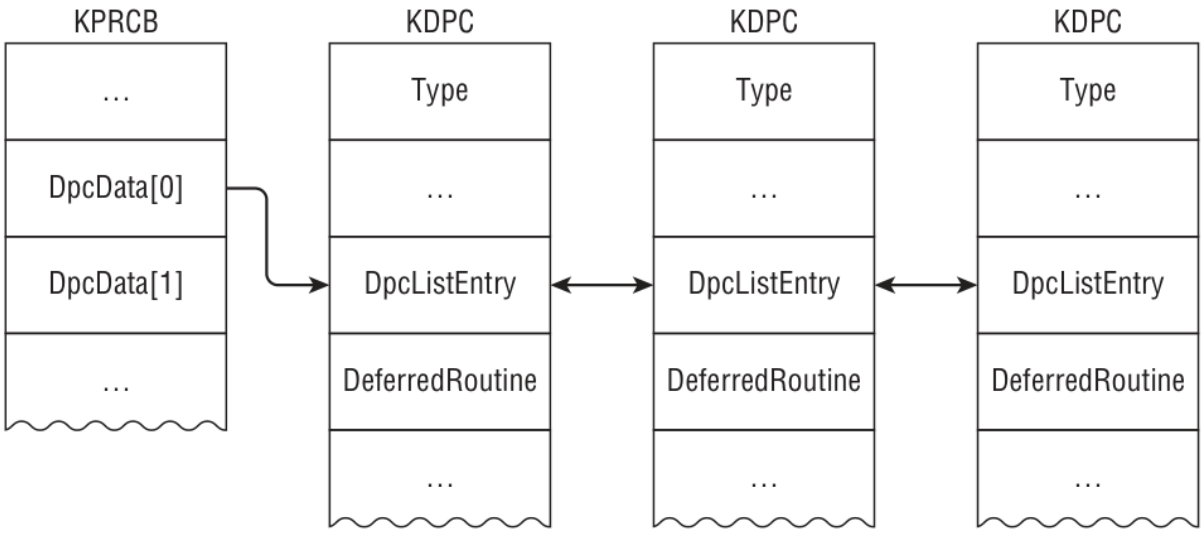

Figure 3-7

Рисунок 3-7. Связь KPRCB с очередью объектов KDPC

`DpcStack` - указатель на блок памяти, используемый как стек процедуры DPC.

В Windows имеется несколько механизмов обработки очереди DPC. Первый механизм использует `KiIdleLoop`. Во время «простоя» она проверяет PRCB, определяя, ожидают ли DPC, и если да, вызывает `KiRetireDpcList` для обработки всех DPC. Именно поэтому во время выполнения DPC эти две функции иногда присутствуют в стеке. Например:

```
0: kd> kn
 # Child-SP          RetAddr           Call Site
00 fffff800`00b9cc88 fffff800`028db5dc USBPORT!USBPORT_IsrDpc
01 fffff800`00b9cc90 fffff800`028d86fa nt!KiRetireDpcList+0x1bc
02 fffff800`00b9cd40 00000000`00000000 nt!KiIdleLoop+0x5a
```

Второй механизм действует, когда ЦП находится на `DISPATCH_LEVEL`. Рассмотрим следующий стек:

```
0: kd> kn
# Child-SP          RetAddr           Call Site
00 fffff800`00ba2ef8 fffff800`028db5dc USBPORT!USBPORT_IsrDpc
01 fffff800`00ba2f00 fffff800`028d6065 nt!KiRetireDpcList+0x1bc
02 fffff800`00ba2fb0 fffff800`028d5e7c nt!KyRetireDpcList+0x5
03 fffff880`04ac67a0 fffff800`0291b793 nt!KiDispatchInterruptContinue
04 fffff880`04ac67d0 fffff800`028cbda2 nt!KiDpcInterruptBypass+0x13
05 fffff880`04ac67e0 fffff960`0002992c nt!KiInterruptDispatch+0x212
06 fffff880`04ac6978 fffff960`000363b3 win32k!vAlphaPerPixelOnly+0x7c
07 fffff880`04ac6980 fffff960`00035fa4 win32k!AlphaScanLineBlend+0x303
08 fffff880`04ac6a40 fffff960`001fd4f9 win32k!EngAlphaBlend+0x4f4
09 fffff880`04ac6cf0 fffff960`001fdbaa win32k!NtGdiUpdateTransform+0x112d
0a fffff880`04ac6db0 fffff960`001fdd19 win32k!NtGdiUpdateTransform+0x17de
0b fffff880`04ac6ed0 fffff960`001fded8 win32k!EngNineGrid+0xb1
0c fffff880`04ac6f70 fffff960`001fe395 win32k!EngDrawStream+0x1a0

0d fffff880`04ac7020 fffff960`001fece7 win32k!NtGdiDrawStreamInternal+0x47d
0e fffff880`04ac70d0 fffff960`0021a480 win32k!GreDrawStream+0x917
0f fffff880`04ac72c0 fffff800`028cf153 win32k!NtGdiDrawStream+0x9c
10 fffff880`04ac7420 000007fe`fd762cda nt!KiSystemServiceCopyEnd+0x13
```

Этот длинный стек указывает, что `win32k.sys` обрабатывал пользовательский запрос некоторой графической операции, после чего была выполнена процедура DPC драйвера USB-порта, не имеющая никакого отношения к win32k. Вероятно, пока `win32k.sys` обрабатывал запрос, возникло прерывание устройства, заставившее ЦП работать на IRQL устройства; затем IRQL в конечном итоге был понижен до `DISPATCH_LEVEL`, что вызвало обработку очереди DPC.

Третий механизм использует системный поток, создаваемый при инициализации процессора. `KiStartDpcThread` создает для каждого процессора поток (`KiExecuteDpc`), который обрабатывает очередь DPC при каждом своем запуске. Например:

```
0: kd> kn
 # Child-SP          RetAddr           Call Site
00 fffff880`03116be8 fffff800`028aadb0 nt!KiDpcWatchdog
01 fffff880`03116bf0 fffff800`028aac4b nt!KiExecuteAllDpcs+0x148
02 fffff880`03116cb0 fffff800`02b73166 nt!KiExecuteDpc+0xcb
03 fffff880`03116d00 fffff800`028ae486 nt!PspSystemThreadStartup+0x5a
04 fffff880`03116d40 00000000`00000000 nt!KiStartSystemThread+0x16
```

Напомним, что диспетчер потоков работает на `DISPATCH_LEVEL`, а код на этом IRQL не может быть прерван другими программными IRQL (то есть уровнями ниже `DISPATCH_LEVEL`). Иными словами, если в процедуре DPC имеется бесконечный цикл, связанный с ней процессор будет вращаться бесконечно, и система практически «зависнет»; в многопроцессорной системе она может не зависнуть, но процессор, выполняющий DPC, станет недоступен диспетчеру потоков. Кроме того, процедура DPC не может ожидать никаких объектов диспетчера, поскольку сам диспетчер работает на `DISPATCH_LEVEL`; именно поэтому функции вроде `KeWaitForSingleObject` и `KeDelayExecutionThread` нельзя вызывать в процедурах DPC.

> **Примечание.** В Windows имеется сторожевая процедура DPC, обнаруживающая DPC, которые выполняются дольше определенного времени, и вызывающая bugcheck с кодом `DPC_WATCHDOG_VIOLATION` (`0x133`). Значение сторожевого таймера можно запросить, вызвав `KeQueryDpcWatchdogInformation`.

Некоторые руткиты используют DPC для синхронизации доступа к глобальным связанным спискам. Например, они могут удалить элемент из списка `ActiveProcessLinks`, чтобы скрыть процессы; поскольку этот список может быть изменен в любой момент любым процессором, некоторые авторы руткитов применяют DPC вместе с другим механизмом синхронизации для безопасной работы с ним. В одном из упражнений вам будет предложено объяснить, почему одним авторам это удается, а другие терпят неудачу (машина выполняет bugcheck).

Упражнения

1. Где и когда инициализируется поле `DpcData` в KPRCB?

2. Напишите драйвер, перечисляющий все DPC во всей системе. Обязательно поддержите многопроцессорные системы! Объясните возникшие трудности и способы их решения.

3. Объясните, как работает процедура `KiDpcWatchdog`.

Таймеры

Таймеры используются для сигнализации об истечении определенного периода времени: периодически или в некоторый момент в будущем. При желании таймер также можно связать с DPC. Например, если драйвер хочет проверять состояние устройства каждые пять минут или выполнить процедуру через 10 минут, он может сделать это с помощью таймеров.

Если говорить конкретно, таймер определяется структурой `KTIMER`:

`Структуры, связанные с таймерами`

```
0: kd> dt nt!_KPRCB
...
   +0x2dfc InterruptRate    : Uint4B
   +0x2e00 TimerTable       : _KTIMER_TABLE
0: kd> dt nt!_KTIMER_TABLE
   +0x000 TimerExpiry      : [64] Ptr64 _KTIMER
   +0x200 TimerEntries     : [256] _KTIMER_TABLE_ENTRY
0: kd> dt nt!_KTIMER
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 DueTime          : _ULARGE_INTEGER
   +0x020 TimerListEntry   : _LIST_ENTRY
   +0x030 Dpc              : Ptr64 _KDPC
   +0x038 Processor        : Uint4B
   +0x03c Period           : Uint4B
0: kd> dt nt!_KTIMER_TABLE_ENTRY
   +0x000 Lock             : Uint8B
   +0x008 Entry            : _LIST_ENTRY
   +0x018 Time             : _ULARGE_INTEGER
```

Процедуры, связанные с таймерами

```
VOID KeInitializeTimer(
  _Out_  PKTIMER Timer
);

BOOLEAN KeSetTimer(
  _Inout_   PKTIMER Timer,

  _In_      LARGE_INTEGER DueTime,
  _In_opt_  PKDPC Dpc
);

BOOLEAN KeSetTimerEx(
  _Inout_   PKTIMER Timer,
  _In_      LARGE_INTEGER DueTime,
  _In_      LONG Period,
  _In_opt_  PKDPC Dpc
);
```

Таймер инициализируется вызовом `KeInitializeTimer`, который просто заполняет некоторые основные поля. После инициализации таймер можно задать с помощью `KeSetTimer` или `KeSetTimerEx`. Разница между ними состоит в том, что `KeSetTimerEx` можно использовать для задания повторяющегося таймера (то есть срабатывающего через каждые X единиц времени). Обратите внимание, что эти функции могут принимать необязательный объект DPC, выполняемый при срабатывании таймера. При вызове этих процедур таймер вставляется в таблицу таймеров в PRCB (`TimerTable->TimerListEntry`). После задания и постановки в очередь таймер можно отменить и тем самым удалить из таблицы таймеров. Это выполняется API `KeCancelTimer`.

Как система узнает, что таймер сработал? При каждом прерывании часов система обновляет свое время выполнения и проверяет список таймеров на наличие срабатывающих элементов; если такие элементы есть, она запрашивает прерывание DPC, которое их обработает. Следовательно, таймеры также обрабатываются на `DISPATCH_LEVEL`.

В операционной системе имеется множество примеров использования таймеров. Например, в системе есть периодический таймер, который синхронизирует системное время и проверяет, не истекает ли срок лицензии (см. `ExpTimeRefreshDpcRoutine`). Существует даже таймер, срабатывающий в конце столетия (см. `ExpCenturyDpcRoutine`).

Упражнения

1. Напишите драйвер, перечисляющий список загруженных модулей каждые 10 минут.

2. Напишите драйвер, перечисляющий все таймеры в системе. Обязательно поддержите многоядерные системы. Объясните, почему данные DPC, связанные с таймером, кажутся бессмысленными.

3. Объясните поле `DpcWatchDogTimer` в PRCB.

4. Напишите драйвер, задающий таймер со связанным DPC. Объясните последовательность вызовов, ведущую к выполнению DPC. Вас могут заинтересовать следующие функции: `KeUpdateRuntime`, `KeAccumulateTicks`, `KiTimerExpiration`, `KiRetireDpcList` и `KiExpireTimerTable`.

5. Объясните, как работает вставка таймера. Вам потребуется изучить функцию `KiInsertTimerTable`.

Функции обратного вызова процессов и потоков

Драйвер может регистрировать функции обратного вызова для различных событий. Два наиболее распространенных типа связаны с процессами и потоками; их можно регистрировать с помощью документированных API, таких как `PsSetCreateProcessNotifyRoutine`, `PsSetCreateThreadNotifyRoutine` и `PsSetLoadImageNotifyRoutine`. Как они работают?

Во время инициализации системы ядро вызывает функцию `PspInitializeCallbacks`, чтобы инициализировать три глобальных массива: `PspCreateThreadNotifyRoutine`, `PspCreateProcessNotifyRoutine` и `PspLoadImageNotifyRoutine`. Когда драйвер регистрирует функцию обратного вызова процесса, потока или образа, она сохраняется в одном из этих массивов. Кроме того, имеется глобальный флаг `PspNotifyEnableMask`, определяющий, какие типы уведомлений включены или отключены. На путях инициализации и завершения потока (`PspInsertThread` и `PspExitThread` соответственно) проверяется наличие флага `PspNotifyEnableMask`, после чего соответствующим образом вызываются функции обратного вызова.

Эти функции обратного вызова предоставляются главным образом для драйверов, поэтому явно ядром не используются. Например, многие антивирусные продукты регистрируют их для наблюдения за поведением системы. Руткиты режима ядра иногда используют их совместно с APC для внедрения кода в новые процессы.

Упражнения

1. В этом разделе дано общее объяснение реализации функций обратного вызова уведомлений о процессах, потоках и образах. Исследуйте следующие функции и объясните, как они работают:

• `PsSetCreateThreadNotifyRoutine`

• `PsSetCreateProcessNotifyRoutine`

• `PsSetLoadImageNotifyRoutine`

• `PspInitializeCallbacks`

2. Если вы выполнили упражнение 1, напишите драйвер, перечисляющий все процедуры уведомления о процессах, потоках и образах в системе, и удалите их.

3. Если вы выполнили упражнение 1, объясните два основных недостатка этих функций обратного вызова уведомлений. Например, можно ли создавать новые процессы/потоки так, чтобы эти функции обратного вызова их не обнаружили? Реализуйте свою идею и оцените ее эффективность. Примечание: это возможно.

4. Если зарегистрировать функцию обратного вызова загрузки образа с помощью `PsSetLoadImageNotifyRoutine`, при каком условии она вызывается? Найдите один недостаток и реализуйте свою идею. Подсказка: возможно, потребуется обратиться к спецификации PE.
5. API `PsSetCreateThreadNotifyRoutine`, `PsSetCreateProcessNotifyRoutine` и `PsSetLoadImageNotifyRoutine` предоставляются диспетчером процессов. Однако диспетчеры объектов и конфигурации также предоставляют собственные функции обратного вызова через `ObRegisterCallbacks` и `CmRegisterCallback` соответственно. Исследуйте реализацию этих функций обратного вызова.

6. Найдите другие похожие функции обратного вызова, документированные в WDK, и исследуйте их работу (процессор, память и так далее).

Процедуры завершения

Модель ввода-вывода Windows представляет собой стек устройств, в котором устройства наслаиваются друг на друга, а каждый слой реализует некоторую определенную функцию. Это означает, что драйверы более высокого уровня могут передавать запросы нижележащим драйверам для обработки. Слой, завершающий запрос, помечает его выполненным, вызывая `IoCompleteRequest`. Процедуры завершения используются для уведомления драйверов о том, что их запрос ввода-вывода завершен (либо отменен или завершился ошибкой). Они работают в произвольном контексте потока и могут задаваться с помощью API `IoSetCompletionRoutine/Ex`. `IoSetCompletionRoutine` документирована в WDK, но никогда не появится в ассемблерном листинге или таблице импорта, поскольку принудительно встраивается; один из способов обнаружить процедуру `IoSetCompletion` - увидеть изменение поля `CompletionRoutine` в `IO_STACK_LOCATION` (см. следующий раздел):

`Определение структуры`

```
0: kd> dt nt!_IO_STACK_LOCATION
   +0x000 MajorFunction    : UChar
   +0x001 MinorFunction    : UChar
   +0x002 Flags            : UChar
   +0x003 Control          : UChar
   +0x008 Parameters       : <unnamed-tag>
   +0x028 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x030 FileObject       : Ptr64 _FILE_OBJECT
   +0x038 CompletionRoutine : Ptr64     long
   +0x040 Context          : Ptr64 Void
```

`Определение функции`

```
VOID
IoSetCompletionRoutine(
    _In_ PIRP Irp,
    _In_opt_ PIO_COMPLETION_ROUTINE CompletionRoutine,
    _In_opt_ __drv_aliasesMem PVOID Context,
    _In_ BOOLEAN InvokeOnSuccess,
    _In_ BOOLEAN InvokeOnError,
    _In_ BOOLEAN InvokeOnCancel
    )
{
    PIO_STACK_LOCATION irpSp;

    irpSp = IoGetNextIrpStackLocation(Irp);
    irpSp->CompletionRoutine = CompletionRoutine;
```

```
    irpSp->Context = Context;
    irpSp->Control = 0;
    if (InvokeOnSuccess) {
        irpSp->Control = SL_INVOKE_ON_SUCCESS;
    }
    if (InvokeOnError) {
        irpSp->Control |= SL_INVOKE_ON_ERROR;
    }
    if (InvokeOnCancel) {
        irpSp->Control |= SL_INVOKE_ON_CANCEL;
    }
}
```

Диспетчер ввода-вывода вызывает зарегистрированную процедуру завершения как часть `IopfCompleteRequest`.

Хотя законное применение процедур завершения очевидно, руткиты могут использовать их в злонамеренных целях. Например, они могут задать процедуру завершения, изменяющую возвращаемый буфер нижележащего драйвера до его возврата в пользовательский режим.

Упражнение

1. Напишите тестовый драйвер, использующий процедуру завершения, и определите, откуда она вызывается.

Пакеты запросов ввода-вывода

Windows использует пакеты запросов ввода-вывода (I/O request packets, IRP) для описания запросов ввода-вывода к компонентам режима ядра (например, драйверам). Когда приложение пользовательского режима вызывает API для запроса данных, диспетчер ввода-вывода создает IRP, описывающий запрос, и определяет, какому устройству отправить IRP для обработки. С момента создания IRP до его завершения драйвером он может пройти через несколько устройств, а для выполнения запроса могут быть созданы дополнительные IRP. IRP можно рассматривать как фундаментальную единицу обмена данными между устройствами для запросов ввода-вывода. В заголовках WDK IRP определяется частично непрозрачной структурой `IRP`, однако большинство ее полей не документированы (отсюда частичная непрозрачность):

```
0: kd> dt nt!_IRP
   +0x000 Type             : Int2B
...

   +0x042 StackCount       : Char
   +0x043 CurrentLocation  : Char
...
   +0x058 Overlay          : <unnamed-tag>
   +0x068 CancelRoutine    : Ptr64     void
   +0x070 UserBuffer       : Ptr64 Void
   +0x078 Tail             : <unnamed-tag>
```

С точки зрения программирования IRP можно разделить на две области: статическую и динамическую. Статическая часть - структура `IRP` с основными сведениями о запросе, например о том, кто запросил операцию (ядро или пользователь), запрашивающем потоке и данных, переданных пользователем. Поля `Overlay` и `Tail` являются объединениями, содержащими метаданные запроса. Динамическая часть находится непосредственно после заголовка; это массив структур `IO_STACK_LOCATION`, содержащих сведения о запросе для конкретных устройств. `IO_STACK_LOCATION` содержит основную и дополнительную функции IRP, параметры запроса и необязательную процедуру завершения. Как и `IRP`, это частично непрозрачная структура:

```
0: kd> dt nt!_IO_STACK_LOCATION
   +0x000 MajorFunction    : UChar
   +0x001 MinorFunction    : UChar
   +0x002 Flags            : UChar
   +0x003 Control          : UChar
   +0x008 Parameters       : <unnamed-tag>
   +0x028 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x030 FileObject       : Ptr64 _FILE_OBJECT
   +0x038 CompletionRoutine : Ptr64     long
   +0x040 Context          : Ptr64 Void
```

Поле `Parameters` является объединением, поскольку параметр зависит от номера основной и дополнительной функции. Windows имеет предопределенный список универсальных основных и дополнительных функций для описания всех типов запросов. Например, запрос чтения файла приводит к созданию IRP с основной функцией `IRP_MJ_READ`; когда Windows запрашивает ввод у драйвера класса клавиатуры, она также использует `IRP_MJ_READ`. При создании IRP диспетчер ввода-вывода определяет, сколько структур `IO_STACK_LOCATION` выделить, исходя из количества устройств в текущем стеке устройств. Каждое устройство отвечает за подготовку `IO_STACK_LOCATION` для следующего устройства. Напомним, что драйвер может задать процедуру завершения с помощью API `IoSetCompletionRoutine`; в действительности это встроенная процедура, задающая поле `CompletionRoutine` в `IO_STACK_LOCATION`.

На рисунке 3-8 показана связь между этими двумя структурами в IRP.

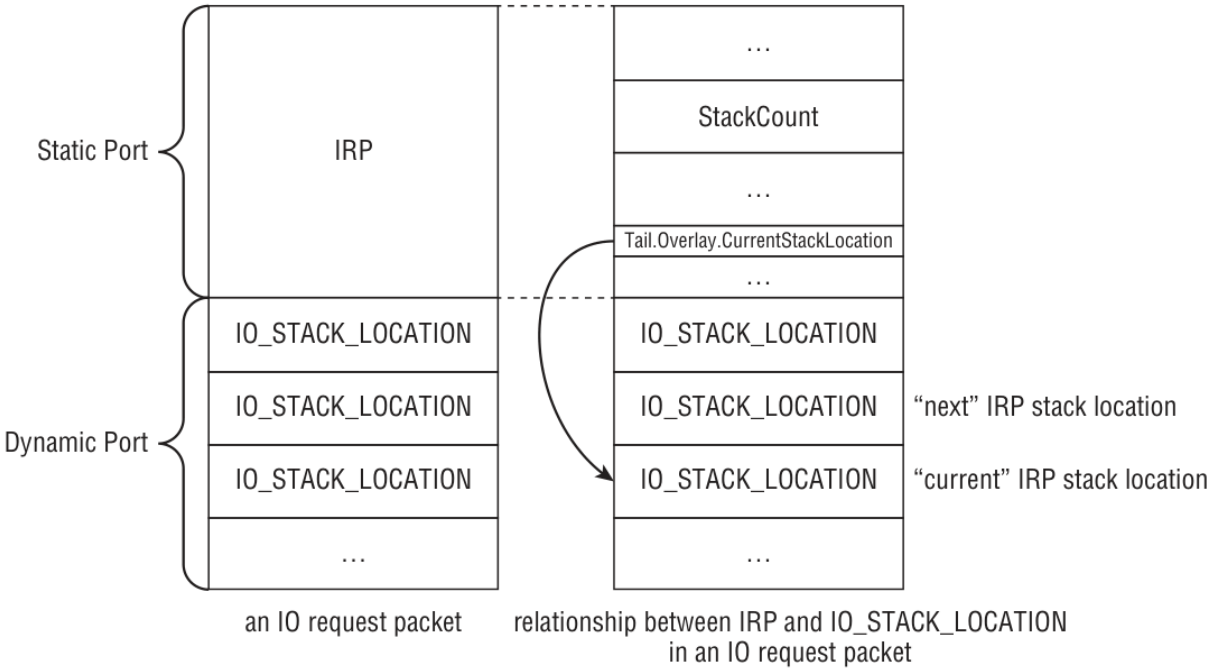

Figure 3-8

Рисунок 3-8. Статическая и динамическая части IRP и связь с IO_STACK_LOCATION

Обратите внимание, что «следующее» расположение стека является элементом непосредственно над «текущим» (а не после него). Это важно знать, поскольку процедуры расположения стека, такие как `IoGetCurrentIrpStackLocation`, `IoSkipCurrentIrpStackLocation`, `IoGetNextIrpStackLocation` и другие, просто возвращают указатели на эти элементы массива с помощью арифметики указателей.

Хотя IRP обычно создаются диспетчером ввода-вывода в ответ на запросы пользователей или других устройств, их также можно создавать с нуля и отправлять другим устройствам для обработки. Драйвер может выделить IRP с помощью `IoAllocateIrp`, связать его с потоком, заполнить основной и дополнительный коды IRP, задать количество/размер `IO_STACK_LOCATION`, заполнить параметры и отправить его целевому устройству для обработки с помощью `IoCallDriver`. Некоторые руткиты используют этот механизм, чтобы напрямую отправлять запросы драйверу файловой системы и тем самым обходить перехват системных вызовов. В упражнении вы проанализируете один такой руткит.

Структура драйвера

Драйвер - это программное обеспечение, которое взаимодействует с ядром и/или управляет аппаратными ресурсами. Хотя существует множество различных типов драйверов, нас главным образом интересуют следующие типы драйверов режима ядра:

• Устаревший программный драйвер - программное обеспечение, которое работает в кольце 0 и взаимодействует с ядром через документированные и недокументированные интерфейсы. К этому типу относится большинство руткитов и драйверов безопасности.
• Устаревший драйвер-фильтр - драйвер, который подключается к существующему драйверу и изменяет его входные данные.

• Мини-фильтр файловой системы - драйвер, который взаимодействует с файловой системой, перехватывая запросы файлового ввода-вывода. Большинство антивирусных программ использует драйверы этого типа для перехвата операций записи/чтения файлов с целью сканирования; программное обеспечение шифрования данных на диске обычно реализуется с помощью этого механизма.

Стандартной моделью драйверов Windows является Windows Driver Model (WDM). WDM определяет как набор интерфейсов, которые должны реализовывать драйверы, так и правила, которым следует подчиняться для безопасного взаимодействия с ядром. Эта модель определена со времен Windows 2000, и все анализируемые вами драйверы основаны на ней. Поскольку писать надежные аппаратные драйверы Plug and Play с полноценным управлением питанием и обработкой всех особенностей синхронизации, используя только интерфейсы WDM, чрезвычайно сложно, Microsoft представила инфраструктуру Windows Driver Foundation (WDF). По сути, WDF - это набор библиотек, построенных поверх WDM, который упрощает разработку драйверов, ограждая разработчиков от непосредственного взаимодействия с WDM. WDF делится на две категории: инфраструктуру драйверов режима ядра (kernel-mode driver framework, KMDF) и инфраструктуру драйверов пользовательского режима (user-mode driver framework, UMDF). KMDF предназначена для драйверов режима ядра (например, клавиатур и USB-устройств), а UMDF - для драйверов пользовательского режима (например, драйверов принтеров). В этой книге рассматриваются только драйверы, основанные на модели WDM.

Драйвер можно рассматривать как DLL, загружаемую в адресное пространство ядра и выполняющуюся с теми же привилегиями, что и ядро. У него есть четко определенная точка входа, и он может регистрировать процедуры диспетчеризации для обслуживания запросов пользователей или других драйверов. Обратите внимание, что у драйвера нет главного потока

выполнения; он просто содержит код, который ядро может вызывать при определенных обстоятельствах. Именно поэтому драйверам обычно приходится регистрировать процедуры диспетчеризации у диспетчера ввода-вывода (см. следующий раздел). При анализе драйверов первая и самая важная задача - найти эти процедуры диспетчеризации и понять, как они взаимодействуют с ядром.

Точки входа

Все драйверы имеют точку входа `DriverEntry`, определяемую следующим образом:

`DriverEntry`

```
NTSTATUS
DriverEntry (
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath
);
```

`DRIVER_OBJECT`

```
typedef struct _DRIVER_OBJECT {
    CSHORT Type;

    CSHORT Size;
    PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension;
    UNICODE_STRING DriverName;
    PUNICODE_STRING HardwareDatabase;
    PFAST_IO_DISPATCH FastIoDispatch;
    PDRIVER_INITIALIZE DriverInit;
    PDRIVER_STARTIO DriverStartIo;
    PDRIVER_UNLOAD DriverUnload;
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

> **Примечание.** С технической точки зрения точка входа не обязана называться `DriverEntry`.

Когда требуется загрузить драйвер, его образ отображается в память пространства ядра, для него создается объект драйвера, регистрируемый у диспетчера объектов, а затем диспетчер ввода-вывода вызывает точку входа. `DRIVER_OBJECT` - структура, заполняемая диспетчером ввода-вывода в процессе загрузки драйвера; официальная документация указывает, что это частично непрозрачная структура, однако ее полное определение можно увидеть в заголовочных файлах. Полю `DriverInit` присваивается точка входа драйвера, и диспетчер ввода-вывода непосредственно вызывает это поле. Основная обязанность `DriverEntry` - инициализировать параметры, специфичные для драйвера, и при необходимости зарегистрировать процедуры диспетчеризации IRP. Эти процедуры хранятся в массиве `MajorFunction`. Как упоминалось ранее, Windows имеет предопределенный набор основных функций IRP для универсального описания каждого запроса ввода-вывода; когда драйверу поступает запрос ввода-вывода, диспетчер ввода-вывода вызывает обработчик соответствующей основной функции IRP для обработки запроса. Поэтому в `DriverEntry` часто встречается код следующего вида:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateCloseHandler;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = CreateCloseHandler;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DeviceControlHandler;
...
```

Обратите внимание, что одну и ту же процедуру диспетчеризации можно назначить нескольким основным функциям IRP. Иногда они инициализируются в цикле:

```
for (i=0; i<IRP_MJ_MAXIMUM; i++) {
    DriverObject->MajorFunction[i] = GenericHandler;
}
DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateHandler;
DriverObject->MajorFunction[IRP_MJ_PNP] = PnpHandler;
...
```

Если таблица `MajorFunction` не инициализирована, она содержит стандартный обработчик `IopInvalidDeviceRequest`, который просто возвращает запрашивающей стороне ошибку.

Если драйвер поддерживает динамическую выгрузку, он также должен заполнить поле `DriverUnload`; в противном случае драйвер навсегда останется в памяти (до перезагрузки). Процедура `DriverUnload` обычно выполняет задачи очистки, специфичные для драйвера. Многие руткиты не регистрируют процедуру выгрузки.

`RegistryPath` - путь драйвера в реестре. Он создается как часть обычного процесса установки драйвера.

Объекты драйверов и устройств

В предыдущем разделе говорилось, что диспетчер ввода-вывода создает `DRIVER_OBJECT` для каждого драйвера, загруженного в систему. Драйвер может создать один или несколько объектов устройств. Объекты устройств определяются частично непрозрачной структурой `DEVICE_OBJECT`:

```
typedef struct _DEVICE_OBJECT {
    CSHORT Type;
    USHORT Size;
    LONG ReferenceCount;
    struct _DRIVER_OBJECT *DriverObject;
    struct _DEVICE_OBJECT *NextDevice;
    struct _DEVICE_OBJECT *AttachedDevice;
    struct _IRP *CurrentIrp;
    ...
    PVOID DeviceExtension;
    DEVICE_TYPE DeviceType;
    CCHAR StackSize;
```

```
    ...
    ULONG ActiveThreadCount;
    PSECURITY_DESCRIPTOR SecurityDescriptor;
    ...
    PVOID Reserved;
} DEVICE_OBJECT, *PDEVICE_OBJECT;
```

`DriverObject` - объект драйвера, связанный с этим объектом устройства. Если драйвер создал несколько объектов устройств, `NextDevice` указывает на следующий объект устройства в цепочке. Драйвер может создать несколько объектов устройств для управления различными обрабатываемыми им аппаратными ресурсами. Если объекты устройств не созданы, никто не сможет отправлять устройству запросы. Обычно драйверы создают объекты устройств в `DriverEntry` с помощью API `IoCreateDevice`.

`DeviceExtension` - указатель на данные конкретного устройства, хранящиеся в невыгружаемом пуле. Их размер задается параметром `IoCreateDevice`. Разработчики обычно хранят здесь сведения контекста или важные данные о драйвере и других связанных устройствах. Восстановление структуры расширения устройства, вероятно, является второй по важности задачей при анализе драйверов.

Драйвер может «подключить» один из собственных объектов устройств к другому объекту устройства, чтобы получать запросы ввода-вывода, предназначенные целевому объекту устройства. Например, если устройство A подключается к устройству B, все запросы IRP, отправленные B, сначала направляются A. Этот механизм подключения используется для поддержки драйверов-фильтров, чтобы они могли изменять или проверять запросы к другим драйверам. Поле `AttachedDevice` указывает на устройство, к которому подключен текущий объект устройства. Подключение устройств выполняется семейством API `IoAttachDevice`.

Обработка IRP

Как упоминалось ранее, `DriverEntry` обычно регистрирует процедуры диспетчеризации для обработки различных основных функций IRP. Прототип этих процедур диспетчеризации выглядит следующим образом:

```
NTSTATUS
XXX_Dispatch (
    PDEVICE_OBJECT *DeviceObject,
    PIRP *Irp
);
```

Первый аргумент - целевой объект устройства запроса. Второй аргумент - IRP, описывающий запрос.

Процедура диспетчеризации обычно сначала определяет, какую основную функцию IRP она получила, а затем определяет параметры запроса. Для этого она проверяет `IO_STACK_LOCATION` в IRP. Если процедура диспетчеризации успешно завершает запрос, она вызывает `IoCompleteRequest` и возвращает управление. Если завершить запрос невозможно, у нее есть три варианта: вернуть ошибку, передать IRP другому драйверу или оставить IRP в ожидании. Например, драйвер-фильтр может решить самостоятельно обрабатывать только запросы `IRP_MJ_READ`, а все остальные запросы передавать подключенному устройству. Драйвер может передавать IRP другому драйверу с помощью API `IoCallDriver`.

Поскольку параметры IRP для каждого запроса хранятся в собственной `IO_STACK_LOCATION`, драйвер должен убедиться, что обращается к правильному расположению. Это делается с помощью API `IoGetCurrentIrpStackLocation`. Если драйвер хочет передать тот же IRP другому драйверу, он должен либо скопировать текущие параметры в следующую `IO_STACK_LOCATION`

(IoCopyCurrentIrpStackLocationToNext), либо передать параметр следующему драйверу (IoSkipCurrentStackLocation).

Распространенный механизм обмена между пользовательским режимом и ядром

Для обмена между пользовательским режимом и ядром используется множество механизмов. Например, драйвер может взаимодействовать с кодом пользовательского режима через общую область памяти, дважды отображенную в пользовательском пространстве и пространстве ядра. Другой способ состоит в том, что драйвер создает событие, которого может ожидать поток пользовательского режима; состояние события можно использовать как триггер дальнейшего действия. Еще один (хотя и хакерский) способ основан на обработке прерываний. Драйвер может вручную установить специальный обработчик прерывания в IDT, а код пользовательского режима может инициировать его командой INT; вероятно, вы никогда не увидите этот прием в коммерческом драйвере.

Хотя точный механизм обмена зависит от конечной цели разработчика, для передачи данных между пользователем и ядром обычно используется универсальный документированный интерфейс. Этот механизм поддерживается операцией IRP_MJ_DEVICE_CONTROL и обычно называется управлением вводом-выводом устройства или просто IOCTL. Он работает следующим образом:

1. Драйвер определяет один или несколько кодов IOCTL для каждой поддерживаемой операции.

2. Для каждой поддерживаемой операции драйвер указывает, как следует обращаться к пользовательским входным данным и возвращать данные пользователю. Существует три метода доступа: буферизованный ввод-вывод, прямой ввод-вывод и метод «ни то ни другое». Эти методы рассматриваются в следующем разделе.

3. Внутри обработчика IRP_MJ_DEVICE_CONTROL драйвер получает код IOCTL из своей IO_STACK_LOCATION и обрабатывает данные в соответствии с методом ввода.

Код пользовательского режима может запрашивать эти операции IOCTL через API DeviceIoControl.

Методы буферизации

Драйвер может обращаться к буферу пользовательского режима одним из следующих трех способов:

• Буферизованный ввод-вывод - в ядре он называется METHOD_BUFFERED. При использовании этого метода ядро проверяет, что пользовательский буфер находится в доступной памяти пользовательского режима, выделяет блок памяти в невыгружаемом пуле и копирует в него пользовательский буфер. Драйвер обращается к этому буферу режима ядра через поле AssociatedIrp.SystemBuffer структуры IRP. Обрабатывая запрос, драйвер может изменять системный буфер (например, если требуется вернуть пользователю некоторые данные); после завершения запроса ядро копирует содержимое системного буфера обратно в буфер пользовательского режима и автоматически освобождает системный буфер.

• Прямой ввод-вывод - в ядре он называется METHOD_IN_DIRECT или METHOD_OUT_DIRECT. Первый вариант используется для передачи данных драйверу, второй - для получения данных от драйвера. Этот метод похож на буферизованный ввод-вывод, за исключением того, что драйвер получает MDL, описывающий пользовательский буфер. Диспетчер ввода-вывода создает MDL и закрепляет

его в памяти перед передачей драйверу. Драйверы могут обращаться к этому MDL через поле `MdlAddress` структуры IRP.

• Ни то ни другое - в ядре этот метод называется `METHOD_NEITHER`. При его использовании диспетчер ввода-вывода не выполняет никакой проверки пользовательских данных, а передает драйверу необработанные данные. Драйверы могут обращаться к данным через поле `Parameters.DeviceIoControl.Type3InputBuffer`

своей `IO_STACK_LOCATION`. Хотя этот метод может показаться самым быстрым из трех (поскольку проверка и отображение дополнительных буферов отсутствуют), он, безусловно, самый небезопасный. Вся проверка возлагается на разработчика. Без надлежащей проверки драйвер, использующий этот метод, может открыть уязвимости безопасности, например повреждение памяти ядра или утечку/раскрытие данных.

Письменного правила для выбора метода в драйверах не существует, поскольку выбор зависит от конкретных требований драйвера. Однако на практике большинство программных драйверов использует буферизованный ввод-вывод, поскольку он обеспечивает хороший баланс простоты и безопасности. Прямой ввод-вывод часто применяется в аппаратных драйверах, так как позволяет передавать большие блоки данных без накладных расходов на буферизацию.

Код управления вводом-выводом

Код IOCTL - это 32-разрядное целое число, кодирующее тип устройства, код конкретной операции, метод буферизации и доступ безопасности. Драйверы обычно определяют коды IOCTL с помощью макроса `CTL_CODE`:

```
#define CTL_CODE( DeviceType, Function, Method, Access ) (                 \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) \
)
```

`DeviceType` обычно является одной из констант `FILE_DEVICE_*`, однако сторонние драйверы могут использовать любое значение выше `0x8000`. (Это лишь рекомендуемое значение, и ничто не обеспечивает его соблюдение.) `Access` задает универсальные операции чтения/записи, разрешенные IOCTL; он может быть сочетанием `FILE_ANY_ACCESS`, `FILE_READ_ACCESS` и `FILE_WRITE_ACCESS`. `Function` - специфичный для драйвера код IOCTL; им может быть любое значение выше `0x800`. `Method` задает один из методов буферизации. Обычный способ определения кода IOCTL выглядит следующим образом:

```
#define FILE_DEVICE_GENIOCTL 0xa000 // наш тип устройства
#define GENIOCTL_PROCESS     0x800  // наш специальный код IOCTL

#define IOCTL_PROCESS CTL_CODE(FILE_DEVICE_GENIOCTL, \
                                   GENIOCTL_PROCESS, \
                                   METHOD_BUFFERED, FILE_READ_DATA)
```

Здесь для специального драйвера с методом `METHOD_BUFFERED` определяется IOCTL с именем `IOCTL_PROCESS`.

При анализе драйвера важно разложить IOCTL на тип устройства, код, доступ и метод буферизации. Это можно сделать с помощью пары простых документированных макросов:

```
#define DEVICE_TYPE_FROM_CTL_CODE(ctrlCode) \
                              (((ULONG)(ctrlCode & 0xffff0000)) >> 16)
#define METHOD_FROM_CTL_CODE(ctrlCode)      ((ULONG)(ctrlCode & 3))
```

Прочие системные механизмы

В этом разделе рассматриваются конструкции, которые, хотя и не являются необходимыми для понимания драйверов ядра, часто встречаются в настоящих драйверах.

Системные управляющие регистры

Для достижения своих целей многие разработчики руткитов прибегают к перехвату функций в ядре. Однако весь код ядра отображается только для чтения, поэтому попытка его исправления приведет к bugcheck. На x86/x64 этот механизм защиты в действительности обеспечивается на аппаратном уровне специальным управляющим регистром CR0. CR0 определяет несколько важных параметров процессора, например работает ли он в защищенном режиме и включена ли страничная адресация; кроме того, он определяет, может ли ЦП записывать данные на страницы только для чтения (бит WP). CR0 доступен только коду, выполняющемуся в кольце 0. По умолчанию Windows включает бит WP, запрещающий запись на страницы, помеченные только для чтения.

> **Примечание.** На x64 и ARM существует функция Windows под названием Kernel Patch Protection, также известная как PatchGuard, которая пытается обнаруживать перехваты и изменения различных критически важных для безопасности структур данных и выполняет bugcheck машины. Поэтому на этих платформах перехваты редко встречаются в поставляемых/производственных драйверах. Тем не менее перехват по-прежнему широко распространен, поскольку существует множество машин x86, и вы будете часто с ним сталкиваться.

Существует несколько способов обойти это ограничение, и самый простой - переключить бит WP. Поэтому этот шаблон кода часто встречается в руткитах. Например, в образце G:

```
01: .text:0001062F   push    eax
02: .text:00010630   mov     eax, cr0
03: .text:00010633   mov     [esp+8+var_4], eax
04: .text:00010637   and     eax, 0FFFEFFFFh
05: .text:0001063C   mov     cr0, eax
06: .text:0001063F   pop     eax
```

Строки 2-3 копируют CR0 в EAX и сохраняют его в локальной переменной. Строки 4-5 отключают бит 16 в EAX и записывают значение обратно в CR0. Бит 16 в CR0 является битом WP.

Существуют как минимум два других решения, которые не изменяют CR0 напрямую. Они используют MDL и знания MMU платформы. Вам потребуется реализовать это в одном из упражнений.

KeServiceDescriptorTable

Как уже говорилось, многие руткиты прибегают к перехвату системных вызовов. Однако, как вы узнали, системные вызовы идентифицируются номером, который используется как индекс в таблице системных вызовов. Кроме того, таблица системных вызовов (`KiServiceTable`) не экспортируется, поэтому простого способа обратиться к ней из драйвера нет. Как авторы руткитов обходят это ограничение?

Ядро экспортирует символ `KeServiceDescriptorTable`, содержащий структуру `KSERVICE_TABLE_DESCRIPTOR` со сведениями о системных вызовах. (Напомним, что на x64 этот символ не экспортируется.) Именно так большинство руткитов обращается к таблице системных вызовов. Следующий шаг - определить расположение целевого системного вызова. Напомним, что системные вызовы идентифицируются номером, а не именем. У авторов руткитов есть несколько способов найти нужный системный вызов. Один способ - жестко задать индекс системного вызова. Другой - дизассемблировать заглушку системного вызова и получить индекс из нее. Оба метода имеют компромисс: их просто реализовать, но они опираются на шаблоны кода или данных, которые могут меняться от пакета обновления к пакету обновления; на некоторых платформах они могут быть надежными, а на других наверняка приведут к нестабильности системы. Несмотря на ненадежность, оба метода часто применяются руткитами в реальных условиях. Например, образец G содержит следующий код:

```
01: .text:000117D4 sub_117D4 proc near
02: .text:000117D4   push    ebp
03: .text:000117D5   mov     ebp, esp
04: .text:000117D7   push    ecx
05: .text:000117D8   mov     ecx, ds:KeServiceDescriptorTable
06: .text:000117DE   mov     ecx, [ecx]
07: .text:000117E0   push    esi
08: .text:000117E1   mov     esi, ds:ZwQuerySystemInformation
09: ...
10: .text:00011808   call    DisableWP
11: .text:0001180D   mov     ecx, ds:KeServiceDescriptorTable
12: .text:00011813   mov     eax, [esi+1]
13: .text:00011816   mov     ecx, [ecx]
14: .text:00011818   mov     dword ptr [ecx+eax*4], offset sub_1123E
15: ...
16: .text:00011836 sub_117D4 endp
```

Строки 5-10 сохраняют адрес `KiServiceTable` в ECX, сохраняют адрес `ZwQuerySystemInformation` в ESI и отключают бит WP. Строка 12 получает второй байт `ZwQuerySystemInformation`; это делается потому, что код предполагает: первая команда функции перемещает номер системного вызова в регистр, а потому 32-разрядное значение после кода операции содержит фактический номер системного вызова (см. следующую врезку). Строки 13-14 перезаписывают этот элемент таблицы служб новой функцией `sub_1123e`. Теперь все вызовы `ZwQuerySystemInformation` будут перенаправляться в `sub_1123e`.

> **Примечание.** Ранее мы упоминали, что строка 12 получает второй байт `ZwQuerySystemInformation`. В 32-разрядной Windows 7 первая команда в
>
> `ZwQuerySystemInformation` имеет вид `b805010000 mov eax` и `105h`. `b8` - код операции MOV, а `05010000` (`0x105`) кодирует непосредственное значение, которым в данном случае является номер системного вызова.

Разделы

Раздел - это объект, используемый для описания памяти, поддерживаемой некоторым видом хранилища. Раздел может поддерживаться обычным файлом или файлом подкачки. Раздел, поддерживаемый файлом, - это раздел, содержимое памяти которого соответствует содержимому файла на диске; изменения раздела записываются непосредственно на диск. Раздел, поддерживаемый файлом подкачки, - это раздел, содержимое которого поддерживается файлом подкачки; изменения такого раздела отбрасываются после его закрытия. Драйвер может создать раздел с помощью API `ZwCreateSection`, а затем отобразить его представление в другой процесс с помощью `ZwMapViewOfSection`. Каждое представление, по сути, является диапазоном виртуальных адресов, через который можно обращаться к памяти, представленной связанным объектом раздела. Следовательно, раздел может иметь несколько представлений.

Пошаговые разборы

Теперь, когда вы хорошо усвоили понятия ядра Windows и драйверов, пришло время применить эти знания при анализе настоящих руткитов. Этот раздел преследует две цели: объяснить ход мыслей при обратном анализе режима ядра и продемонстрировать применение методов разработки драйверов к пониманию руткитов.

Руткиты имеют множество различных форм. Одни перехватывают системные вызовы, другие скрывают файлы путем фильтрации ответов ввода-вывода, третьи перехватывают сетевой обмен, четвертые регистрируют нажатия клавиш и так далее. Однако, подобно всем драйверам, они имеют одну и ту же общую структуру; например, у всех есть функция `DriverEntry` с необязательными обработчиками диспетчеризации IRP, взаимодействующими с ядром через документированные и недокументированные интерфейсы. Обладая этими знаниями, можно расчленить основные компоненты драйвера и систематически их анализировать. Общий процесс анализа выглядит следующим образом:

1. Найдите `DriverEntry` и определите обработчики диспетчеризации IRP, если они имеются.

2. Определите, подключается ли драйвер к другому устройству для фильтрации/перехвата его запросов ввода-вывода. Если да, каково целевое устройство?

3. Если драйвер создает объект устройства, определите имя и размер расширения устройства.

4. Восстановите структуру расширения устройства, наблюдая за использованием ее полей.
5. Если драйвер поддерживает IOCTL, найдите все коды IOCTL и соответствующие им функции. Определите используемый ими метод буферизации.

6. Найдите DPC, рабочие элементы, APC, таймеры, процедуры завершения, функции обратного вызова и системные потоки.

7. Постарайтесь понять, как все части сочетаются друг с другом.

Руткит x86

Пошаговый разбор начинается с образца A.

Его `DriverEntry` начинается по адресу `0x105F0` и заканчивается по адресу `0x106AD`. Сначала она инициализирует структуру `UNICODE_STRING` строками `\Device\fsodhfn2m` и `\DosDevices\fsodhfn2m`. В режиме ядра большинство строк описывается структурой `UNICODE_STRING`:

```
typedef struct _UNICODE_STRING {
  USHORT  Length;
  USHORT  MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

Она инициализируется с помощью API `RtlInitUnicodeString`. Строка «Device» - имя устройства в диспетчере объектов; строка «DosDevices» используется как символическая ссылка на фактическое имя устройства. Диспетчер объектов Windows хранит и организует объекты в структуре, похожей на файловую систему, с корнем в `\`. Существуют четко определенные каталоги, например `\Devices`, `\BaseNamedObjects`, `\KernelObjects` и так далее. `\DosDevices` - псевдоним каталога `\??`; он существует потому, что, когда приложения пользовательского режима задают путь к объекту, к которому хотят обратиться, к нему добавляется префикс `\??\`; каталог `\??` содержит символические ссылки, указывающие на настоящие объекты. Например, когда пользователь хочет обратиться к `"c:\test.txt"` через API `CreateFile`, фактическим путем, отправляемым ядру, является `"\??\c:\test.txt"`; поскольку `"c:"` - символическая ссылка на `\Device\HarddiskVolume2` (в вашей системе она может отличаться), весь путь в конечном итоге разрешается в `\Device\HarddiskVolume2\test.txt`. Символическая ссылка необходима потому, что API пользовательского режима обычно обращаются к устройствам через каталог `\??`; если бы символических ссылок там не было, устройство могло бы оказаться недоступным приложениям пользовательского режима.

После инициализации двух строк функция переходит к созданию фактического объекта устройства. `IoCreateDevice` определяется следующим образом:

```
NTSTATUS
  IoCreateDevice(
    IN PDRIVER_OBJECT  DriverObject,
    IN ULONG  DeviceExtensionSize,
    IN PUNICODE_STRING  DeviceName  OPTIONAL,
    IN DEVICE_TYPE  DeviceType,
    IN ULONG  DeviceCharacteristics,

    IN BOOLEAN  Exclusive,
    OUT PDEVICE_OBJECT  *DeviceObject
    );
```

`DriverObject` - `DRIVER_OBJECT` вызывающей стороны; это объект драйвера, с которым связан новый объект устройства. `DeviceExtensionSize` - количество байтов памяти невыгружаемого пула, которое следует выделить для структуры, специфичной для драйвера. Поскольку это структура, определяемая пользователем, восстановить ее поля очень важно. `DeviceName` - собственное имя устройства. `DeviceType` - один из предопределенных типов `FILE_DEVICE_*`; если устройство не относится ни к одной универсальной категории, вместо этого используется `FILE_DEVICE_UNKNOWN`. `DeviceCharacteristics` обозначает характеристику устройства; чаще всего встречается `FILE_DEVICE_SECURE_OPEN`. `Exclusive` определяет, может ли существовать более одного дескриптора устройства. `DeviceObject` получает фактический объект устройства.

По дизассемблированному коду можно декомпилировать первый базовый блок и условие выхода из него следующим образом:

```
01: UNICODE_STRING devname;
02: UNICODE_STRING symname;
03:
04: NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, \
                         PUNICODE_STRING regpath)
05: {
06:   NTSTATUS status;
07:   PDEVICE_OBJECT devobj;
08:
09:   RtlInitUnicodeString(&devname, L"\\Device\\fsodhfn2m");
10:   RtlInitUnicodeString(&symname, L"\\DosDevices\\fsodhfn2m");
11:   status = IoCreateDevice(
12:                DriverObject,
13:                0,
14:                &devname,
15:                FILE_DEVICE_UNKNOWN,
16:                FILE_DEVICE_SECURE_OPEN,
17:                FALSE,
18:                &devobj);
19:   if (!NT_SUCCESS(status)) {
20:     return status; // loc_106A3
21:   }
22: }
```

`NT_SUCCESS()` - распространенный макрос, проверяющий, что `status` больше или равен 0. После успешного создания объекта выполнение переходит к следующему коду:

```
01: .text:00010643   mov     ecx, [ebp+DriverObject]
02: .text:00010646   mov     dword ptr [ecx+38h], offset sub_10300
03: .text:0001064D   mov     edx, [ebp+DriverObject]
04: .text:00010650   mov     dword ptr [edx+40h], offset sub_10300
05: .text:00010657   mov     eax, [ebp+DriverObject]
06: .text:0001065A   mov     dword ptr [eax+70h], offset sub_10300

07: .text:00010661   mov     ecx, [ebp+DriverObject]
08: .text:00010664   mov     dword ptr [ecx+34h], offset sub_10580
09: .text:0001066B   push    offset SymbolicLinkName  ; SymbolicLinkName
10: .text:00010670   call    ds:IoDeleteSymbolicLink
11: .text:00010676   push    offset DestinationString ; DeviceName
12: .text:0001067B   push    offset SymbolicLinkName  ; SymbolicLinkName
13: .text:00010680   call    ds:IoCreateSymbolicLink
14: .text:00010686   mov     [ebp+var_4], eax
15: .text:00010689   cmp     [ebp+var_4], 0
16: .text:0001068D   jge     short loc_106A1
```

Строки 1-8 присваивают некоторым полям `DRIVER_OBJECT` два указателя на функции. Что находится по смещениям `0x38`, `0x40`, `0x70` и `0x34`?

```
0: kd> dt _DRIVER_OBJECT
nt!_DRIVER_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
...
   +0x034 DriverUnload     : Ptr32     void
   +0x038 MajorFunction    : [28] Ptr32     long
```

По смещению `0x34` находится процедура `DriverUnload`; теперь мы знаем, что драйвер поддерживает динамическую выгрузку, а `sub_10580` является процедурой выгрузки. Смещение `0x38` - начало массива `MajorFunction`; напомним, что это массив обработчиков диспетчеризации IRP. Поскольку существует не более 28 универсальных основных функций IRP, массив `MajorFunction` имеет 28 элементов. Первый индекс равен 0 и соответствует `IRP_MJ_CREATE`; следовательно, мы знаем, что `sub_10300` является обработчиком этого IRP. Смещение `0x40` - третий элемент массива `MajorFunction` (индекс 2); он соответствует `IRP_MJ_CLOSE`, и `sub_10300`

повторно используется как обработчик. Смещение 0x70 - 16-й элемент массива (индекс 0xe), соответствующий IRP_MJ_DEVICE_CONTROL, и его обработчиком также является sub_10300. На этом этапе мы знаем, что sub_10300 является обработчиком IRP чтения, закрытия и управления устройством.

Строки 10-13 удаляют существующую символическую ссылку и создают новую, указывающую на ранее созданный объект устройства.

Теперь можно продолжить декомпиляцию этого блока в DriverEntry следующим образом:

```
01: DriverObject->MajorFunction[IRP_MJ_READ] = sub_10300;
02: DriverObject->MajorFunction[IRP_MJ_CLOSE] = sub_10300;
03: DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = sub_10300;
04: DriverObject->DriverUnload = sub_10580;
05:
06: IoDeleteSymbolicLink(&symname);
07: status = IoCreateSymbolicLink(&symname, &devname);
08: if (!NT_SUCCESS(status)) {
09:   ... // блок .text:0001068F
10:   return status;

11: }
12: return status;
```

Чтобы облегчить дальнейшую работу, можно переименовать sub_10300 в IRP_ReadCloseDeviceIo, а sub_10580 - в DRV_Unload.

Следующий блок по адресу 0x1068F удаляет ранее созданный объект устройства, если создать символическую ссылку не удалось. Обратите внимание, что он получает объект устройства из объекта драйвера, а не использует указатель, переданный IoCreateDevice. Этот блок можно декомпилировать следующим образом:

```
01: IoDeleteDevice(DriverObject->DeviceObject);
```

На этом декомпиляция DriverEntry данного руткита завершена. Подведем итог уже полученным сведениям:

• Драйвер создает объект устройства с именем \Device\fsodhfn2m.

• Он поддерживает динамическую выгрузку, а процедурой выгрузки является sub_10580 (переименованная в DRV_Unload).

• Он поддерживает операции IRP_MJ_READ, IRP_MJ_WRITE и IRP_MJ_DEVICE_CONTROL, а обработчиком является sub_10300 (переименованная в IRP_ReadCloseDeviceIo).

• Он создает символическую ссылку на объект устройства. Если это не удается, драйвер возвращает ошибку.

Следующий шаг - понять, что делает процедура DriverUnload. WDK определяет прототип процедуры выгрузки драйвера следующим образом:

```
VOID
Unload(
    PDRIVER_OBJECT  *DriverObject
    );
```

После небольшой обработки наша процедура выгрузки выглядит так:

```
01: .text:00010580 ; void __stdcall DRV_Unload(PDRIVER_OBJECT drvobj)
02: .text:00010580 DRV_Unload proc near
03: .text:00010580
04: .text:00010580 drvobj= dword ptr  8
05: .text:00010580
06: .text:00010580   push    ebp
07: .text:00010581   mov     ebp, esp
08: .text:00010583   push    offset SymbolicLinkName  ; SymbolicLinkName
09: .text:00010588   call    ds:IoDeleteSymbolicLink
10: .text:0001058E   mov     eax, [ebp+drvobj]
11: .text:00010591   mov     ecx, [eax+DRIVER_OBJECT.DeviceObject]
12: .text:00010594   push    ecx                      ; DeviceObject
13: .text:00010595   call    ds:IoDeleteDevice
14: .text:0001059B   pop     ebp
15: .text:0001059C   retn    4
16: .text:0001059C DRV_Unload endp
```

Предыдущий код можно декомпилировать следующим образом:

```
01: VOID DRV_Unload(PDRIVER_OBJECT drvobj)
02: {
03:   IoDeleteSymbolicLink(&symname);
04:   IoDeleteDevice(drvobj->DeviceObject);
05: }
```

Как уже говорилось, важнейшим ключом к пониманию функциональности драйвера служат его обработчики диспетчеризации IRP. Начиная анализ `_IRP_ReadCloseDeviceIo`, рассмотрим ее начало:

```
01: .text:00010300 ; NTSTATUS __stdcall IRP_ReadCloseDeviceIO(
                       PDEVICE_OBJECT devobj, PIRP Irp)
02: .text:00010300 IRP_ReadCloseDeviceIO proc near
03: .text:00010300 var_14= dword ptr -14h
04: .text:00010300 var_10= dword ptr -10h
05: .text:00010300 var_C= dword ptr -0Ch
06: .text:00010300 var_8= dword ptr -8
07: .text:00010300 var_4= dword ptr -4
08: .text:00010300 devobj= dword ptr  8
09: .text:00010300 Irp= dword ptr  0Ch
10: .text:00010300
11: .text:00010300   push    ebp
12: .text:00010301   mov     ebp, esp
13: .text:00010303   sub     esp, 14h
14: .text:00010306   mov     [ebp+var_4], 0
15: .text:0001030D   mov     eax, [ebp+Irp]
16: .text:00010310   mov     ecx, [ebp+var_4]
17: .text:00010313   mov     [eax+18h], ecx
18: .text:00010316   mov     edx, [ebp+Irp]
19: .text:00010319   mov     dword ptr [edx+1Ch], 0
20: .text:00010320   mov     eax, [ebp+Irp]
21: .text:00010323   mov     ecx, [eax+60h]
22: .text:00010326   mov     [ebp+var_10], ecx
23: .text:00010329   mov     edx, [ebp+var_10]
24: .text:0001032C   movzx   eax, byte ptr [edx]
25: .text:0001032F   cmp     eax, 0Eh
26: .text:00010332   jnz     short loc_1037D
```

Ее прототип нам уже известен, поскольку он одинаков для всех обработчиков IRP. При анализе обработчиков IRP необходимо помнить несколько фактов:

• IRP - динамическая структура, после заголовка которой находится массив `IO_STACK_LOCATION`.

• Большинство параметров IRP находится в `IO_STACK_LOCATION` (включая номер основной/дополнительной функции IRP).

• Драйвер обращается к своей `IO_STACK_LOCATION` с помощью процедуры `IoGetCurrentIrpStacLocation`. Поскольку эта процедура принудительно встраивается,

необходимо распознавать ее по встроенным шаблонам. Получение `IO_STACK_LOCATION` в начале обработчика IRP - распространенный шаблон программирования.

Строки 15-17 читают структуру IRP и записывают 0 в поле по смещению `0x18`. Рассмотрев структуру IRP, можно увидеть следующее:

```
0: kd> dt nt!_IRP
   +0x000 Type             : Int2B
   +0x002 Size             : Uint2B
...
   +0x00c AssociatedIrp    : <unnamed-tag>
...
   +0x018 IoStatus         : _IO_STATUS_BLOCK
      +0x000 Status           : Int4B
      +0x000 Pointer          : Ptr32 Void
      +0x004 Information      : Uint4B
...
   +0x020 RequestorMode    : Char
...
   +0x040 Tail             : <unnamed-tag>
```

Структура `IO_STATUS_BLOCK` хранит сведения о состоянии IRP:

```
typedef struct _IO_STATUS_BLOCK {
  union {
    NTSTATUS  Status;
    PVOID  Pointer;
  };
  ULONG_PTR  Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

Обработчик IRP обычно задает поле `Status`, чтобы указать, был ли IRP выполнен успешно или требует дальнейшей обработки. `Information` хранит специфичные для запроса сведения IRP; драйвер может использовать его для хранения указателя на буфер или задания состояния завершения. `Pointer` зарезервирован.

Следовательно, мы знаем, что строка 17 задает полю `IRP->IoStatus.Status` значение 0, а локальная переменная `var_4` имеет тип `NTSTATUS`. Строки 18-19 обращаются к структуре IRP и записывают 0 по смещению `0x1c`, то есть в поле `Information` структуры `IoStatus`. Это просто присваивание `IRP->IoStatus.Information` значения 0. Строки 20-22 обращаются по смещению `0x60` структуры IRP и сохраняют его адрес в локальной переменной. Структура IRP заполнена объединениями в поле `Tail` (начинающемся со смещения `0x40`), поэтому определить, к какому полю объединения выполняется обращение, может быть непросто. Выведем некоторые из объединений:

```
0: kd> dt nt_IRP Tail.Overlay.
   +0x040 Tail          :
      +0x000 Overlay       :
         +0x000 DeviceQueueEntry : _KDEVICE_QUEUE_ENTRY
         +0x000 DriverContext : [4] Ptr32 Void
```

```
+0x010 Thread        : Ptr32 _ETHREAD
+0x014 AuxiliaryBuffer : Ptr32 Char
+0x018 ListEntry     : _LIST_ENTRY
+0x020 CurrentStackLocation : Ptr32 _IO_STACK_LOCATION
+0x020 PacketType    : Uint4B
+0x024 OriginalFileObject : Ptr32 _FILE_OBJECT
```

Это показывает, что смещение `0x60` может быть либо указателем на `IO_STACK_LOCATION`, либо беззнаковым целым числом, обозначающим тип пакета. На основании контекста кода (обращение происходит в начале обработчика IRP) можно обоснованно предположить, что это поле `CurrentStackLocation`. Кроме того, мы знаем, что встроенная процедура `IoGetCurrentIrpStackLocation` определяется следующим образом:

```
FORCEINLINE
PIO_STACK_LOCATION
IoGetCurrentIrpStackLocation(PIRP Irp)
{
  return Irp->Tail.Overlay.CurrentStackLocation;
}
```

Следовательно, строки 20-22 сохраняют текущую `IO_STACK_LOCATION` в локальной переменной. Локальная переменная `_var_10` имеет тип `PIO_STACK_LOCATION`.

> **Примечание.** Многие из этих функций объявлены как `FORCEINLINE`, поэтому никогда не появляются как цели вызовов, то есть вы никогда не увидите увидите символ `IoGetCurrentIrpStackLocation` в ассемблерном коде. Рекомендуется написать простой драйвер, использующий эти принудительно встраиваемые процедуры, чтобы привыкнуть к шаблону кода.

Строки 23-25 обращаются к первому байту по смещению 0 в `IO_STACK_LOCATION` с помощью команды MOVZX. Это указывает, что поле имеет тип `unsigned char`. Из раздела об IRP мы знаем, что это поле `MajorFunction`. Строка 5 проверяет, равен ли номер `MajorFunction` значению `0xe`, то есть `IRP_MJ_DEVICE_CONTROL`.

Теперь первый блок `IRP_ReadCloseIo` можно декомпилировать следующим образом:

```
NTSTATUS IRP_ReadCloseIo(PDEVICE_OBJECT devobj, PIRP Irp)
{
  NTSTATUS status = STATUS_SUCCESS;
  PIO_STACK_LOCATION isl;
  Irp->IoStatus.Status = status;
  Irp->IoStatus.Information = 0;

  isl = IoGetCurrentIrpStackLocation(Irp);
  if (isl->MajorFunction != IRP_MJ_DEVICE_CONTROL) {
    ... // loc_1037D
  }
  ... // .text:00010334
}
```

Далее проанализируем блок `0x10334`, выполняемый, если основной код равен `IRP_MJ_DEVICE_CONTROL`:

```
01: .text:00010334   mov     ecx, [ebp+var_10]
02: .text:00010337   mov     edx, [ecx+0Ch]
03: .text:0001033A   mov     [ebp+var_C], edx
04: .text:0001033D   mov     eax, [ebp+Irp]
05: .text:00010340   mov     ecx, [eax+0Ch]
06: .text:00010343   mov     [ebp+var_8], ecx
07: .text:00010346   mov     edx, [ebp+Irp]
08: .text:00010349   mov     dword ptr [edx+1Ch], 644h
09: .text:00010350   mov     eax, [ebp+var_C]
10: .text:00010353   mov     [ebp+var_14], eax
11: .text:00010356   cmp     [ebp+var_14], 22C004h
12: .text:0001035D   jz      short loc_10361
```

В предыдущем абзаце мы заключили, что `var_10` имеет тип `PIO_STACK_LOCATION`. Строки 1-2 обращаются по смещению `0xC` структуры `IO_STACK_LOCATION`. Снова напомним, что `IO_STACK_LOCATION` содержит параметры запроса ввода-вывода, которые хранятся в объединениях. Как определить нужное объединение? Мы знаем, что будет использоваться поле `DeviceIoControl`, поскольку обрабатывается запрос `IRP_MJ_DEVICE_CONTROL`. Кроме того, `IoControlField` находится по смещению `0xC` относительно базы `IO_STACK_LOCATION`:

```
1: kd> dt nt!_IO_STACK_LOCATION Parameters.
   +0x004 Parameters  :
      +0x000 Create      : <unnamed-tag>
      +0x000 CreatePipe  : <unnamed-tag>
      +0x000 CreateMailslot : <unnamed-tag>
      +0x000 Read        : <unnamed-tag>
      +0x000 Write       : <unnamed-tag>
      +0x000 QueryDirectory : <unnamed-tag>
...
      +0x000 DeviceIoControl : <unnamed-tag>
...
1: kd> dt nt!_IO_STACK_LOCATION Parameters.DeviceIoControl.
   +0x004 Parameters                  :
      +0x000 DeviceIoControl             :
         +0x000 OutputBufferLength          : Uint4B
         +0x004 InputBufferLength           : Uint4B
         +0x008 IoControlCode               : Uint4B
         +0x00c Type3InputBuffer            : Ptr32 Void
```

Следовательно, строки 1-3 получают поле `IoControlCode` и сохраняют его в `var_C`, которая, как теперь известно, имеет тип `ULONG`.

Строки 4-6 обращаются по смещению `0xC` в IRP и сохраняют указатель в локальной переменной `var_8`. Из предыдущего раздела мы знаем, что по смещению `0xC` находится объединение `AssociatedIrp`:

```
1: kd> dt nt!_IRP AssociatedIrp.
   +0x00c AssociatedIrp  :

      +0x000 MasterIrp      : Ptr32 _IRP
      +0x000 IrpCount       : Int4B
      +0x000 SystemBuffer   : Ptr32 Void
```

Какое из трех полей следует использовать? По имеющимся сведениям определить это невозможно. Контекст, необходимый для выбора правильного поля, содержится в строках 9-12, которые получают сохраненный код IOCTL (`var_C`) и сравнивают его с `0x22c004`. Мы знаем, что код IOCTL кодирует тип устройства, код функции, доступ и метод буферизации. Поэтому после декодирования `0x22c004` становятся известны следующие сведения:

• Тип устройства - `FILE_DEVICE_UNKNOWN` (`0x22`).

• Код IOCTL - `0x1`.

• Доступ - (`FILE_READ_DATA | FILE_WRITE_DATA`).

• Метод буферизации - `METHOD_BUFFERED`.

Напомним, что мы находимся в обработчике IOCTL и при определении кода IOCTL драйверы должны указывать метод буферизации. Для буферизованного ввода-вывода поле `SystemBuffer` указывает на буфер невыгружаемого пула, хранящий пользовательские входные данные. Теперь можно утверждать, что строки 4-6 обращаются к полю `SystemBuffer`.

Строки 7-8 записывают `0x644` по смещению `0x1c` внутри IRP, то есть в поле `IRP->IoStatus.Information`. Неясно, почему автор выбрал это значение.

Учитывая эти сведения, мы знаем, что код управления должен был быть построен следующим образом:

```
#define IOCTL_1 CTL_CODE(FILE_DEVICE_UNKNOWN, 1, METHOD_BUFFERED, \
                                FILE_READ_DATA | FILE_WRITE_DATA)
```

Поскольку мы еще не полностью проанализировали и не поняли операцию IOCTL, ей дано универсальное имя `IOCTL_1`. Теперь этот блок можно декомпилировать следующим образом:

```
PVOID userinput = Irp->AssociatedIrp.SystemBuffer;
Irp->IoStatus.Information = (ULONG_PTR) 0x644;
if (isl->Parameters.DeviceIoControl.IoControlCode == IOCTL_1)
{
  ... // loc_10361
}
... // 0001035F
```

Чтобы понять, что делает IOCTL, нужно проанализировать `loc_10361` и функцию `sub_103B0`. Однако прежде закончим с соседними блоками (они проще):

```
; помните, var_4 - локальная переменная status (тип NTSTATUS)
01: .text:0001035F   jmp     short loc_1036C
02: .text:00010361 loc_10361:
03: .text:00010361   mov     ecx, [ebp+var_8]
04: .text:00010364   push    ecx

05: .text:00010365   call    IOCTL_1_handler
06: .text:0001036A   jmp     short loc_1037D
07: .text:0001036C loc_1036C:
08: .text:0001036C   mov     [ebp+var_4], 0C0000010h
09: .text:00010373   mov     edx, [ebp+Irp]
10: .text:00010376   mov     dword ptr [edx+1Ch], 0
11: .text:0001037D loc_1037D:
12: .text:0001037D   cmp     [ebp+var_4], 103h
13: .text:00010384   jz      short loc_1039A
14: .text:00010386   xor     dl, dl                  ; PriorityBoost
15: .text:00010388   mov     ecx, [ebp+Irp]          ; Irp
16: .text:0001038B   call    ds:IofCompleteRequest
17: .text:00010391   mov     eax, [ebp+Irp]
18: .text:00010394   mov     ecx, [ebp+var_4]
19: .text:00010397   mov     [eax+18h], ecx
20: .text:0001039A loc_1039A:
21: .text:0001039A   mov     eax, [ebp+var_4]
22: .text:0001039D   mov     esp, ebp
23: .text:0001039F   pop     ebp
24: .text:000103A0   retn    8
25: .text:000103A0 IRP_ReadCloseDeviceIO endp
```

В `0x1035F` выполнение попадает, если код IOCTL не совпадает. Там сразу происходит переход к строке 7, которая присваивает локальной переменной состояния значение `0xC0000010`, то есть `STATUS_INVALID_OPERATION`, а `Irp->IoStatus.Information` - значение 0. Затем в строке 11 проверяется, равно ли локальное состояние `0x103` (`STATUS_PENDING`); в действительности этот блок избыточен, поскольку переменная состояния в этой функции может иметь только два значения (`STATUS_SUCCESS` или `STATUS_INVALID_OPERATION`). Когда IRP помечен `STATUS_PENDING`, это означает, что операция не завершена и ожидает завершения другим драйвером. Такое часто встречается в драйверах, поэтому полезно запомнить магическую константу `0x103`. Если состояние равно `STATUS_PENDING`, обработчик немедленно возвращает его (строки 13 и 20). В противном случае он вызывает `IoCompleteRequest`, чтобы пометить IRP завершенным, сохраняет состояние в `IRP->IoStatus.Status` (строка 19) и возвращает его. В действительности это ошибка, поскольку драйвер должен задавать поле `IoStatusBlock` до завершения запроса; после завершения IRP к нему больше нельзя обращаться. Эти блоки можно декомпилировать следующим образом:

```
status = STATUS_INVALID_OPERATION;
Irp->IoStatus.Information = 0;
if (status == STATUS_PENDING) {
  return status;
}
IoCompleteRequest(Irp, IO_NO_INCREMENT);
Irp->IoStatus.Status = status;
return status;
```

Возвращаясь к процедуре `IOCTL_1_handler`, обратите внимание, что она вызывает только две другие функции: `sub_10460` и `sub_10550`. `sub_10550` - небольшая листовая процедура, поэтому сначала проанализируем ее:

```
01: .text:00010550 ; void __stdcall sub_10550(PMDL Mdl, PVOID BaseAddress)
02: .text:00010550 sub_10550 proc near
03: .text:00010550   push    ebp
04: .text:00010551   mov     ebp, esp
05: .text:00010553   mov     eax, [ebp+Mdl]
06: .text:00010556   push    eax                     ; MemoryDescriptorList
07: .text:00010557   mov     ecx, [ebp+BaseAddress]
08: .text:0001055A   push    ecx                     ; BaseAddress
09: .text:0001055B   call    ds:MmUnmapLockedPages
10: .text:00010561   mov     edx, [ebp+Mdl]
11: .text:00010564   push    edx                     ; MemoryDescriptorList
12: .text:00010565   call    ds:MmUnlockPages
13: .text:0001056B   mov     eax, [ebp+Mdl]
14: .text:0001056E   push    eax                     ; Mdl
15: .text:0001056F   call    ds:IoFreeMdl
16: .text:00010575   pop     ebp
17: .text:00010576   retn    8
18: .text:00010576 sub_10550 endp
```

Эта функция отменяет отображение MDL, снимает его закрепление и освобождает его. Неясно, что описывают MDL, поскольку другие процедуры еще не проанализированы. Функцию можно декомпилировать следующим образом:

```
void UnmapMdl(PMDL mdl, PVOID baseaddr)
{
  MmUnmapLockedPages(baseaddr, mdl);
  MmUnlockPages(mdl);
  IoFreeMdl(mdl);
}
```

sub_10460 - еще одна листовая процедура, работающая с MDL; ее основная функция - создать, закрепить и отобразить MDL для заданного буфера и длины. Ее прототип выглядит следующим образом:

```
PVOID MapMdl(PMDL *mdl, PVOID VirtualAddress, ULONG Length);
```

По умолчанию дизассемблер не смог вывести тип первого параметра. Мы можем определить, что это PMDL *, по команде по адресу 0x1049D. Ассемблерный листинг приведен ниже без построчных комментариев, поскольку он очень прост:

```
01: .text:00010460 ; PVOID __stdcall MapMdl(PMDL *mdl,
                         PVOID VirtualAddress, ULONG Length)
02: .text:00010460 MapMdl proc near
03: .text:00010460   push    ebp
04: .text:00010461   mov     ebp, esp
05: .text:00010463   push    0FFFFFFFFh
06: .text:00010465   push    offset unk_10748
07: .text:0001046A   push    offset _except_handler3

08: .text:0001046F   mov     eax, large fs:0
09: .text:00010475   push    eax
10: .text:00010476   mov     large fs:0, esp
11: .text:0001047D   add     esp, 0FFFFFFF0h
12: .text:00010480   push    ebx
13: .text:00010481   push    esi
14: .text:00010482   push    edi
15: .text:00010483   mov     [ebp+var_18], esp
16: .text:00010486   push    0                   ; Irp
17: .text:00010488   push    0                   ; ChargeQuota
18: .text:0001048A   push    0                   ; SecondaryBuffer
19: .text:0001048C   mov     eax, [ebp+Length]
20: .text:0001048F   push    eax                 ; Length
21: .text:00010490   mov     ecx, [ebp+VirtualAddress]
22: .text:00010493   push    ecx                 ; VirtualAddress
23: .text:00010494   call    ds:IoAllocateMdl
24: .text:0001049A   mov     edx, [ebp+mdl]
25: .text:0001049D   mov     [edx], eax
26: .text:0001049F   mov     eax, [ebp+mdl]
27: .text:000104A2   cmp     dword ptr [eax], 0
28: .text:000104A5   jnz     short loc_104AE
29: .text:000104A7   xor     eax, eax
30: .text:000104A9   jmp     loc_10534
31: .text:000104AE loc_104AE:
32: .text:000104AE   mov     [ebp+var_4], 0
33: .text:000104B5   push    1                   ; Operation
34: .text:000104B7   push    0                   ; AccessMode
35: .text:000104B9   mov     ecx, [ebp+mdl]
36: .text:000104BC   mov     edx, [ecx]
37: .text:000104BE   push    edx                 ; MemoryDescriptorList
38: .text:000104BF   call    ds:MmProbeAndLockPages
39: .text:000104C5   mov     [ebp+var_4], 0FFFFFFFFh
40: .text:000104CC   jmp     short loc_104F6
41: .text:000104CE loc_104CE:
42: .text:000104CE   mov     eax, 1
43: .text:000104D3   retn
44: .text:000104D4 loc_104D4:
45: .text:000104D4   mov     esp, [ebp+var_18]
46: .text:000104D7   mov     eax, [ebp+mdl]
47: .text:000104DA   mov     ecx, [eax]
48: .text:000104DC   push    ecx                 ; Mdl
49: .text:000104DD   call    ds:IoFreeMdl
50: .text:000104E3   mov     [ebp+var_20], 0
51: .text:000104EA   mov     [ebp+var_4], 0FFFFFFFFh
52: .text:000104F1   mov     eax, [ebp+var_20]
53: .text:000104F4   jmp     short loc_10534
54: .text:000104F6 loc_104F6:
55: .text:000104F6   push    10h                 ; Priority
56: .text:000104F8   push    0                   ; BugCheckOnFailure
57: .text:000104FA   push    0                   ; BaseAddress
```

```
58: .text:000104FC   push    0                   ; CacheType
59: .text:000104FE   push    0                   ; AccessMode
60: .text:00010500   mov     edx, [ebp+mdl]
61: .text:00010503   mov     eax, [edx]
62: .text:00010505   push    eax                 ; MemoryDescriptorList
63: .text:00010506   call    ds:MmMapLockedPagesSpecifyCache
64: .text:0001050C   mov     [ebp+var_1C], eax
65: .text:0001050F   cmp     [ebp+var_1C], 0
66: .text:00010513   jnz     short loc_10531
67: .text:00010515   mov     ecx, [ebp+mdl]
68: .text:00010518   mov     edx, [ecx]
69: .text:0001051A   push    edx                 ; MemoryDescriptorList
70: .text:0001051B   call    ds:MmUnlockPages
71: .text:00010521   mov     eax, [ebp+mdl]
72: .text:00010524   mov     ecx, [eax]
73: .text:00010526   push    ecx                 ; Mdl
74: .text:00010527   call    ds:IoFreeMdl
75: .text:0001052D   xor     eax, eax
76: .text:0001052F   jmp     short loc_10534
77: .text:00010531 loc_10531:
78: .text:00010531   mov     eax, [ebp+var_1C]
79: .text:00010534 loc_10534:
80: .text:00010534   mov     ecx, [ebp+var_10]
81: .text:00010537   mov     large fs:0, ecx
82: .text:0001053E   pop     edi
83: .text:0001053F   pop     esi
84: .text:00010540   pop     ebx
85: .text:00010541   mov     esp, ebp
86: .text:00010543   pop     ebp
87: .text:00010544   retn    0Ch
88: .text:00010544 MapMdl endp
```

Хотя эта функция кажется длинной и сложной, понять ее нетрудно, если посмотреть, как API используются вместе. `IoAllocateMdl`, `MmProbeAndLockPages` и `MmMapLockedPagesSpecifyCache` - процедуры для создания, закрепления и отображения MDL; `MmProbeAndLockPages` должна выполняться внутри блока `try/except`, поэтому в начале генерируется дополнительный код, настраивающий обработчик исключений (то есть строки с `fs:0`). По существу, эта процедура отображает буфер в пространство ядра с правом записи и возвращает адрес нового отображения этого буфера. В общих чертах всю процедуру можно декомпилировать следующим образом:

```
PVOID MapMdl(PMDL *mdl, PVOID VirtualAddress, ULONG Length)
{
  PVOID addr; // виртуальный адрес отображенной MDL

  *mdl = IoAllocateMdl(VirtualAddress, Length, FALSE, FALSE, NULL);
  if (*mdl == NULL) return NULL;
  __try {
    MmProbeAndLockPages(*mdl, KernelMode, IoWriteAccess);
    addr = MmMapLockedPagesSpecifyCache(

      *mdl,
      KernelMode,
      MmNonCached,
      NULL,
      FALSE,
      NormalPagePriority);
    if (addr == NULL) {
      MmUnlockPages(*mdl);
      IoFreeMdl(*mdl);
    }
  } __except (EXCEPTION_EXECUTE_HANDLER) {
    IoFreeMdl(*mdl);
  }
  return addr;
}
```

Разобравшись с этими двумя процедурами, теперь можно перейти к обработчику. Обратите внимание, что он принимает один параметр - `Irp->AssociatedIrp.SystemBuffer`. Напомним: после завершения IRP содержимое этого буфера может быть скопировано обратно в режим пользователя:

```
01: .text:000103B0 ; void __stdcall IOCTL_1_handler(PVOID buffer)
02: .text:000103B0 IOCTL_1_handler proc near
03: .text:000103B0   push    ebp
04: .text:000103B1   mov     ebp, esp
05: .text:000103B3   sub     esp, 10h
06: .text:000103B6   push    esi
07: .text:000103B7   call    ds:KeRaiseIrqlToDpcLevel
08: .text:000103BD   mov     [ebp+NewIrql], al
09: .text:000103C0   mov     eax, ds:KeServiceDescriptorTable
10: .text:000103C5   mov     ecx, [eax+8]
11: .text:000103C8   shl     ecx, 2
12: .text:000103CB   push    ecx                 ; Length
13: .text:000103CC   mov     edx, ds:KeServiceDescriptorTable
14: .text:000103D2   mov     eax, [edx]
15: .text:000103D4   push    eax                 ; VirtualAddress
16: .text:000103D5   lea     ecx, [ebp+Mdl]
17: .text:000103D8   push    ecx                 ; mdl
18: .text:000103D9   call    MapMdl
19: .text:000103DE   mov     [ebp+BaseAddress], eax
20: .text:000103E1   cmp     [ebp+BaseAddress], 0
21: .text:000103E5   jz      short loc_10449
22: .text:000103E7   mov     [ebp+var_8], 0
23: .text:000103EE   jmp     short loc_103F9
24: .text:000103F0 loc_103F0:
25: .text:000103F0   mov     edx, [ebp+var_8]
26: .text:000103F3   add     edx, 1
27: .text:000103F6   mov     [ebp+var_8], edx
28: .text:000103F9 loc_103F9:
29: .text:000103F9   mov     eax, [ebp+buffer]

30: .text:000103FC   mov     ecx, [ebp+var_8]
31: .text:000103FF   cmp     ecx, [eax]
32: .text:00010401   jnb     short loc_1043C
33: .text:00010403   mov     edx, [ebp+var_8]
34: .text:00010406   mov     eax, [ebp+buffer]
35: .text:00010409   cmp     dword ptr [eax+edx*4+4], 0
36: .text:0001040E   jz      short loc_1043A
37: .text:00010410   mov     ecx, [ebp+var_8]
38: .text:00010413   mov     edx, [ebp+BaseAddress]
39: .text:00010416   mov     eax, [ebp+var_8]
40: .text:00010419   mov     esi, [ebp+buffer]
41: .text:0001041C   mov     ecx, [edx+ecx*4]
42: .text:0001041F   cmp     ecx, [esi+eax*4+4]
43: .text:00010423   jz      short loc_1043A
44: .text:00010425   mov     edx, [ebp+var_8]
45: .text:00010428   mov     eax, [ebp+buffer]
46: .text:0001042B   mov     ecx, [eax+edx*4+4]
47: .text:0001042F   mov     edx, [ebp+var_8]
48: .text:00010432   mov     eax, [ebp+BaseAddress]
49: .text:00010435   lea     edx, [eax+edx*4]
50: .text:00010438   xchg    ecx, [edx]
51: .text:0001043A loc_1043A:
52: .text:0001043A   jmp     short loc_103F0
53: .text:0001043C loc_1043C:
54: .text:0001043C   mov     eax, [ebp+BaseAddress]
55: .text:0001043F   push    eax                 ; BaseAddress
56: .text:00010440   mov     ecx, [ebp+Mdl]
57: .text:00010443   push    ecx                 ; Mdl
58: .text:00010444   call    UnmapMdl
59: .text:00010449 loc_10449:
60: .text:00010449   mov     cl, [ebp+NewIrql]   ; NewIrql
```

```
61: .text:0001044C   call    ds:KfLowerIrql
62: .text:00010452   pop     esi
63: .text:00010453   mov     esp, ebp
64: .text:00010455   pop     ebp
65: .text:00010456   retn    4
66: .text:00010456 IOCTL_1_handler endp
```

Сначала эта функция повышает IRQL до `DISPATCH_LEVEL` (строка 7), что фактически приостанавливает диспетчер потоков на текущем процессоре. Что бы ни делала эта функция, она не может ожидать или вызвать страничное исключение; иначе произойдет аварийная остановка машины. Того же эффекта можно добиться с помощью `KeRaiseIrql`. Строка 8 сохраняет прежний IRQL, чтобы позднее его можно было восстановить (см. строку 61). Строки 9-11 извлекают недокументированное поле записи `KeServiceDescriptorTable` и умножают его на 4. Строки 12-18 передают `KiServiceTable`, длину (в четыре раза превышающую размер таблицы системных вызовов) и указатель MDL функции `MapMdl`. Поскольку мы уже проанализировали `MapMdl`, нам известно, что она просто отображает буфер от `KiServiceTable` до `KiServiceTable+(NumberOfSyscalls*4)`. В строке 12 сохраняется виртуальный адрес вновь отображенного буфера. Строки 20-22 проверяют состояние отображения; если оно оказалось неуспешным, IRQL понижается и код возвращает управление (строки 60-65); в противном случае начинается цикл, счетчик которого определяется пользовательским вводом (строки 29-31). Тело цикла занимает строки 33-50, и его можно представить следующим образом:

```
DWORD *userbuffer = Irp->AssociatedIrp.SystemBuffer;
DWORD *mappedKiServiceTable = MapMdl(mdl, KiServiceTable, nsyscalls*4);
for (i=0; i < userbuffer[0] ; i++)
{
  if ( userbuffer[i+1] != 0) {
    if ( userbuffer[i+1] != mappedKiServiceTable[i]) {
      swap(mappedKiServiceTable[i], userbuffer[i+1]);
    }
  }
}
...
UnmapMdl(mdl);
KeLowerIrql(oldirql);
```

После многих страниц объяснений и декомпиляции всего драйвера теперь можно понять назначение образца. По какой-то причине разработчик этого драйвера хотел при помощи IOCTL перезаписать таблицу собственных системных вызовов NT произвольными адресами. Буфер режима пользователя представляет собой структуру следующего формата:

```
[количество системных вызовов]
[адрес замены системного вызова 1]
[адрес замены системного вызова 2]
...
[адрес замены системного вызова n]
```

Хотя разработчик, возможно, и достиг своей цели, в драйвере есть несколько критических проблем, способных привести к нестабильности системы и уязвимостям безопасности. Некоторые из них упоминались в ходе пошагового разбора, но вы должны суметь обнаружить и многие другие. Вот несколько вопросов, с которых можно начать поиски:

• Будет ли этот драйвер работать в многоядерной системе? Объясните свои рассуждения.

• Почему автор считает, что IRQL необходимо повысить до `DISPATCH_LEVEL`? Действительно ли это необходимо?

• Как обычный пользователь может применить этот драйвер для выполнения произвольного кода в контексте кольца 0?

• Предположим, автор хотел заменить некоторые системные вызовы собственной реализацией в пользовательском пространстве. Какие проблемы при этом могут возникнуть?

Этот драйвер очень мал и прост, однако в нем присутствует большинство важных конструкций, обычно встречающихся в программных драйверах: диспетчерские процедуры, управление вводом-выводом устройства из режима пользователя, методы буферизации, символические ссылки, повышение и понижение уровней IRQL, управление MDL, структуры `IO_STACK_LOCATION` и так далее. Те же приемы анализа, которые были показаны здесь, можно применять к другим драйверам. Только не подражайте его приемам разработки в реальной жизни.

Руткит x64

В этом разделе анализируется образец B - драйвер x64. Поскольку он довольно велик и сложен, мы сосредоточимся только на областях, связанных с обратными вызовами. Мы не станем приводить каждую строку этой функции, поэтому вам придется следить за ней в дизассемблере.

Обратите внимание, что этот драйвер задает уведомления о создании процессов и загрузке образов с помощью документированных API. `0x4045F8` - начало процедуры обратного вызова при создании процесса. Сначала она обнуляет структуру `LARGE_INTEGER`. Структура `LARGE_INTEGER` обычно применяется для представления размера файла или времени (обратите внимание: позднее, по адресу `0x4046FF`, она используется как аргумент `KeDelayExecutionThread`). Затем функция получает идентификатор текущего процесса с помощью `PsGetCurrentProcessId`. Получает ли она идентификатор только что созданного процесса? Не обязательно. Прототип обратного вызова при создании процесса выглядит следующим образом:

```
VOID
(*PCREATE_PROCESS_NOTIFY_ROUTINE) (
    IN HANDLE  ParentId,
    IN HANDLE  ProcessId, // processId созданного/завершенного процесса
    IN BOOLEAN  Create // TRUE=создание FALSE=завершение
    );
```

Параметр `Creation` сохраняется и проверяется по адресам `0x404604` и `0x404631` соответственно; если он равен `TRUE`, обратный вызов просто возвращает управление. Следовательно, нам известно, что этот обратный вызов отслеживает только завершение процессов. При завершении процесса обратный вызов выполняется в контексте умирающего процесса. Получив идентификатор завершающегося процесса (который вообще не используется), он извлекает объект `EPROCESS` текущего процесса через `IoGetCurrentProcess` (`0x40461C` и `0x404622`).

Непонятно, почему IoGetCurrentProcess вызывается дважды (возможно, это опечатка в исходном коде). Затем процедура извлекает и сохраняет строку с именем файла образа процесса при помощи PsGetProcessImageFileName (0x404633). Хотя эта процедура не документирована, она проста, экспортируется и часто используется ядром. После этого код пытается получить блокировку ресурса, ранее инициализированную в DriverEntry (0x4025EB); перед получением блокировки ресурса он входит в критическую область, поскольку KeAcquireResourceExclusiveLite требует отключенных обычных APC ядра (именно это делает KeEnterCriticalRegion). Далее он получает указатель на связный список и сравнивает имя образа завершающегося процесса с каждой записью списка (смещение 0x20). Мы знаем, что это связный список, поскольку цикл перебирает указатели (0x404679) и завершается, когда два указателя совпадают (0x40465F). Если совпадения нет, код освобождает блокировку ресурса и приостанавливает текущий поток (0x4046FF) на одну секунду относительно текущего времени. Если имя файла завершающегося процесса совпадает с одним из имен в списке, код отменяет отображение MDL из записи списка, снимает закрепление ее страниц и освобождает ее (смещение 0x1070). Если буфер по смещению 0x10b0 в записи списка равен NULL, он освобождается; в противном случае запись удаляется из списка макросом RemoveEntryList:

```
01: .text:00000000004046CA loc_4046CA:
02: .text:00000000004046CA   mov     rax, [rbx+8]
03: .text:00000000004046CE   mov     r8, [rbx]
04: .text:00000000004046D1   mov     edx, edi      ; Tag
05: .text:00000000004046D3   mov     [rax], r8
06: .text:00000000004046D6   mov     rcx, rbx      ; P
07: .text:00000000004046D9   mov     [r8+8], rax
08: .text:00000000004046DD   call    cs:ExFreePoolWithTag
```

И здесь операцию со списком можно распознать по характерной схеме манипуляций с Flink (смещение 0x0) и Blink (смещение 0x8). Фактически теперь мы можем утверждать, что qword_40A590 имеет тип LIST_ENTRY.

Хотя этот обратный вызов - лишь одна часть головоломки, предыдущие факты можно применить, чтобы косвенно понять другие компоненты руткита. Например, можно определить, что руткит либо отображает, либо внедряет код в процессы и отслеживает их в большом связном списке (используя имя процесса как ключ). Когда процесс завершается, эти MDL необходимо отменить, поскольку система аварийно остановится, если у мертвого процесса останутся закрепленные страницы. Первоначальные отображения MDL, скорее всего, создавались процедурой обратного вызова при загрузке образа (0x406494).

Еще одна интересная процедура в этом файле находится по адресу 0x4038F0. Мы проведем ее построчный анализ, поскольку она использует конструкции, которые вам часто будут встречаться в других драйверах. Кроме того, она преподает несколько ценных уроков анализа оптимизированного кода x64:

```
01: ; NTSTATUS __cdecl sub_4038F0(PFILE_OBJECT FileObject, \
                                  HANDLE Handle, BOOLEAN flag)
02: sub_4038F0 proc near
03:   push    rbx
04:   push    rbp
05:   push    rsi
06:   push    rdi
07:   push    r12
08:   sub     rsp, 60h
09:   mov     bpl, r8b
10:   mov     r12, rdx
11:   mov     rdi, rcx
12:   call    cs:IoGetRelatedDeviceObject
13:   mov     [rsp+88h+arg_18], 1
14:   xor     edx, edx       ; ChargeQuota
15:   mov     cl, [rax+4Ch]  ; StackSize
16:   mov     rsi, rax
17:   call    cs:IoAllocateIrp
```

```
18:   test    rax, rax
19:   mov     rbx, rax
20:   jnz     short loc_403932
21:   mov     eax, 0C0000017h
22:   jmp     loc_403A0C
23: loc_403932:
24:   lea     rax, [rsp+88h+arg_18]
25:   xor     r8d, r8d       ; State

26:   lea     rcx, [rsp+88h+Event] ; Event
27:   mov     [rbx+18h], rax ; IRP.AssociatedIrp.SystemBuffer
28:   lea     rax, [rsp+88h+Event]
29:   lea     edx, [r8+1]    ; Type
30:   mov     [rbx+50h], rax ; IRP.UserEvent
31:   lea     rax, [rsp+88h+var_58]
32:   mov     [rbx+48h], rax ; IRP.UserIosb
33:   mov     rax, gs:+188h  ; KPCR.Prcb.CurrentThread
34:   mov     [rbx+0C0h], rdi ; IRP.Tail.Overlay.OriginalFileObject
35:   mov     [rbx+98h], rax ; IRP.Tail.Overlay.Thread
36:   mov     byte ptr [rbx+40h], 0 ; IRP.RequestorMode
37:   call    cs:KeInitializeEvent
38:   test    bpl, bpl
39:   mov     rcx, [rbx+0B8h]
40:   mov     byte ptr [rcx-48h], 6 ; IRP_MJ_SET_INFORMATION
41:   mov     [rcx-20h], rsi ; IO_STACK_LOCATION.DeviceObject
42:   mov     [rcx-18h], rdi ; IO_STACK_LOCATION.FileObject
43:   jz      short loc_4039A6
44:   mov     rax, [rdi+28h] ; FILE_OBJECT.SectionObjectPointer
45:   test    rax, rax
46:   jz      short loc_4039A6
47:   mov     [rax+10h], 0 ; SECTION_OBJECT_POINTERS.ImageSectionObject
48: loc_4039A6:
49:   mov     [rcx-28h], r12
                      ; IO_STACK_LOCATION.Parameters.SetFile.DeleteHandle
50:   mov     [rcx-30h], rdi
                      ; IO_STACK_LOCATION.Parameters.SetFile.FileObject
51:   mov     dword ptr [rcx-38h], 0Dh ; FileDispositionInformation
                      ; IO_STACK_LOCATION.Parameters.SetFile.FileInformationClass
52:   mov     dword ptr [rcx-40h], 1
                      ; IO_STACK_LOCATION.Parameters.SetFile.Length
53:   mov     rax, [rbx+0B8h] ; CurrentIrpStackLocation
54:   lea     rcx, sub_4038B4 ; completionroutine
55:   mov     [rax-10h], rcx  ; IO_STACK_LOCATION.CompletionRoutine
56:   mov     rcx, rsi       ; DeviceObject
57:   mov     rdx, rbx       ; Irp
58:   mov     qword ptr [rax-8], 0
59:   mov     byte ptr [rax-45h], 0E0h ; flag
60:   call    cs:IofCallDriver
61:   cmp     eax, 103h ; STATUS_PENDING
62:   jnz     short loc_403A09
63:   lea     rcx, [rsp+88h+Event] ; Object
64:   mov     r9b, 1         ; Alertable
65:   xor     r8d, r8d       ; WaitMode
66:   xor     edx, edx       ; WaitReason
67:   mov     [rsp+88h+var_68], 0
68:   call    cs:KeWaitForSingleObject
69: loc_403A09:
70:   mov     eax, [rbx+30h] ; IRP.IoStatus.Status
71: loc_403A0C:
72:   add     rsp, 60h
73:   pop     r12
74:   pop     rdi

75:   pop     rsi
76:   pop     rbp
77:   pop     rbx
78:   retn
79: sub_4038F0 endp
```

Сначала восстановим прототип функции, заметив, что вызывающий ее код использует три регистра: RCX, RDX, R8 (см. адреса с 0x404AC8 по 0x404ADB). Хотя дизассемблер помечает

соглашение о вызовах функции как CDECL, на самом деле это неверно. Напомним, что Windows на платформе x64 использует только одно соглашение о вызовах: первые четыре аргумента передаются через регистры (RCX, RDX, R8 и R9), а остальные помещаются в стек. В строке 12 вызывается IoGetRelatedDeviceObject с параметром FileObject; этот API возвращает объект устройства, связанный с объектом файла. Связанный объект устройства сохраняется в RSI. В строках 14-17 с нуля выделяется IRP при помощи IoAllocateIrp; поле StackSize объекта устройства используется как размер IO_STACK_LOCATION нового IRP. Если выделить IRP по какой-либо причине не удается, процедура возвращает STATUS_NO_MEMORY (строки 20-22). В противном случае новый IRP сохраняется в RBX (строка 19), и выполнение продолжается со строки 24. Строки 24-37 инициализируют основные поля IRP и вызывают KeInitializeEvent. Строка 33 может показаться странной из-за операнда GS:188h. Напомним, что в Windows x64 ядро хранит в GS указатель на PCR, в котором находится PRCB, хранящий сведения о планировании. Фактически эта процедура представляет собой встроенную форму KeGetCurrentThread. Строка 39 обращается к полю по смещению 0xb8 в структуре IRP. Что это за поле?

```
0: kd> dt nt!_IRP Tail.Overlay.
   +0x078 Tail          :
      +0x000 Overlay       :
         +0x000 DeviceQueueEntry : _KDEVICE_QUEUE_ENTRY
         +0x000 DriverContext : [4] Ptr64 Void
         +0x020 Thread        : Ptr64 _ETHREAD
         +0x028 AuxiliaryBuffer : Ptr64 Char
         +0x030 ListEntry     : _LIST_ENTRY
         +0x040 CurrentStackLocation : Ptr64 _IO_STACK_LOCATION
         +0x040 PacketType    : Uint4B
         +0x048 OriginalFileObject : Ptr64 _FILE_OBJECT
```

Код обращается к указателю CurrentStackLocation в объединении Overlay. Звучит знакомо? Строка 39 в действительности является всего лишь IoGetCurrentIrpStackLocation. Строки 40-42 задают несколько полей, используя отрицательные смещения от текущей позиции стека. Напомним, что динамическая часть IRP - это массив структур IO_STACK_LOCATION, а "следующая" позиция стека на самом деле является элементом над текущим. Еще раз взгляните на эту структуру и ее размер:

```
0: kd> sizeof(_IO_STACK_LOCATION)
unsigned int64 0x48
0: kd> dt _IO_STACK_LOCATION
nt!_IO_STACK_LOCATION

   +0x000 MajorFunction    : UChar
   +0x001 MinorFunction    : UChar
   +0x002 Flags            : UChar
   +0x003 Control          : UChar
   +0x008 Parameters       : <unnamed-tag>
   +0x028 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x030 FileObject       : Ptr64 _FILE_OBJECT
   +0x038 CompletionRoutine : Ptr64     long
   +0x040 Context          : Ptr64 Void
```

Размер IRP в Windows x64 равен 0x48. Следовательно, строка 40 должна обращаться к "следующей" структуре IO_STACK_LOCATION, поскольку из текущей позиции вычитается 0x48 байт; в ней полю MajorFunction присваивается значение 0x6 (IRP_MJ_SET_INFORMATION). Это говорит о том, что параметры этого запроса будут описываться членом объединения SetFile. Строка 41 обращается к "следующему" IRP с отрицательными смещениями 0x20 и 0x18, которые соответствуют полям DeviceObject и FileObject. Здесь произошло следующее: разработчик использовал IoGetNextIrpStackLocation, а затем заполнил поле, и агрессивно оптимизирующий компилятор Microsoft x64 преобразовал код именно таким образом. Оптимизатор решил, что, поскольку выполняется работа с массивом структур, непосредственное обращение к предыдущему элементу по отрицательным смещениям дешевле (с точки зрения занимаемого места); альтернативой было бы вычислить новый базовый указатель для предыдущего элемента и

обращаться к его полям по положительным смещениям. Такая оптимизация будет довольно часто встречаться вам в двоичных файлах x64.

Строка 43 проверяет флаг, чтобы определить, следует ли выполнять дополнительные проверки объектов секций. Строки 44-47 соответствующим образом задают поле `ImageSectionObject`. Строки 48-52 инициализируют различные поля в "следующей" позиции стека IRP, снова используя отрицательные смещения. Эти смещения находятся внутри объединения `Parameters`; поскольку нам уже известен старший код функции IRP (`IRP_MJ_SET_INFORMATION`), мы знаем, что будет использоваться член объединения `SetFile`:

```
1: kd> dt nt!_IO_STACK_LOCATION Parameters.SetFile.
   +0x008 Parameters          :
      +0x000 SetFile             :
         +0x000 Length              : Uint4B
         +0x008 FileInformationClass : _FILE_INFORMATION_CLASS
         +0x010 FileObject          : Ptr64 _FILE_OBJECT
         +0x018 ReplaceIfExists     : UChar
         +0x019 AdvanceOnly         : UChar
         +0x018 ClusterCount        : Uint4B
         +0x018 DeleteHandle        : Ptr64 Void
```

Вычислив смещения, мы понимаем, что строка 49 задает поле `DeleteHandle` вторым параметром, строка 50 задает поле `FileObject`, строка 51 - поле `FileInformationClass` (`0xD` означает `FileDispositionInformation`), а строка 52 - поле `Length`. В документации класса `FileDispositionInformation` сказано, что он принимает структуру с однобайтовым полем; если оно равно 1, то файл помечается для удаления. Таким образом, теперь нам понятно, почему строки 13 и 27 присваивают `IRP.AssociatedIrp.SystemBuffer` значение 1. Строки 53-55 задают `sub_4038B4` в качестве процедуры завершения этого IRP. Строка 60 передает вновь заполненный IRP другому драйверу (полученному в строке 16) для обработки (скорее всего, драйверу файловой системы). Строка 61 сравнивает состояние с `STATUS_PENDING`, чтобы определить, завершена ли операция; если да, состояние IRP возвращается в `EAX`; если нет, вызывается `KeWaitForSingleObject`, ожидающая события, инициализированного в строке 37. По завершении процедура завершения установит событие и освободит IRP:

```
01: sub_4038B4 proc near
02:   push    rbx
03:   sub     rsp, 20h
04:   movdqu  xmm0, xmmword ptr [rdx+30h]
05:   mov     rax, [rdx+48h]
06:   mov     rbx, rdx
07:   xor     r8d, r8d       ; Wait
08:   xor     edx, edx       ; Increment
09:   movdqu  xmmword ptr [rax], xmm0
10:   mov     rcx, [rbx+50h] ; Event
11:   call    cs:KeSetEvent
12:   mov     rcx, rbx       ; Irp
13:   call    cs:IoFreeIrp
14:   mov     eax, 0C0000016h
15:   add     rsp, 20h
16:   pop     rbx
17:   retn
18: sub_4038B4 endp
```

Всю процедуру можно декомпилировать следующим образом:

```
NTSTATUS sub_4038F0(PFILE_OBJECT FileObj, HANDLE hdelete, BOOLEAN flag)
{
  NTSTATUS status;
  PIO_STACK_LOCATION iosl;
  PIRP Irp;
  PDEVICE_OBJECT devobj;
  KEVENT event;
  IO_STATUS_BLOCK iosb;
  CHAR buf = 1;

  devobj = IoGetRelatedDeviceObject(FileObj);
  Irp = IoAllocateIrp(devobj->StackSize, FALSE);
  if (Irp == NULL) { return STATUS_NO_MEMORY; }
  Irp->AssociatedIrp.SystemBuffer = &buf;
  Irp->UserEvent = &event;
  Irp->UserIosb = &iosb;
  Irp->Tail.Overlay.Thread = KeGetCurrentThread();
  Irp->Tail.Overlay.OriginalFileObject = FileObj;
  Irp->RequestorMode = KernelMode;
  KeInitializeEvent(&event, SynchronizationEvent, FALSE);

  iosl = IoGetNextIrpStackLocation(Irp);
  iosl->DeviceObject = devobj;
  iosl->FileObject = FileObj;
  if (!flag && FileObj->SectionObjectPointer != NULL) {
    FileObj->SectionObjectPointer.ImageSectionObject = NULL;
  }
  iosl->Parameters.SetFile.FileObject = FileObj;
  iosl->Parameters.SetFile.DeleteHandle = hdelete;
  iosl->Parameters.SetFile.FileInformationClass = \
                                  FileDispositionInformation;
  iosl->Parameters.SetFile.Length = 1;
  IoSkipCurrentIrpStackLocation(Irp);
  IoSetCompletionRoutine(Irp, sub_4038B4, NULL, TRUE, TRUE, TRUE);
  if (IoCallDriver(devobj, Irp) == STATUS_PENDING) {
    KeWaitForSingleObject(&event, Executive, KernelMode, TRUE, NULL);
  }
  return Irp->IoStatus.Status;
}
```

Теперь видно, что драйвер использует эту функцию для удаления файла из системы без API удаления файлов (`ZwDeleteFile`). Для этого он самостоятельно формирует IRP, описывающий операцию удаления файла, и передает его нижележащему драйверу (предположительно файловой системе). Кроме того, он использует процедуру завершения, чтобы получить уведомление о завершении IRP (успешном, неуспешном либо каким-то образом отмененном). Хотя этот способ несколько эзотеричен, он очень полезен, поскольку позволяет обойти защитное ПО, которое пытается обнаруживать удаление файлов путем перехвата системных вызовов.

Этот пошаговый разбор продемонстрировал два главных момента. Во-первых, если вы знаете и понимаете объекты и механизмы, которые драйверы используют для взаимодействия с ядром, задача анализа упрощается. Во-вторых, нужно быть готовым иметь дело с кодом, который выглядит странно из-за агрессивного оптимизатора. Особенно это справедливо для кода x64. Единственный способ совершенствоваться - практиковаться.

Следующие шаги

Мы рассмотрели большинство важных предметно-ориентированных концепций, относящихся к коду режима ядра в Windows. Эти знания можно немедленно применить к задачам обратной разработки драйверов. Однако для большей эффективности полезно понять, как выглядят обычные драйверы в исходном виде. Лучший способ изучить это - исследовать образцы драйверов, входящие в WDK, и/или разрабатывать собственные драйверы. Хотя они не являются руткитами, они демонстрируют правильную структуру и конструкции, применяемые в драйверах.

Куда двигаться дальше? Мы рекомендуем следующее (именно в таком порядке):

• Внимательно прочитайте руководство WDK. Начать можно с раздела "Kernel-Mode Driver Architecture" ("Архитектура драйверов режима ядра"). Поначалу он сбивает с толку, но после прочтения этой главы разобраться будет гораздо проще, поскольку мы обошли все несущественные темы.

• От корки до корки прочитайте книгу Питера Дж. Вискаролы и У. Энтони Мэйсона *Windows NT Device Driver Development* ("Разработка драйверов устройств Windows NT"; главу о DMA и программируемом вводе-выводе можно пропустить).

• Напишите несколько небольших простых драйверов. Затем проанализируйте их в дизассемблере, не заглядывая в исходный код. Обязательно сделайте это как для x86, так и для x64.

• Изучите презентацию Брюса Дэнга и Рольфа Роллеса на Recon 2011 *Decompiling kernel drivers and IDA plugins* ("Декомпиляция драйверов ядра и подключаемые модули IDA").

• Прочитайте документацию по отладчику Microsoft, посвященную полезным расширениям ядра (например, `!process`, `!thread`, `!pcr`, `!devobj`, `!drvobj` и т. д.).

• Прочитайте все статьи, опубликованные в *The NT Insider*, а также статьи о ядре в *Uninformed*. Первый ресурс, вероятно, наиболее полезен для разработки драйверов ядра Windows в целом. Второй в большей степени ориентирован на энтузиастов безопасности.

• Выполните все упражнения в конце этой главы. Абсолютно все. Некоторые могут потребовать значительного времени, поскольку вам придется изучить недокументированные области, не рассмотренные в книге. Чтение и исследование - этапы процесса обучения.

• Откройте двоичный файл ядра Windows в дизассемблере и попытайтесь понять, как работают некоторые распространенные API.

• Читайте форумы http://kernel-mode.info.

• Проанализируйте как можно больше руткитов. В ходе анализа размышляйте о том, почему и как автор руткита решил использовать те или иные объекты и механизмы, и оценивайте уместность такого выбора.

• Найдите и изучите драйверы Windows с открытым исходным кодом.

• Когда вам покажется, что вы хорошо поняли основные концепции, можно исследовать другие области ядра, например сетевой стек и стек хранения. Это две чрезвычайно сложные области, поэтому потребуется много времени и терпения.

• Подпишитесь на списки рассылки NTDEV и NTFSD, чтобы читать о проблемах других разработчиков и о том, как они их решали.

Продолжайте читать, практиковаться и учиться! Кривая обучения крута, но после ее преодоления все пойдет как по маслу. Помните: без неудач трудно оценить успех. Удачных аварийных остановок.

Упражнения

Мы считаем, что лучший способ учиться - сочетать обсуждение концепций, практические руководства и самостоятельные упражнения. Первые два элемента были рассмотрены в предыдущих разделах. Следующие самостоятельные упражнения призваны помочь вам обрести уверенность, закрепить понимание концепций ядра Windows, исследовать и расширить знания в областях, не рассмотренных в книге, а также продолжить анализ реальных драйверов. Как и в других главах, все упражнения взяты из реальных ситуаций. Файлы обозначаются как образец A, B, C и так далее. SHA1-хеш каждого образца приведен в приложении.

Обретение уверенности и закрепление знаний

На каждое из этих упражнений обычно можно ответить за 30 минут. Некоторые могут потребовать дополнительного чтения или размышлений, поэтому на них может уйти больше времени.

1. Объясните, почему код, выполняющийся на уровне `DISPATCH_LEVEL`, не может вызвать страничное исключение. Объяснений может быть несколько. Вы должны суметь привести как минимум два.

2. Предположим, вы прочитали в Интернете статью о ядре Windows, в которой утверждается, что потоки режима ядра всегда имеют более высокий приоритет, чем потоки режима пользователя; следовательно, если написать все в режиме ядра, оно будет работать быстрее. Оцените справедливость этого утверждения, используя свои знания об IRQL, диспетчеризации потоков и приоритетах потоков.

3. Напишите драйвер для Windows 7/8, который выводит базовый адрес каждого вновь загруженного образа. Повторите то же самое для процессов и потоков. Этому драйверу не нужно задавать обработчик IRP, поскольку ему не требуется обрабатывать запросы пользователей или других драйверов.

4. Объясните последствия использования `METHOD_NEITHER` для безопасности и то, что разработчики драйверов делают для их смягчения.

5. Имея виртуальный адрес режима ядра, вручную преобразуйте его в физический адрес. Проверьте ответ с помощью расширения `!vtop` отладчика ядра.

6. Разработайте драйвер, использующий все операции со списками, и найдите все встроенные процедуры работы со списками в ассемблерном виде. Есть ли для каждой процедуры общий шаблон? Если да, объясните их. Если нет, объясните почему.

7. Вы узнали о связных списках, однако ядро также поддерживает хеш-таблицы, деревья поиска и битовые карты. Исследуйте их применение и разработайте драйвер, использующий все эти структуры.

8. Объясните, как работает макрос `FIELD_OFFSET`.
9. Экспортируемая функция `ExGetCurrentProcessorCpuUsage` не документирована, однако документированный API NDIS `NdisGetCurrentProcessorCpuUsage` использует ее внутри. Объясните, как `ExGetCurrentProcessorCpuUsage` работает в Windows x64 и x86.

10. Объясните, как `KeGetCurrentIrql` работает на x86 и x64.

11. Объясните, как следующие API работают в Windows 7/8 на x86/x64/ARM:

• `IoThreadToProcess`

• `PsGetThreadProcessId`

• `PsIsSystemThread`

• `PsGetCurrentThreadId`

• `PsGetCurrentThreadPreviousMode`

• `PsGetCurrentThreadProcess`

• `PsGetCurrentThreadStackBase`

• `PsGetCurrentThreadWin32Thread`

• `PsGetThreadId`

• `PsGetThreadSessionId`

• `PsIsSystemProcess`

• `PsGetProcessImageFileName`

12. Структуры PCR, PRCB, EPROCESS, KPROCESS, ETHREAD и KTHREAD хранят много полезной информации. К сожалению, все они непрозрачны и могут меняться от одной версии Windows к другой. Поэтому многие руткиты жестко задают смещения в этих структурах. Исследуйте эти структуры в Windows XP, 2003, Vista и 7 и отметьте различия. Можете ли вы разработать способы универсального получения смещений некоторых полезных полей без жесткого задания? Если да, можете ли вы сделать так, чтобы они работали на всех перечисленных платформах? (Подсказка: можно использовать дизассемблер, сопоставление шаблонов и относительное расстояние.)

13. API `MmGetPhysicalAddress` принимает виртуальный адрес и возвращает соответствующий физический адрес. Иногда возвращенный физический адрес содержит мусорные данные. Объясните, почему это может происходить и как это смягчить.

14. Настройте режим тестовой подписи на своих 32- и 64-разрядных машинах и подпишите драйвер тестовой подписью. Убедитесь, что он работает.

15. Объясните, как работает `AuxKlibGetImageExportDirectory`. Затем объясните, как работают `RtlImageNtHeader` и `RtlImageDirectoryEntryToData`.

16. Предположим, вы хотите отслеживать появление и завершение процессов. Какую структуру данных вы использовали бы и какие свойства можно применять для однозначной идентификации процесса?
17. Где в процессе хранится таблица каталогов страниц (`CR3` в x86 и `TTBR` в ARM)?

Исследование и расширение знаний

Эти упражнения требуют дополнительного фонового исследования. Возможно, вам придется разрабатывать драйверы, использующие недокументированные API, или обращаться к недокументированным структурам. Знания, полученные в ходе экспериментов, следует использовать только во благо.

1. Многие современные операционные системы поддерживают функцию предотвращения выполнения данных (Data Execution Prevent, DEP). Иногда ее называют Never Execute (NX) или Execute Never (XN). Эта функция просто блокирует выполнение кода на страницах памяти, которые не помечены как исполняемые. Исследуйте, как эта функция реализована в аппаратуре (x86, x64 и

ARM) и как операционная система обеспечивает ее поддержку. Затем исследуйте, как эту функцию можно было бы реализовать без какой-либо аппаратной поддержки.

2. Хотя мы рассмотрели основную идею APC, мы не объяснили, как ими пользоваться. Исследуйте (недокументированные) API, связанные с APC режима ядра, и способы их применения. Напишите драйвер, который их использует.

3. Разработайте и реализуйте как минимум два способа запуска процесса режима пользователя из драйвера режима ядра. Оцените преимущества и недостатки каждого способа.

4. Предположим, вы работаете в SMP-системе с четырьмя процессорами и хотите изменить общий глобальный ресурс. Глобальный ресурс находится в невыгружаемом пуле, и любой процессор может изменить его в любой момент. Разработайте механизм синхронизации, позволяющий безопасно изменить этот ресурс. (Подсказка: подумайте об IRQL и диспетчере потоков.)

5. Напишите драйвер, блокирующий загрузку всех будущих драйверов с именем `bda.sys`.

6. Исследуйте работу стека ввода Windows и реализуйте регистратор нажатий клавиш. Кейлоггер можно реализовать несколькими разными способами (с перехватом и без него). Оцените преимущества и недостатки каждого способа регистрации нажатий. Можно ли определить приложение, которое получает нажатия клавиш?

7. Реализуйте функцию, которая принимает виртуальный адрес и изменяет защиту его страницы на чтение, запись и выполнение. Повторите ту же задачу для виртуального адреса в пространстве сеанса (например, `win32k.sys`).

8. Мы объяснили, что `DriverEntry` - первая функция, вызываемая в драйвере. Объясните, какая функция в действительности вызывает эту процедуру. Как вы это выяснили?
9. Отладчик ядра Microsoft предоставляет механизм, который передает управление отладчику при загрузке драйвера. Для этого используется команда `sxe ld:drivername`. Соберите простой драйвер и поэкспериментируйте с этой командой. Объясните, как она работает. Перечислите все различные причины, по которым она может не сработать.

10. Отладчики режима пользователя могут легко "замораживать" потоки процесса, однако в отладчике ядра такой возможности нет. Разработайте способ замораживания и размораживания потока режима пользователя из ядра.

11. Периодические таймеры используются драйверами для регулярного выполнения определенных действий. Разработайте драйвер, который каждые 10 минут будет выводить "hello". Затем разработайте способ изменения времени срабатывания таймера после его постановки в очередь. Для этого можно использовать отладчик.

12. Реализуйте драйвер, устанавливающий собственный обработчик прерывания, и убедитесь, что его можно активировать из режима пользователя. В Windows x64 вы столкнетесь с PatchGuard, поэтому обязательно проверяйте его только в режиме отладки.

13. Привилегии процессов определяются с помощью маркеров доступа. Наивысшая привилегия - LocalSystem (процесс SYSTEM выполняется в этом контексте). Разработайте драйвер, изменяющий привилегию работающего процесса так, чтобы тот выполнялся с привилегией LocalSystem.

14. Windows Vista и более поздние версии поддерживают криптографические операции в режиме ядра через драйвер KSECDD. Хотя он не документирован в официальном WDK, его описание есть на MSDN в разделе библиотеки bcrypt режима пользователя. Разработайте драйвер, использующий AES, RSA, MD5, SHA1 и генератор случайных чисел.

15. Разработайте драйвер, перечисляющий адреса и имена всех экспортируемых символов в NTDLL, KERNEL32 и KERNELBASE. Повторите то же самое для USER32 и GDI32. Столкнулись ли вы с какими-либо трудностями? Если да, как вы их устранили?

16. Разработайте драйвер, который перехватывает экспортируемую функцию NTDLL в процессе `explorer.exe`. Оцените достоинства своего метода. Исследуйте и оцените другие методы.

17. Разработайте драйвер, который присоединяется к процессу SMSS.EXE и исправляет системный вызов `win32k`, находясь в контексте этого процесса. Объясните, с какими проблемами вы столкнулись и как их решили.

18. Предположим, кто-то утверждает, что исключения режима пользователя никогда не переходят в ядро. Исследуйте, как в Windows x86 и x64 работает обработка исключений режима пользователя, и оцените приведенное утверждение.

19. Предположим, в системе имеется вредоносный драйвер, который перехватывает `INT 1` и `INT 3`, чтобы затруднить отладку и трассировку. Разработайте способ получить

трассу выполнения (или отладить код), даже когда эти перехваты действуют. Никаких ограничений у вас нет. Какие особые случаи необходимо обработать?

20. Команда `INT 3` может быть представлена в двух формах. Наиболее распространена однобайтовая форма `0xCC`. Менее распространенная двухбайтовая форма - `0xCD03`. Объясните, что происходит при использовании двухбайтовой формы в Windows.

Анализ реальных драйверов

Эти упражнения предназначены для практики аналитических навыков на реальном драйвере. Мы приводим хеши файлов и задаем вопросы о них. На большинство вопросов (если не на все) можно ответить посредством статического анализа, однако при необходимости вы можете запустить образец.

1. (Образец D.) Проанализируйте и объясните, что делает функция `0x10001277`. Откуда берется второй аргумент и может ли он когда-либо оказаться недопустимым? Что делают функции по смещениям `0x100012B0` и `0x100012BC`?

2. (Образец E.) Этот файл довольно велик и сложен; некоторые его структуры огромны (их размер близок к 4000 байт). Однако в нем есть функции, выполняющие интересные задачи, рассмотренные в этой главе, поэтому из него взято несколько упражнений. Для этого упражнения восстановите прототипы функций `0x40400D`, `0x403ECC`, `0x403FAD`, `0x403F48`, `0x404088`, `0x4057B8`, `0x404102` и `0x405C7C`, а также объясните различия и связи между ними (если они есть); расскажите, как вы пришли к решению. Затем объясните значение выделения 30-байтового невыгружаемого пула в функциях `0x403F48`, `0x403ECC` и `0x403FA`; заодно восстановите его тип. Кроме того, объясните, почему в некоторых из предыдущих процедур в начале выполняется освобождение пула. Эти процедуры используют недокументированные функции, поэтому вам, возможно, придется поискать прототип в Интернете.

3. (Образец E.) В `DriverEntry` найдите все системные рабочие потоки. По смещению `0x402C12` создается системный поток, который делает нечто обыденное интересным способом. Проанализируйте и объясните назначение функции `0x405775` и всех вызываемых ею функций. В частности, объясните механизм, используемый в функции `0x403D65`. Поняв механизм, напишите драйвер, выполняющий тот же трюк (но применительно к другому запросу ввода-вывода). Завершите упражнение декомпиляцией всех четырех процедур. Это упражнение очень поучительно и принесет вам большую пользу.

4. (Образец E.) Функция `0x402CEC` принимает в качестве одного из параметров объект устройства, связанный с `\Device\Disk\DR0`, и отправляет ему запрос с помощью `IoBuildDeviceIoControlRequest`. Этот объект устройства описывает первый раздел вашего загрузочного диска. Декодируйте используемый IOCTL и найдите его содержательное имя. (Подсказка: выполните поиск по всем файлам, входящим в WDK, включая файлы режима пользователя.) Определите структуру, связанную с этим запросом.

Затем приведите вывод IDA в порядок, назначив каждой локальной переменной тип и содержательное имя. Наконец, декомпилируйте процедуру обратно в C и объясните, что она делает (возможно, даже напишите другой драйвер, использующий этот метод).

5. (Образец E.) Декомпилируйте функцию `0x401031` и дайте ей содержательное имя. Если вы не знакомы с работой SCSI, рекомендуется прочитать *SCSI Commands Reference Manual* ("Справочное руководство по командам SCSI").

6. (Образец F.) Объясните, что делает функция `0x100051D2` и почему. Чем так примечательно смещение `0x38` в структуре расширения устройства? Восстановите как можно больше типов и декомпилируйте эту процедуру. Наконец, найдите все таймеры, DPC и рабочие элементы, используемые драйвером.

Глава 4. Отладка и автоматизация

Отладчики - это программы, использующие поддержку процессора и операционной системы для трассировки других программ, чтобы можно было находить ошибки или просто понимать логику отлаживаемой программы. Отладчики - важнейший инструмент специалистов по обратной разработке, поскольку, в отличие от дизассемблеров, позволяют исследовать состояние программы во время выполнения.

Цель этой главы - познакомить вас с бесплатными средствами отладки Microsoft. Она не предназначена для обучения приемам отладки или поиску причин утечек памяти, взаимных блокировок и тому подобного. Вместо этого основное внимание уделено важнейшим командам и средствам автоматизации/написания сценариев, а также созданию расширений отладчика исключительно для помощи в задачах обратной разработки.

В главе рассматриваются следующие темы:

• Средства отладки и основные команды - в этом разделе рассматриваются основы отладки, различные команды, вычисление выражений и операторы, команды для работы с процессами и потоками, а также операции с памятью.
• Написание сценариев - язык сценариев движка отладчика не отличается дружелюбием к пользователю. В этом разделе язык объясняется структурированно и понятно, приводятся различные примеры и набор сценариев, иллюстрирующих каждую тему. Прочитав этот раздел, вы начнете использовать мощь сценариев в отладчике.

• Использование SDK - когда сценариев недостаточно, всегда можно написать расширения на C или C++. В этом разделе изложены основы создания расширений на C/C++.

Средства отладки и основные команды

Пакет Debugging Tools for Windows представляет собой набор отладочных утилит, который можно бесплатно загрузить с веб-сайта Microsoft. В комплект входят четыре отладчика, основанные на одном и том же движке отладки (DbgEng).

DbgEng - это COM-объект, позволяющий другим программам использовать расширенные API отладки, а не только обычные Windows Debugging APIs. Более того, пакет Debugging Tools включает SDK, показывающий, как создавать расширения для DbgEng или размещать его в собственных программах.

Пакет Debugging Tools for Windows включает следующие отладчики:

• NTSD/CDB - Microsoft NT Symbolic Debugger (NTSD) и Microsoft Console Debugger (CDB) идентичны за исключением того, что первый при запуске создает новое окно консоли, а второй наследует окно консоли, из которого был запущен.

• WinDbg - графический интерфейс к DbgEng. Он поддерживает отладку на уровне исходного кода и сохранение рабочих пространств.

• KD - Kernel Debugger (KD) используется для отладки ядра.

У отладчиков есть богатый набор параметров командной строки. Один из особенно полезных параметров - `-z`, применяемый для анализа аварийных дампов (`*.dmp`) и CAB-файлов (`*.cab`), содержащих файл аварийного дампа. Еще одно применение параметра `-z` - анализ PE-файлов (исполняемых файлов или DLL), при котором DbgEng отображает их так, словно они находятся в аварийном дампе.

В следующем примере отладчик CDB запускается с параметром `-z`, чтобы отобразить `calc.exe` в отладчике:

```
C:\>cdb -z c:\windows\syswow64\calc.exe

Microsoft (R) Windows Debugger Version 6.13.0009.1140 X86
Copyright (c) Microsoft Corporation. All rights reserved.

Loading Dump File [c:\windows\syswow64\calc.exe]
Symbol search path is: SRV*C:\cache*http://msdl.microsoft.com/download/
symbols

Executable search path is:
ModLoad: 00400000 004c7000   c:\windows\syswow64\calc.exe
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=0041a592 esp=00000000 ebp=00000000 iopl=0         nv up di pl nz na po nc
cs=0000  ss=0000  ds=0000  es=0000  fs=0000  gs=0000             efl=00000000
calc!WinMainCRTStartup:
0041a592 e84bf0ffff      call    calc!__security_init_cookie (004195e2)
0:000>
```

Обратите внимание на две вещи:

• `Calc.exe` был отображен в отладчике, а `EIP` указывает на его точку входа (в отличие от живых целевых процессов, где он указывает внутрь `ntdll.dll`).

• Многие команды отладчика будут недоступны, особенно команды управления процессом (поскольку программа отображена для анализа/исследования, а не для динамической трассировки/отладки).

Используя параметр `-z`, можно писать мощные сценарии для анализа программ и извлечения информации.

> **Примечание.** WinDbg можно настроить в качестве оперативного (just-in-time, JIT) отладчика (для посмертной отладки), один раз запустив `Windbg.exe -I` от имени привилегированного пользователя.

В следующих разделах объясняются различные команды отладчика и по ходу приводятся примеры.

Настройка пути к символам

Перед запуском любого из отладчиков (WinDbg, CDB, NTSD или KD) настроим переменную среды `_NT_SYMBOL_PATH`:

```
_NT_SYMBOL_PATH=SRV*c:\ cache*http://msdl.microsoft.com/download/symbols
```

Ее также можно настроить из самого отладчика при помощи команды `.sympath`:

> **Примечание.** Настройка пути к символам важна, чтобы при отладке рассматриваемых программ можно было исследовать некоторые базовые структуры ОС. Например, команда расширения `!peb` не будет работать без загруженных символов NTDLL.

Окна отладчика

В WinDbg доступны следующие окна (в соответствующих случаях указаны сочетания клавиш):

• Окно команд/вывода (`Alt+1`) - в этом окне можно вводить команды и просматривать результаты операций. Хотя выполнять отладку можно с помощью других окон и пунктов меню, окно команд позволяет использовать всю мощь встроенных команд DbgEng и доступных расширений.
• Окно регистров (`Alt+4`) - отображает настроенные регистры. Это представление можно настроить, указав, какие регистры показывать или скрывать.

• Память (`Alt+5`) - окно дампа памяти. В нем можно просматривать содержимое памяти, прокручивать, копировать и даже редактировать его.

• Вызовы (`Alt+6`) - отображает сведения о стеке вызовов.

• Дизассемблирование (`Alt+7`) - окно команд показывает дизассемблерный листинг текущей команды, тогда как окно дизассемблирования отображает целую страницу дизассемблированного кода. В этом окне также можно выполнять действия с помощью сочетаний клавиш:

• добавлять или удалять точки останова на выбранной строке (`F9`);

• управлять процессом (пошаговое выполнение - `F11`, продолжение - `F5` и т. д.);

• перемещаться по коду (`Page Up`/`Page Down` для исследования дизассемблированного кода).

> **Примечание.** WinDbg поддерживает рабочие пространства, позволяющие сохранять и восстанавливать конфигурацию окон.

Вычисление выражений

Отладчик понимает два синтаксиса вычисления выражений: Microsoft Macro Assembler (MASM) и C++.

Чтобы определить используемый по умолчанию вычислитель выражений, выполните `.expr` без аргументов:

```
0:000> .expr
Current expression evaluator: MASM - Microsoft Assembler expressions
```

Чтобы изменить текущий синтаксис вычисления выражений, используйте:

```
0:000> .expr /s c++
Current expression evaluator: C++ - C++ source expressions
```

или:

```
0:000> .expr /s masm
Current expression evaluator: MASM - Microsoft Assembler expressions
```

Для вычисления выражений (в синтаксисе по умолчанию) используется команда `?`.

Команда `??` служит для вычисления выражения C++ (независимо от выбранного синтаксиса по умолчанию).

> **Примечание.** Синтаксис C++ предпочтителен, когда доступны сведения о типах/символах и требуется обращаться к членам структур или просто использовать операторы C++.

Числа без префикса, задающего основание, интерпретируются с использованием текущей системы счисления по умолчанию. Команда `n` показывает текущее основание системы счисления, а `n base_value` задает новое основание по умолчанию.

При использовании синтаксиса MASM число можно записать в выбранной системе счисления при помощи следующих префиксов:

• `0n123` - десятичное число;

• `0x123` - шестнадцатеричное число;

• `0t123` - восьмеричное число;

• `0y10101` - двоичное число.

В отличие от вычисления в синтаксисе MASM, при использовании `??` для вычисления команд переопределить основание нельзя:

```
? 0y101  -> работает
?? 0y101 -> не работает.
```

> **Примечание.** Когда основание системы счисления по умолчанию равно 16, при попытке вычислить выражение вроде `abc` возникает неоднозначность: это может быть символ с именем `abc` или шестнадцатеричное число `abc` (2748 в десятичной системе). Чтобы вместо этого разрешить символ, поставьте перед именем переменной `!`: `? !abc`.

Как и в языке C++, синтаксис вычислителя C++ допускает только префикс `0x` для шестнадцатеричных чисел и префикс `0` для восьмеричных. Если префикс не задан, используется основание 10.

Для сочетания выражений разных типов применяются формы `@@c++(expression)` и `@@masm(expression)`:

```
0:000> .expr
Current expression evaluator: MASM - Microsoft Assembler expressions
0:000> ? @@c++(@$peb->ImageSubsystemMajorVersion) + @@masm(0y1)
Evaluate expression: 7 = 00000007
```

Префикс `@@` - сокращенная форма, которой можно обозначить альтернативный синтаксис вычисления выражений (не тот, который установлен в данный момент):

```
0:000> .expr
Current expression evaluator: MASM - Microsoft Assembler expressions
0:000> ? @@(@$peb->ImageSubsystemMajorVersion) + @@masm(0y1)
Evaluate expression: 7 = 00000007
```

Указывать `@@c++(...)` необязательно: когда по умолчанию выбран MASM, форма `@@(...)` использует синтаксис C++, и наоборот.

Полезные операторы

В этом разделе показаны различные полезные операторы, которые можно применять в выражениях. Для демонстрации используются предопределенные псевдорегистры `$ip` и `$peb`, обозначающие соответственно текущий указатель команд и `_PEB *` текущего процесса. Другие псевдорегистры упоминаются далее в этой главе.

Используется обозначение "оператор (синтаксис выражения)", где синтаксисом выражения будет либо C++, либо MASM. Обратите внимание: в следующих примерах по умолчанию установлен вычислитель выражений MASM.

• `Pointer->Field` (C++) - как и в предыдущем примере, оператор-стрелка используется для доступа к значению поля, на которое указывает `$peb`, и смещению поля `ImageSubsystemMajorVersion`.

• `sizeof(type)` (C++) - этот оператор возвращает размер структуры. Он может пригодиться при разборе структур данных или создании мощных условных точек останова:

```
0:000> ? @@c++(sizeof(_PEB))
Evaluate expression: 592 = 00000250
```

• `#FIELD_OFFSET(Type, Field)` (C++) - этот макрос возвращает байтовое смещение поля в типе:

```
0:000> ? #FIELD_OFFSET(_PEB, ImageSubsystemMajorVersion)
Evaluate expression: 184 = 000000b8
```

• Тернарный оператор (C++) - он ведет себя так же, как в языке C++:

```
0:000> ? @@c++(@$peb->ImageSubsystemMajorVersion >= 6 ? 1 : 0)
Evaluate expression: 1 = 00000001
```

• `(type) Value` (C++) - приведение типов позволяет преобразовывать один тип в другой:

```
0:000> ? #FIELD_OFFSET(_PEB, BeingDebugged)
Evaluate expression: 2 = 00000002
0:000> ? @$peb
Evaluate expression: 2118967296 = 7e4ce000
0:000> ? #FIELD_OFFSET(_PEB, BeingDebugged) + (char *)@$peb
Evaluate expression: 2118967298 = 7e4ce002
```

Обратите внимание: перед прибавлением смещения `BeingDebugged` значение `@$peb` приводится к `(char*)`.

• `*(pointer)` (C++) - оператор разыменования:

```
0:000> dd @$ip L 4
012a9615  2ec048a3 8b5e5f01 90c35de5 90909090
0:000> ? *( (unsigned long *)0x12a9615 )
Evaluate expression: 784353443 = 2ec048a3
```

Обратите внимание: перед разыменованием указателю нужно назначить правильный тип (путем приведения).

• `poi(address)` (MASM) - разыменование указателя:

```
0:000> ? @@masm(poi(0x12a9615))
Evaluate expression: 784353443 = 2ec048a3
```

• `hi|low(number)` (MASM) - возвращает старшее или младшее 16-разрядное значение числа:

```
0:000> ? hi(0x11223344)
Evaluate expression: 4386 = 00001122
0:000> ? low(0x11223344)
Evaluate expression: 13124 = 00003344
```

• `by/wo/dwo(address)` (MASM) - возвращает значение типа byte/word/dword при разыменовании адреса:

```
0:000> db @$ip L 4
012a9615  a3 48 00 00
0:000> ? by(@$ip)
Evaluate expression: 163 = 000000a3
0:000> ? wo(@$ip)
Evaluate expression: 18595 = 000048a3
0:000> ? dwo(@$ip)
Evaluate expression: 18595 = 000048a3
```

• `pointer[index]` (C++) - оператор индексирования массива позволяет разыменовывать память с использованием индексов:

```
0:000> db @$ip L 10
012a9615  a3 48 c0 2e 01 5f 5e 8b e5 5d
0:000> ? @@c++(((unsigned char *)@$ip)[3])
Evaluate expression: 46 = 0000002e
```

Того же результата можно добиться с помощью синтаксиса MASM и `poi()` или `by()`:

```
0:000> ? poi(@$ip+3) & 0xff
Evaluate expression: 46 = 0000002e
0:000> ? by(@$ip+3)
Evaluate expression: 46 = 0000002e
```

> **Примечание.** При использовании `pointer[index]` учитывается размер базового типа (в отличие от `poi()`, где размер типа приходится учитывать самостоятельно).

• `$scmp("string1", "string2")`/`$sicmp("String1", "String2")` (MASM) - сравнение строк (с учетом/без учета регистра). Возвращает -1, 0 или 1, как функции C `strcmp()`/`stricmp()`:

```
0:000> ? $scmp("practical", "practica")
Evaluate expression: 1 = 00000001
```

```
0:000> ? $scmp("practical", "practical")
Evaluate expression: 0 = 00000000
0:000> ? $scmp("practica", "practical")
Evaluate expression: -1 = ffffffff
0:000> ? $scmp("Practical", "practical")
Evaluate expression: -1 = ffffffff
0:000> ? $sicmp("Practical", "practical")
Evaluate expression: 0 = 00000000
```

• `$iment(address)` (MASM) - возвращает точку входа образа, находящегося по этому адресу. Для этого разбирается и используется заголовок PE:

```
0:000> lmvm ole32
start    end        module name
74b70000 74c79000   ole32
...
0:000> ? $iment(74b70000)
Evaluate expression: 1958154432 = 74b710c0
0:000> u $iment(74b70000)
ole32!_DllMainCRTStartup:
74b710c0 8bff            mov     edi,edi
74b710c2 55              push    ebp
74b710c3 8bec            mov     ebp,esp
```

• `$vvalid(address, length)` (MASM) - проверяет доступность памяти от указанного адреса до `address + length` (возвращает 1, если она доступна, и 0, если недоступна):

```
0:000> ? @@masm($vvalid(@$ip, 100))
Evaluate expression: 1 = 00000001
0:000> ? @@masm($vvalid(0x0, 100))
Evaluate expression: 0 = 00000000
```

• `$spat("string", "pattern")` (MASM) - использует сопоставление с шаблоном, чтобы определить, присутствует ли шаблон в строке, и возвращает истину или ложь.

Управление процессом и события Debut

В этом разделе представлены основные команды управления процессом (например, выполнение одной команды, шаг с обходом и т. д.) и команды, с помощью которых можно изменить реакцию отладчика на определенные события отладки.

Управление процессами и потоками

Ниже перечислены некоторые команды, позволяющие управлять ходом выполнения в отладчике:

• `t` (`F11`) - шаг с заходом.

• `gu` (`Shift+F11`) - переход вверх. Выходит из текущей функции и возвращается к вызывающей.
• `p` (`F10`) - шаг с обходом.

• `g` (`F5`) - продолжить. Возобновляет выполнение программы.

• `Ctrl+Break` - когда отлаживаемая программа выполняется, это сочетание клавиш приостанавливает ее.

Обратите внимание: приведенные команды работают только с живыми целями.

Существуют полезные варианты команд "продолжить", "шаг с заходом" и "шаг с обходом", в том числе следующие:

• `[t|p]a Address` - шаг с заходом. Выполняет шаги с обходом, пока не будет достигнут указанный адрес.

• `gc` - применяется для возобновления выполнения после того, как условная точка останова приостановила его.

• `g[h|n]` - применяется для возобновления выполнения с пометкой исключения как обработанного или необработанного.

Еще один набор команд трассировки/пошагового выполнения полезен для обнаружения базовых блоков:

• `[p|t]c` - выполнять шаги с обходом/заходом до встречи команды `CALL`.

• `[p|t]h` - выполнять шаги с обходом/заходом до встречи команды ветвления (любого перехода, возврата или вызова).

• `[p|t]t` - выполнять шаги с обходом/заходом до встречи команды `RET`.

• `[p|t]ct` - выполнять шаги с обходом/заходом до встречи команды `CALL` или `RET`.

Большинство приведенных команд (трассировки и шагов с обходом) неявно работает в контексте текущего потока.

Чтобы вывести все потоки, используйте команду ~:

```
0:004> ~
   0  Id: 1224.13d8 Suspend: 1 Teb: ff4ab000 Unfrozen
   1  Id: 1224.1758 Suspend: 1 Teb: ff4a5000 Unfrozen
   2  Id: 1224.2920 Suspend: 1 Teb: ff37f000 Unfrozen
   3  Id: 1224.1514 Suspend: 1 Teb: ff37c000 Unfrozen
. 4  Id: 1224.b0 Suspend: 1 Teb: ff2f7000 Unfrozen
```

Первый столбец - номер потока (назначенный DbgEng), за которым следует пара `SystemProcessId.SystemThreadId` в шестнадцатеричном формате.

Команды DbgEng работают с идентификаторами DbgEng, а не с идентификаторами процессов/потоков операционной системы.

Чтобы переключиться на другой поток, используйте команду `~Ns`, где `N` - номер потока, на который требуется переключиться:

```
0:004> ~1s
eax=00000000 ebx=00bb1ab0 ecx=00000000 edx=00000000 esi=02faf9ec edi=00b2ec00
eip=7712c46c esp=02faf8a4 ebp=02fafa44 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202

ntdll!NtWaitForWorkViaWorkerFactory+0xc:
7712c46c c21400          ret     14h
0:001>
```

В приглашении отладчика также отображается выбранный идентификатор потока в форме `ProcessID:ThreadId>`.

Перед выполнением команды переключаться на поток необязательно. Например, чтобы вывести регистры потока с идентификатором 3, используйте префикс `~3`, за которым следует нужная команда отладчика (в данном случае команда `r`):

```
0:001> ~3r
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000001 edi=00000001
eip=7712af2c esp=031afb38 ebp=031afcb8 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
ntdll!NtWaitForMultipleObjects+0xc:
7712af2c c21400          ret     14h
0:001> ~3t
eax=00000000 ebx=00000000 ecx=77072772 edx=00000000 esi=00000001 edi=00000001
eip=758c11b5 esp=031afb50 ebp=031afcb8 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
KERNELBASE!WaitForMultipleObjectsEx+0xdc:
758c11b5 8bf8            mov     edi,eax
```

Чтобы вывести значения регистров всех потоков, просто передайте * в качестве номера потока.

> **Примечание.** Не ко всем командам отладчика можно добавить префикс `~N cmd`, чтобы они выдавали сведения о потоке N. Вместо этого используйте команду для конкретного потока `~eN cmd`.

Если вы отлаживаете несколько процессов режима пользователя (то есть отладчик запущен с параметром `-o`), переключаться между процессами можно командой `|`. В следующем примере используется Internet Explorer, поскольку обычно он порождает несколько дочерних процессов (с разными уровнями целостности и для разных целей):

```
C:\ dbg64>windbg -o "c:\Program Files (x86)\Internet Explorer\iexplore.exe"
```

Позвольте ему запуститься, откройте несколько вкладок, затем возобновите выполнение отладчика командой g, после чего приостановите его и введите `|`:

```
0:030> |
.  0   id: 1818     child  name: iexplore.exe
   1   id: 1384     child  name: iexplore.exe
```

Чтобы переключиться с одного процесса на другой, введите `|Ns`, где `N` - номер процесса:

```
0:030> |1s
1:083> |
#  0   id: 1818     child  name: iexplore.exe
.  1   id: 1384     child  name: iexplore.exe
```

После переключения на новый процесс все последующие команды будут применяться к нему. Точки останова, заданные для одного процесса, в другом процессе отсутствуют.

> **Примечание.** Псевдонимы и псевдорегистры будут общими для всех отлаживаемых процессов.

Наблюдение за событиями отладки и исключениями

Можно перехватывать определенные события отладки и исключения по мере их возникновения, позволяя отладчику приостановиться, вывести событие, обработать его, оставить необработанным или полностью проигнорировать.

DbgEng может приостановить целевой процесс и дать пользователю возможность решить, какое действие предпринять, в следующих двух случаях:

• Исключения - эти события возникают, когда в контексте приложения срабатывает исключение (нарушение доступа, деление на ноль, исключение одиночного шага и т. д.).

• События - эти события не являются ошибками: операционная система инициирует их, чтобы уведомить отладчик об определенной происходящей активности (создан или завершен новый поток, загружен или выгружен модуль, создан или завершен новый процесс и т. д.).

Чтобы вывести все события, используйте команду `sx`. Если вы работаете в WinDbg, можно также перейти в меню Debug/Event Filters и настроить события графически, как показано на рисунке 4-1.

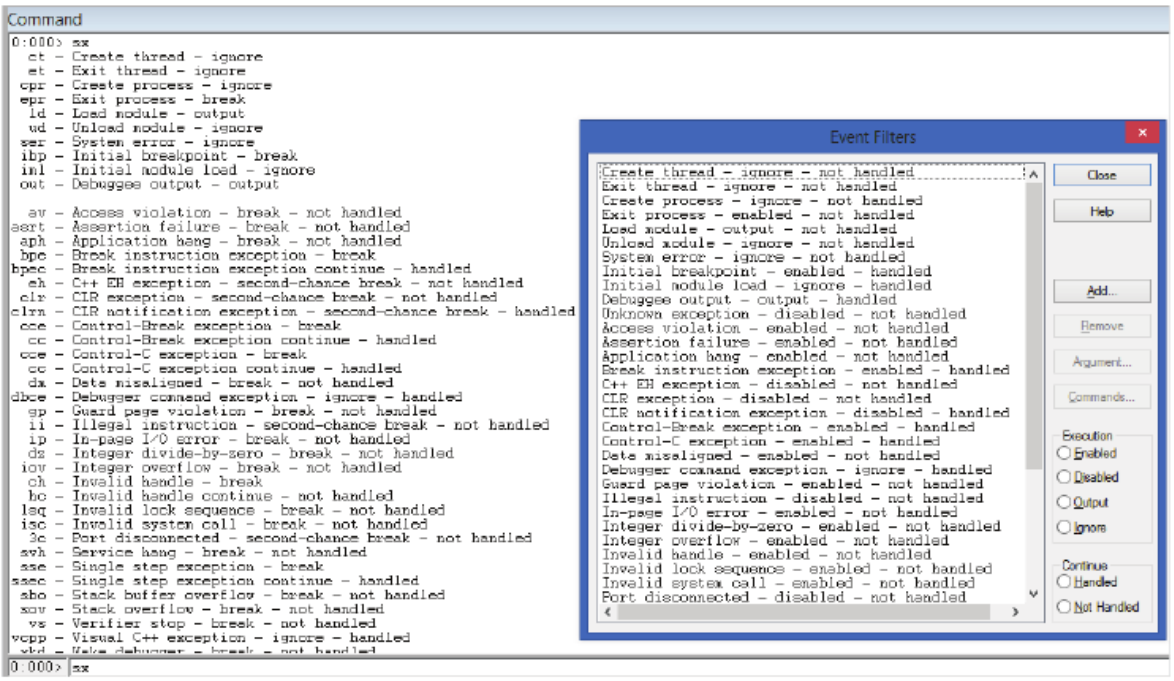

Figure 4-1

Рисунок 4-1. Окно Event Filters и вывод команды sx

На снимке экрана показаны два набора параметров для управления событиями:

• Execution - определяет, что делать при возникновении события.

• Continue - определяет способ продолжения после события или исключения.

• Handled - помечает исключение как обработанное (обработчик исключений приложения не сработает). Это полезно, когда отладчик останавливается, вы вручную исправляете ситуацию, а затем возобновляете приложение командой gh.

• Not Handled - позволяет обработчику исключений приложения обработать исключение. Для продолжения используйте команду gn.

Для управления обработкой событий/исключений используются следующие команды:

• `sxe event` - включает остановку при событии;
• `sxd event` - отключает остановку при событии;
• `sxr event` - включает только вывод события;
• `sxi event` - игнорирует событие (ничего не выводит).

Параметром `event` может быть числовой код исключения, сокращенное имя события или *, означающая любое событие.

Одно из довольно полезных применений команд `sxe` и `sxd` - перехват загрузки или выгрузки модулей. Например, при отладке ядра, чтобы остановить отладчик при загрузке определенного драйвера, используйте следующую команду:

```
sxe ld:driver_name.sys
```

Чтобы связать с событием команду, используйте команду `sx- -c command event`. Например, чтобы выводить стек вызовов при каждой загрузке модуля, используйте:

```
sx- -c "k" ld
```

Регистры, память и символы

В этом разделе рассматриваются некоторые полезные команды для управления регистрами, исследования и изменения содержимого памяти, работы с символами и структурами, а также другие удобные команды.

Регистры

Команда `r` применяется для вывода или изменения значений регистров.

> **Примечание.** Командой `r` также можно изменять значения псевдонимов с фиксированными именами и псевдорегистров. Это применение рассматривается в последующих разделах.

Общий синтаксис команды `r` выглядит следующим образом:

```
r[M Mask|F|X] [RegisterName_Or_FlagName[:[Num]Type] [=[Expression_Or_Value]]]
```

Вот простейший синтаксис команды `r`:

```
r RegisterName|FlagName [= Expression_Or_Value ]
```

Если выражение или значение опущено, `r` выводит текущее значение регистра:

```
0:001> r eax
eax=7ffda000
0:001> r eax = 2
0:001> r eax
eax=00000002
```

Чтобы вывести регистры, задействованные в текущей команде, используйте `r.`:

```
0:000> u rip L1
00007ff6`f54d6470 48895c2420      mov     qword ptr [rsp+20h],rbx
0:000> r.
rsp=000000c9`e256fbb8  rbx=00000000`00000000
0:000> u eip L1
user32!MessageBoxA+0x3:
773922c5 8bec            mov     ebp,esp
0:000> r.
ebp=0018ff98  esp=0018ff78
```

Маски регистров

К команде `r` можно добавить суффикс `M`, за которым следует 32-разрядное значение маски. Маска определяет, какие регистры выводятся, когда `r` вводится без параметров. В таблице 4-1 приведен краткий перечень значений маски.

Таблица 4-1. Значения маски регистров

Значение маски регистров	Описание
`2`	Регистры общего назначения
`4`	Регистры с плавающей запятой
`8`	Сегментные регистры
`0x10`	MMX
`0x20`	Отладочные регистры
`0x40`	SSE XMM
`0x80`	Режим ядра: управляющие регистры
`0x100`	Режим ядра: TSS

Примечание. Для объединения нескольких масок используйте оператор OR (`|`).

Чтобы увидеть текущую маску, введите `rm`:

```
0:000> rm
Register output mask is a:
       2 - Integer state (64-bit)
       8 - Segment registers
```

Теперь при выполнении `r` должны выводиться только регистры общего назначения и сегментные регистры:

```
eax=025ad9d4 ebx=00000000 ecx=7c91056d edx=00ba0000 esi=7c810976 edi=10000080
eip=7c810978 esp=025ad780 ebp=025adbec iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
```

Чтобы вывести все возможные регистры, задайте единицами все биты параметра маски (маска `0x1ff`):

```
kd> rM1ff
eax=025ad9d4 ebx=00000000 ecx=7c91056d edx=00ba0000 esi=7c810976 edi=10000080
eip=7c810978 esp=025ad780 ebp=025adbec iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
fpcw=027F: rn 53 puozdi  fpsw=0000: top=0 cc=0000 --------  fptw=FFFF
fopcode=0000  fpip=0000:00000000  fpdp=0000:00000000
st0= 0.000000000000000000000e+0000  st1= 0.303405511757512497160e-4933
st2=-3.685298464319287816590e-4320  st3= 0.000000015933281407050e-4357
st4=-0.008610620845784322250e-4310  st5= 0.000000125598791309870e-4184
st6=-0.008011795206688037930e+0474  st7=-1.#QNAN0000000000000000e+0000
mm0=0000000000000000  mm1=0127b52000584c8e
mm2=2390ccb400318a24  mm3=000000057c910732
mm4=003187cc00000000  mm5=000000117c910732
mm6=003187ec00000000  mm7=7c9107387c90ee18
xmm0=1.79366e-043 0 6.02419e+036 6.02657e+036
xmm1=0 3.08237e-038 3.08148e-038 0
xmm2=3.30832e-029 5.69433e-039 0 3.08147e-038
xmm3=5.6938e-039 0 9.62692e-043 5.69433e-039
xmm4=3.04894e-038 2.12997e-042 3.07319e-038 5.69433e-039
xmm5=5.69528e-039 6.02651e+036 4.54966e-039 1.16728e-042
```

```
xmm6=5.69567e-039 0 5.69509e-039 6.02419e+036
xmm7=4.54901e-039 5.69575e-039 0 5.69559e-039
cr0=8001003b cr2=7c99a3d8 cr3=07f40280
dr0=00000000 dr1=00000000 dr2=00000000
dr3=00000000 dr6=ffff4ff0 dr7=00000400 cr4=000006f9
gdtr=8003f000   gdtl=03ff idtr=8003f400   idtl=07ff tr=0028  ldtr=0000
```

> **Примечание.** Некоторые регистры процессора (GDT, IDT, управляющие регистры и т. д.) можно вывести только при отладке в режиме ядра.

Чтобы задать маску по умолчанию, используйте команду `rm`, после которой укажите нужное значение маски:

```
0:000> rm 2|4|8
0:000> rm
Register output mask is f:
       2 - Integer state (64-bit)
       4 - Floating-point state
       8 - Segment registers
```

DbgEng предоставляет сокращенные флаги для некоторых масок, а именно для регистров с плавающей запятой и MMX.

Чтобы вывести регистры с плавающей запятой, используйте `rF`, а для вывода регистров XMM - `rX`:

```
0:000> rF
fpcw=027F: rn 53 puozdi  fpsw=4020: top=0 cc=1000 --p-----  fptw=FFFF
fopcode=0000  fpip=0023:74b785bc  fpdp=002b:00020a84
st0= 0.00000000000000000000e+0000  st1= 0.00000000000000000000e+0000
...
0:000> rX
xmm0=0 0 0 0
xmm1=0 0 0 0
xmm2=0 0 0 0
...
```

Формат вывода регистров

Можно указывать, как именно должны выводиться регистры. Во многих случаях это очень полезно, как показывают следующие примеры.

Вывод регистров в форматах с плавающей запятой

Предположим, во время отладки вы заметили, что регистр `eax` содержит значение с плавающей запятой:

```
0:000> r eax
eax=3f8ccccd
```

Чтобы вывести его правильно, используйте:

```
0:000> r eax:f
eax=1.1
```

Чтобы вывести содержимое `rax` как значение с плавающей запятой двойной точности, используйте:

```
0:000> r rax
rax=4014666666666666
0:000> r rax:d
rax=5.1
```

Вывод регистров в форматах byte/word/dword/qword

Когда регистры участвуют в передаче данных, полезно видеть отдельные байты регистра:

```
msvcrt!memcpy+0x220:
00007ff9`5f671a5d f30f7f40f0      movdqu  xmmword ptr [rax-10h],xmm0
0:000> r xmm0
xmm0=           0 1.05612e-038 1.01939e-038 1.00102e-038
0:000> r xmm0:ub
xmm0=00 00 00 00 00 73 00 6c 00 6f 00 62 00 6d 00 79
0:000> rX xmm0:uw
xmm0=0000 0000 0073 006c 006f 0062 006d 0079
0:000> rX xmm0:ud
xmm0=00000000 0073006c 006f0062 006d0079
0:000> rX xmm0:uq
xmm0=000000000073006c 006f0062006d0079
```

В предыдущем примере `memcpy()` использует регистры XMM для передачи по 16 байт за раз. Формат `ub` используется для вывода содержимого `xmm0` в виде беззнаковых байтов, `uw` - в формате слов, `ud` - в формате двойных слов, а `uq` - в формате учетверенных слов. Для вывода в знаковом формате вместо префикса `u` используйте `i`.

Команда вывода селектора

Команда вывода селектора имеет следующий синтаксис:

```
dg FirstSelector [LastSelector]
```

Она выводит сведения о заданном селекторе (или диапазоне селекторов). В данном случае нас интересуют значения селекторов, установленные в одном из регистров x86/x64, а именно в регистрах `cs`, `ds`, `ss`, `gs` и `fs`.

Селекторы используются в сегментной части адреса в защищенном режиме.

В следующем примере команда `dg` выполняется соответственно для `cs`, `ds`, `ss`, `gs` и `fs`:

```
0:001> .foreach /s (sel "cs ds ss gs fs") { dg sel; }
(cs Selector)
                                  P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0023 00000000 ffffffff Code RE Ac 3 Bg Pg P  Nl 00000cfb
(ds Selector)
                                  P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
002B 00000000 ffffffff Data RW Ac 3 Bg Pg P  Nl 00000cf3
(ss Selector)

                                  P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
```

```
002B 00000000 ffffffff Data RW Ac 3 Bg Pg P  Nl 00000cf3
(gs Selector)
                                    P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
002B 00000000 ffffffff Data RW Ac 3 Bg Pg P  Nl 00000cf3
(fs Selector)
                                    P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0053 7ffda000 00000fff Data RW Ac 3 Bg By P  Nl 000004f3
```

В приложениях MS Windows режима пользователя селекторы `cs`, `ds`, `es`, `ss` и `gs` имеют нулевое базовое значение, поэтому линейный адрес совпадает с виртуальным.

Напротив, регистр `fs` изменчив: его значение меняется от потока к потоку. Сегмент `fs` в процессах режима пользователя указывает на структуру TEB (Thread Environment Block):

```
0:003> dg fs
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0053 ff306000 00000fff Data RW Ac 3 Bg By P  Nl 000004f3

(Переключение на другой поток)
0:003> ~2s
0:002> dg fs
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0053 ff4a5000 00000fff Data RW Ac 3 Bg By P  Nl 000004f3
```

Память

Прежде чем описывать команды для работы с памятью, важно объяснить обозначения адреса и диапазона, поскольку они передаются в качестве аргументов большинству команд, которым требуются адрес памяти и количество элементов.

Параметром `Address` может быть любое значение, выражение или символ, разрешающийся в числовое значение, которое можно интерпретировать как адрес. Число `0x401000` можно рассматривать как адрес, если этот адрес отображен в памяти. Имя `kernel32` разрешится в базовый адрес образа модуля:

```
0:000> lmm kernel32
start    end        module name
75830000 75970000   KERNEL32
0:000> ? kernel32
Evaluate expression: 1971519488 = 75830000
```

Символ вида `module_name!SymbolName` можно использовать как адрес, если он разрешается:

```
0:000> ? kernel32!GetProcAddress
Couldn't resolve error at 'kernel32!GetProcAddress'
0:000> ? kernelbase!GetProcAddress
Evaluate expression: 1979722334 = 76002a5e
```

В качестве адреса можно использовать любое выражение (независимо от того, разрешается ли значение в допустимый адрес):

```
0:000> ? (kernelbase!GetProcAddress - kernel32) / 0n4096
Evaluate expression: 2002 = 000007d2
```

Параметр `Range` можно задать двумя способами. Первый - парой начального и конечного адресов:

```
0:000> db 02c0000 02c0005
002c0000  23 01 00 00 00                                  #.....
```

Второй способ - адрес, за которым следует символ `L` и выражение (`address L Expression_Or_Value`), задающее количество элементов.

Если количество положительно, начальным адресом будет указанный адрес, а конечный адрес подразумевается и равен `address + count`:

```
0:000> db 02c0000 L5
002c0000  23 01 00 00 00                                  #....
```

Если количество отрицательно, указанный адрес становится конечным, а начальный адрес равен `address - count`:

```
0:000> db 02c0005 L-5
002c0000  23 01 00 00 00                                  #....
```

По умолчанию выражение или значение, переданное после `L`, не может превышать 256 МБ. Это предотвращает случайную передачу очень больших значений. Чтобы отменить это ограничение, вместо простого `L` используйте `L?`. Например, обратите внимание, как DbgEng жалуется на слишком большой размер:

```
0:000> db @$ip L0xffffffff
                          ^ Range error in 'db @$ip l0xffffffff
```

При использовании `L?` DbgEng охотно выполнит запрос:

```
0:000> db @$ip L?0xffffffff
760039c2  83 e4 f8 83 ec 18 8b 4d-1c 8b c1 25 b7 7f 00 00  .......M...%....
...
```

Вывод содержимого памяти

Команда `d` применяется для вывода содержимого памяти. Общий синтаксис выглядит так:

```
d[a|b|c|d|D|f|p|q|u|w|W] [Options] [Range]
```

Содержимое памяти можно выводить в различных форматах. Наиболее распространены следующие:

• `b`, `w`, `d`, `q` - соответственно форматы byte, word, double-word и quad-word;

• `f`, `D` - соответственно значения с плавающей запятой одинарной и двойной точности;

• `a`, `u` - соответственно вывод содержимого памяти в ASCII или Unicode;

• `p` - значения указателей (размер зависит от текущего размера указателя целевой системы).

Если к dp, dd или dq добавить суффикс s, будут выведены символы, соответствующие адресам. Это может пригодиться для обнаружения указателей на функции, заданных в массиве или виртуальной таблице:

```
(1)
0:011> bp combase!CoCreateInstance
(2)
0:024> g
Breakpoint 0 hit
combase!CoCreateInstance:
7526aeb0 8bff            mov     edi,edi
0:011> ? poi(esp+4*5)
Evaluate expression: 112323728 = 06b1ec90
0:011> ? poi(poi(esp+4*5))
Evaluate expression: 0 = 00000000
(3)
0:011> g poi(esp)
combase!CustomUnmarshalInterface+0x15d:
752743e7 fe8ef0000000    dec     byte ptr [esi+0F0h]
ds:002b:08664160=01
0:011> ? poi(06b1ec90)
Evaluate expression: 141774136 = 08734d38
(4)
0:011> dps 08734d38 L1
08734d38  752c9688 combase!CErrorObject::`vftable'
0:011> dps 752c9688 L3
752c9688  752f6bdf combase![thunk]:CErrorObject::QueryInterface`adjustor{8}'
752c968c  752f6bd0 combase![thunk]:CErrorObject::AddRef`adjustor{8}'
752c9690  752a9b91 combase![thunk]:CErrorObject::Release`adjustor{8}'
```

Метка 1 добавляет точку останова на следующую функцию:

```
HRESULT CoCreateInstance(
  REFCLSID rclsid,

  LPUNKNOWN pUnkOuter,
  DWORD dwClsContext,
  REFIID riid,
  LPVOID *ppv)
```

Нас интересует определение значения указателя (параметра 5) вновь созданного интерфейса после возврата из функции. На метке 2 мы возобновляем выполнение. Позднее программа достигает точки останова и приостанавливается. Затем мы исследуем пятую позицию указателя и разыменовываем ее. Разыменованное значение должно быть NULL и правильно инициализироваться только в случае успешного возврата функции. На метке 3 мы позволяем отладчику выполнить функцию CoCreateInstance и вернуться к вызывающему коду. Затем снова разыменовываем выходной указатель. Наконец, на метке 4 команда dps выводит адрес vftable, после чего dps применяется еще раз для вывода трех указателей из vftable.

> **Примечание.** На 32-разрядных целевых системах dps эквивалентна dds, а на 64-разрядных - dqs.

Изменение содержимого памяти

Для изменения содержимого памяти используйте команду e. Общий синтаксис выглядит так:

```
e[b|d|D|f|p|q|w] Address [Values]
```

> **Примечание.** Если после команды e не задан суффикс, используется суффикс, который ранее применялся с e. Например, если в первый раз использовалась ed, то при следующем применении одной e она будет действовать как ed.

Чтобы записать по указанному адресу памяти значения byte, word, dword или qword соответственно, используйте спецификаторы формата b, w, d или q:

```
0:000> eb 0x1b0000 11 22 33 44; db 0x1b0000 L 4
001b0000  11 22 33 44
0:000> ed 0x1b0000 0xdeadbeef 0xdeadc0de; dd 0x1b0000 L 2
001b0000  deadbeef deadc0de
```

При использовании форматов w/d или q символьные значения можно вводить в одиночных кавычках. DbgEng учитывает порядок байтов целевой системы:

```
0:000> ed 1b0000 'TAG1'
0:000> db 1b0000 'TAG1' L 4
001b0000  31 47 41 54                                           1GAT
```

Помимо изменения памяти целочисленными значениями, у команды e есть другие спецификаторы формата, позволяющие вводить иные типы:

• `e[f|D] (address values)` - записывает число с плавающей запятой одинарной или двойной точности:

```
0:000> eD @$t0 1999.99
0:000> dD @$t0 L 1
000000c9`e2450000                1999.99
```

• `ep (address values)` - записывает значения размером с указатель. Команда знает размер указателя исходя из отлаживаемой в данный момент целевой системы.

• `e[a|u] (address string)` - записывает по заданному адресу строку ASCII или Unicode. Записанная строка не будет завершена нулем:

```
0:000> f 0x1b0000 L0x40 0x21 0x22 0x23; db 0x1b0000 L0x20;
Filled 0x40 bytes
001b0000  21 22 23 21 22 23 21 22-23 21 22 23 21 22 23 21
!"#!"#!"#!"#!"#!
001b0010  22 23 21 22 23 21 22 23-21 22 23 21 22 23 21 22
"#!"#!"#!"#!"#!"
0:000> ea 0x1b0000 "Hello world"; db 0x1b0000 L0x20
001b0000  48 65 6c 6c 6f 20 77 6f-72 6c 64 23 21 22 23 21  Hello
world#!"#!
001b0010  22 23 21 22 23 21 22 23-21 22 23 21 22 23 21 22
"#!"#!"#!"#!"#!"
```

• `e[za|zu] (address string)` - в отличие от `e[a|u]`, эта команда записывает в конце строки завершающий нулевой символ.

Чтобы заполнить область памяти заданным шаблоном, используйте команду f:

```
f Address L Count Values
```

Например:

```
0:000> f @eax L0x40 0x21 0x22 0x23; db @eax L0x20
Filled 0x40 bytes
001b0000  21 22 23 21 22 23 21 22-23 21 22 23 21 22 23 21  !"#!"#!"#!"#!"#!
001b0010  22 23 21 22 23 21 22 23-21 22 23 21 22 23 21 22  "#!"#!"#!"#!"#!"
```

Прочие команды работы с памятью

Ниже приведен еще один набор полезных команд, связанных с памятью:

• s [-[flags]type] Range Pattern - ищет в памяти заданный шаблон;

• c Range_For_Address1 Address2 - сравнивает две области памяти;

• .dvalloc [Options] Size - выделяет память в пространстве процесса отладчика:

```
0:000> .dvalloc 0x2000
Allocated 2000 bytes starting at 001c0000
```

• .dvfree [Options] BaseAddress Size - освобождает память, ранее выделенную .dvalloc.

• .readmem FileName Range - считывает файл с диска в память отлаживаемой программы:

```
kd> .readmem file.bin @eax L3
Reading 3 bytes.
```

• .writemem FileName Range - записывает память отлаживаемой программы в файл на диске.

Символы

Следующие команды позволяют исследовать символы и структурированные данные:

• dt [type] [address] - очень удобная команда для вывода типа элемента по заданному адресу:

```
$$ Вывести тип структуры UNICODE_STRING
0:000> dt UNICODE_STRING
ole32!UNICODE_STRING
   +0x000 Length           : Uint2B
   +0x002 MaximumLength    : Uint2B
   +0x004 Buffer           : Ptr32 Wchar

$$ Вывести сведения о типе и значения типа по заданному адресу
0:000> dt _UNICODE_STRING 0x18fef4
ntdll!_UNICODE_STRING
 "KERNEL32.DLL"
   +0x000 Length           : 0x18
   +0x002 MaximumLength    : 0x1a
   +0x004 Buffer           : 0x00590168  "KERNEL32.DLL"
```

• dv [flags] [pattern] - выводит сведения о локальных переменных.

• x [options] [module_pattern]![symbol_pattern] - выводит символ или символы в заданном модуле либо модулях.

• !dh [options] Address - выводит заголовки образа PE.

• !drvobj DriverObjectPtr [Flags] - выводит сведения об объекте DRIVER_OBJECT.

• !heap - выводит сведения о куче.

• !pool - выводит сведения о пуле ядра.

Точки останова

На архитектуре x86/x64 DbgEng поддерживает два типа точек останова:

• Программные точки останова - они создаются путем сохранения байта по адресу точки останова с последующей заменой его байтом `0xCC` (на x86/x64). Отладчик реализует внутреннюю логику, необходимую для обработки механизма точки останова.
• Аппаратные точки останова - также называемые процессорными точками останова или точками останова по данным; в зависимости от оборудования, на котором выполняется целевая система, они могут присутствовать или отсутствовать. Их количество ограничено, а срабатывание можно настроить на чтение, запись или выполнение.

Простой синтаксис создания программной точки останова выглядит следующим образом:

```
bp Address ["CommandString"]
bu Address "CommandString"
bm SymbolPattern ["CommandString"]
```

> **Примечание.** Полный синтаксис команд `b*` приведен в документации отладчика.

Чтобы вывести список точек останова, просто используйте команду `bl`:

```
0:001> bl
 0 e 771175c9     0001 (0001)  0:**** ntdll!RtlInitString+0x9
 1 e 77117668     0001 (0001)  0:**** ntdll!RtlInitUnicodeString+0x38
 2 e 771176be     0001 (0001)  0:**** ntdll!_sin_default+0x26
 3 e 7711777e     0001 (0001)  0:**** ntdll!sqrt+0x2a
 4 e 771177c0     0001 (0001)  0:**** ntdll!sqrt+0x6a
```

Для отключения точек останова используйте команду `bd`. Аналогично команда `be` включает точки останова, а `bc` очищает (удаляет) их.

Можно указать последовательность идентификаторов точек останова, которые требуется включить, отключить или очистить:

```
be 0 2 4
```

Либо диапазон:

```
be 1-3
```

Либо просто все точки останова:

```
be *
```

Неразрешенные точки останова

Команда `bu` создает точку останова, адрес которой пока неизвестен/не разрешен либо может измениться, если он принадлежит модулю (то есть учитывает ASLR), многократно загружаемому и выгружаемому по разным базовым адресам.

При загрузке нового модуля отладчик попытается заново вычислить адрес точки останова; если символ совпадет, точка останова станет активной. При выгрузке модуля точка останова становится неактивной до тех пор, пока символ снова не удастся разрешить.

Иными словами, адрес точки останова не фиксирован и автоматически корректируется отладчиком.

Программные точки останова

Программные точки останова создаются командой bp. Если при создании точки останова ее адрес удается разрешить, точка становится активной. Если разрешить точку останова не удается, она ведет себя как неразрешенная и становится активной после разрешения адреса. Если модуль по адресу точки останова выгрузить, а затем загрузить снова, ранее разрешенный адрес точки останова останется фиксированным (в отличие от неразрешенных точек останова).

Аппаратные точки останова

Аппаратные точки останова создаются командой ba. Эти точки останова поддерживаются аппаратурой. Для создания аппаратной точки останова необходимо указать адрес, тип доступа и размер. Тип доступа определяет срабатывание на чтение (чтение/запись), запись (только запись) или выполнение. Размер определяет величину элемента, при доступе к которому требуется остановка. Например, для остановки при "доступе к слову" укажите размер 2.

> **Примечание.** Архитектура ограничивает количество аппаратных точек останова.

Условные точки останова

Условной может быть точка останова любого только что описанного типа. Фактически с каждой точкой останова можно связать команду. Если с точкой останова связана условная команда, эту точку можно считать условной.

В следующем примере создается условная точка останова: когда eax имеет значение 5, она приостанавливает выполнение; в противном случае выполнение продолжает возобновляться:

```
0:000> uf kernelbase!GetLastError
KERNELBASE!GetLastError:
7661d0d6 64a118000000    mov     eax,dword ptr fs:[00000018h]
7661d0dc 8b4034          mov     eax,dword ptr [eax+34h]
7661d0df c3              ret

0:000> bp 7661d0df ".if @eax!=5 { gc; }"
0:000> bl
0 e 7661d0df 0001 (0001) 0:*** KERNELBASE!GetLastError+0x9 ".if @eax!=5
{gc;}"
```

С точкой останова можно связать и более сложное условие. Это рассматривается далее в этой главе, в разделе "Написание сценариев с помощью Debugging Tools".

Исследование процессов и модулей

DbgEng позволяет исследовать работающие процессы, загруженные/выгруженные модули и загруженные драйверы режима ядра.

Чтобы получить список загруженных и выгруженных модулей, используйте `lm`:

```
0:001> lm n
start    end        module name
00400000 00405000   image00400000
5ca40000 5cb44000   MFC42
733a0000 733b9000   dwmapi
73890000 73928000   apphelp
...
```

Аналогично, при отладке в режиме ядра команда `lm` выводит список загруженных драйверов устройств:

```
kd> lm n
start    end        module name
804d7000 806cd280   nt       ntkrnlpa.exe
806ce000 806ee380   hal      halaacpi.dll
b205e000 b2081000   Fastfat  Fastfat.SYS
b2121000 b2161380   HTTP     HTTP.sys
b2d2b000 b2d4cd00   afd      afd.sys
b2d4d000 b2d74c00   netbt    netbt.sys
b2d75000 b2dcca80   tcpip    tcpip.sys
bf800000 bf9c0380   win32k   win32k.sys
f83e6000 f8472480   Ntfs     Ntfs.sys
f86ca000 f86d6c80   VolSnap  VolSnap.sys
f8aaa000 f8aad000   BOOTVID  BOOTVID.dll

...
```

> **Примечание.** Параметр `n` был передан, чтобы сократить стандартный вывод команды `lm`.

Чтобы просмотреть сведения о модуле (версию, размер, базовый адрес и т. д.), используйте параметр `v` для подробного режима и `m`, чтобы задать имя модуля для сопоставления:

```
kd> lm v m *volsnap*
start    end        module name
f86ca000 f86d6c80   VolSnap
    Loaded symbol image file: VolSnap.sys
    Image path: VolSnap.sys
    Image name: VolSnap.sys
    Timestamp:        Tue Aug 03 23:00:14 2004 (41107B6E)

    CheckSum:         00017B61
    ImageSize:        0000CC80
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
```

В режиме ядра доступен полный обзор всех работающих процессов. Чтобы вывести все работающие процессы, используйте команду расширения `!process` с флагами `0 0`:

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 823c8830  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid:
0000
    DirBase: 00334000  ObjectTable: e1000c90  HandleCount: 246.
    Image: System
PROCESS 820ed020  SessionId: none  Cid: 017c    Peb: 7ffdd000  ParentCid:
0004
    DirBase: 07f40020  ObjectTable: e14f9c60  HandleCount:  21.
    Image: smss.exe
PROCESS 81e98740  SessionId: 0  Cid: 0278    Peb: 7ffde000  ParentCid: 017c
    DirBase: 07f40060  ObjectTable: e1010ac8  HandleCount: 517.
    Image: winlogon.exe
PROCESS 81e865c0  SessionId: 0  Cid: 02a4    Peb: 7ffde000  ParentCid: 0278
    DirBase: 07f40080  ObjectTable: e1a7a450  HandleCount: 265.
    Image: services.exe
```

```
PROCESS 821139f0  SessionId: 0  Cid: 0354    Peb: 7ffd9000  ParentCid: 02a4
    DirBase: 07f400e0  ObjectTable: e1a78ce0  HandleCount: 201.
    Image: svchost.exe
PROCESS 81e68558  SessionId: 0  Cid: 0678    Peb: 7ffdd000  ParentCid: 0658
    DirBase: 07f401e0  ObjectTable: e177aa70  HandleCount: 336.
    Image: explorer.exe
```

> **Примечание.** Это эквивалентно использованию команды расширения `!for_each_process` без параметров.

С помощью отладчика ядра можно устанавливать точки останова в процессах режима пользователя. Сначала нужно переключиться в правильный контекст процесса, а для этого требуется значение EPROCESS:

```
kd> !process 0 0 explorer.exe
PROCESS 81e68558  SessionId: 0  Cid: 0678    Peb: 7ffdd000  ParentCid: 0658
    DirBase: 07f401e0  ObjectTable: e177aa70  HandleCount: 336.
    Image: explorer.exe
```

Затем для переключения в контекст нужного процесса используйте команду `.process /r /p` EPROCESS:

```
kd> .process /r /p 81e68558
Implicit process is now 81e68558
.cache forcedecodeuser done
Loading User Symbols.........
```

После переключения контекста команда `lm` выводит не только загруженные драйверы ядра, но и модули режима пользователя.

В следующем примере для данного EPROCESS устанавливается точка останова на `kernel32!CreateFileW`:

```
(1)
kd> bp /p 81e68558 kernel32!CreateFileW
(2)
kd> bl
 0 e 7c810976     0001 (0001) kernel32!CreateFileW
     Match process data 81e68558
(3)
kd> g
Breakpoint 0 hit
kernel32!CreateFileW:
001b:7c810976 8bff            mov     edi,edi
(4)
kd> .printf "%mu\n", poi(@esp+4);
C:\Temp\desktop.ini
```

В метке 1 командой `bp /p` EPROCESS устанавливается фильтр EPROCESS, чтобы точка останова срабатывала только для процесса `explore.exe`. Метка 2 выводит список точек останова. Обратите внимание: совпадение произойдет только для определенного EPROCESS. На метке 3 мы возобновляем выполнение и ожидаем срабатывания точки останова. На метке 4 выводится имя файла, к которому был выполнен доступ. Метка 4 станет гораздо понятнее после того, как далее в этой главе вы прочитаете раздел "Язык".

Теперь предположим, что вы хотите вывести все процессы, вызвавшие API `CreateFileW`, и имена файлов, к которым они обращались:

```
kd> bp kernel32!CreateFileW "!process @$proc 0;.printf "%mu\n",poi(@esp+4);gc;"
```

Точка будет срабатывать всякий раз, когда любой процесс режима пользователя достигнет ее, после чего связанная с точкой останова команда вызовет `!process` с текущим `EPROCESS` (заданным в предопределенном псевдорегистре `$proc`), выведет сведения о контексте текущего процесса и имя файла, а затем возобновит выполнение командой `gc`.

> **Примечание.** `!process @$proc 0` эквивалентна `!process -1 0`.

После возобновления выполнения вы увидите следующий сокращенный вывод:

```
kd> g

PROCESS 82067020  SessionId: 0  Cid: 0138    Peb: 7ffdf000  ParentCid: 02a4
    DirBase: 07f40260  ObjectTable: e1b66ef8  HandleCount: 251.
    Image: vmtoolsd.exe

C:\WINDOWS\SoftwareDistribution\DataStore\DataStore.edb
PROCESS 81dc0da0  SessionId: 0  Cid: 0204    Peb: 7ffd5000  ParentCid: 03fc

    DirBase: 07f40280  ObjectTable: e1ba8ea8  HandleCount: 177.
    Image: wuauclt.exe

PROCESS 81e68558  SessionId: 0  Cid: 0678    Peb: 7ffdd000  ParentCid: 0658
    DirBase: 07f401e0  ObjectTable: e177aa70  HandleCount: 362.
    Image: explorer.exe

C:\WINDOWS\media\Windows XP Start.wav
PROCESS 81e68558  SessionId: 0  Cid: 0678    Peb: 7ffdd000  ParentCid: 0658
    DirBase: 07f401e0  ObjectTable: e177aa70  HandleCount: 351.
    Image: explorer.exe

C:\WINDOWS\WinSxS\Policies\x86_Policy.6.0.Microsoft.Windows.Common-Controls
_6595b64144ccf1df_x-ww_5ddad775\6.0.2600.2180.Policy
PROCESS 820f0020  SessionId: 0  Cid: 0260    Peb: 7ffdf000  ParentCid: 017c
    DirBase: 07f40040  ObjectTable: e1503128  HandleCount: 343.
    Image: csrss.exe
```

Прочие команды

В этом разделе представлены несколько разнородных команд отладчика и команда `.printf` вместе с поддерживаемыми ею спецификаторами формата, а также описано использование языка разметки отладчика (Debugger Markup Language, DML) с `.printf` и другими командами, поддерживающими DML.

Команда .printf

Команда `.printf` - одна из самых полезных команд для вывода сведений из сценариев или команд. Как и в языке C, она принимает спецификаторы формата. Ниже приведены несколько важных:

• `%p (pointer value)` - выводит значение указателя.

• `%d`, `%x`, `%u (number value)` - выводят целочисленные значения. Синтаксис очень похож на спецификаторы формата C.

• `%ma`/`%mu (pointer value)` - выводят строку ASCII/Unicode по указанному указателю.

• %msa/%msu (pointer value) - выводят значение ANSI_STRING/UNICODE_STRING по указанному указателю.

• %y (pointer value) - выводит имя символа (и смещение, если оно есть) по указанному указателю.

Вот простой пример:

```
0:000> .printf "t0=%d t1=%d eax=%x ebx=%d\n", @$t0, @$t1, @eax, @ebx
t0=0 t1=0 eax=5 ebx=8323228
```

Спецификатора %s для раскрытия строковых аргументов нет. В следующем примере значение пользовательского псевдонима раскрывается путем его встраивания в параметр формата:

```
0:000> aS STR "TheValue"
0:000> al
  Alias            Value
 -------          -------
STR              TheValue

0:000> .printf "This value of string is ${STR}\n"
```

Команда .printf может использовать язык разметки отладчика (DML). Чтобы применить DML с .printf, укажите параметр /D.

> **Примечание.** DML работает только в WinDbg.

Чтобы выводить цветные строки, используйте разметку col:

```
0:000> .printf /D "<col fg=\"emphfg\">Hello</col> world\n"
Hello world
```

Также можно применять теги u, i и b соответственно для подчеркивания, курсива и полужирного начертания:

```
0:000> .printf /D "<u>underline</u> <b>bold</b> <i>italics</i>\n";
underline bold italics
```

Очень полезна разметка link, поскольку она делает вывод доступным для щелчка и связывает его с командой:

```
0:000> .printf /D "Click <link cmd=\"u 0x401000\">here</link>\n"
Click here
```

Некоторые команды отладчика также принимают параметр /D. Например, lm /D выводит список модулей, причем каждый модуль доступен для щелчка. При щелчке по модулю выполняется команда lmvm modulename.

> **Примечание.** Команда .prefer_dml 1 переключает глобальную настройку, предписывающую поддерживающим DML командам по возможности предпочитать DML.

Дополнительные сведения приведены в файле dml.doc из дистрибутива средств отладки.

Другие команды

Прежде чем завершить обсуждение команд отладчика, перечислим еще несколько полезных команд:

• # - ищет шаблон дизассемблированного кода.

• `!gle` - возвращает код последней ошибки.

• `.logopen`/`.logfile`/`.logappend`/`.logclose` - команды управления записью вывода окна команд в текстовые файлы.

• `.load` - загружает расширение отладчика.

• `.cls` - очищает окно вывода отладчика. (Эта команда не работает в сценариях, поскольку не является частью языка сценариев DbgEng.)

• `.effmach` - изменяет или выводит режим процессора, используемый отладчиком. Она полезна при отладке процессов WOW64. Эта команда также аналогична команде расширения `!wow64exts.sw`.

Написание сценариев с помощью Debugging Tools

В этом разделе показаны важные возможности написания сценариев в DbgEng, полезные для автоматизации задач обратной разработки и отладки.

Псевдорегистры

DbgEng поддерживает псевдорегистры для хранения определенных значений. Имена всех псевдорегистров начинаются со знака `$`. Префикс `@` перед псевдорегистром или регистром сообщает интерпретатору, что идентификатор не является символом, поэтому исчерпывающий и порой медленный поиск символа выполняться не будет.

Предопределенные псевдорегистры

В этом разделе представлены некоторые полезные предопределенные псевдорегистры. Их можно использовать в выражениях или как параметры команд отладчика и сценариев. Обратите внимание: в зависимости от отлаживаемой цели некоторые псевдорегистры могут быть определены или не определены.

• `$csp` - текущий указатель стека вызовов. Это удобно, поскольку не приходится угадывать, следует использовать `esp` или `rsp`.

• `$ip` - текущий указатель команд. Аналогично текущий указатель команд можно обозначить точкой (`.`).

• `$retreg`/`$retreg64` - регистры возврата (обычно `eax`, `edx:eax` или `rax`).

• $p - первое значение, выведенное последней командой d?:

```
0:000> dd @$ip L 1
012aa5e5  012ec188
0:000> ? @$p
Evaluate expression: 19841416 = 012ec188
0:000> dw @$ip+2 L 1
012aa5e5  c188
0:000> ? @$p
Evaluate expression: 49544 = 0000c188
0:000> db @$ip+2 L 1
012aa5e5  88
0:000> ? @$p
Evaluate expression: 136 = 00000088
```

• $ra - текущий адрес возврата. Эквивалентен poi(@$csp).

• $exentry - адрес точки входа первого исполняемого файла текущего процесса. Это очень полезно при отладке программы с самого начала, поскольку DbgEng останавливается не в точке входа, а в ядре.

• $peb - Process Environment Block. Этот псевдорегистр имеет тип ntdll!_PEB *.

• $proc - адрес EPROCESS* текущего процесса в режиме ядра. В режиме пользователя он эквивалентен $peb.

• $teb - Thread Environment Block текущего потока. Имеет тип ntdll!_TEB*.

• $thread - ETHREAD* в режиме ядра. В режиме пользователя совпадает с $teb.

• $tpid - идентификатор текущего процесса.

• $tid - идентификатор текущего потока.

• $ptrsize - размер указателя с точки зрения отлаживаемой программы. Если основная ОС 64-разрядная, а вы отлаживаете 32-разрядный процесс, то $ptrsize=4. В режиме ядра возвращает размер указателя целевой машины.

• $pagesize - количество байтов на странице памяти (обычно 4096).

• $dbgtime - текущее время (на основе времени компьютера, на котором работает отладчик).

• $bpNUM - адрес, связанный с точкой останова с данным номером:

```
0:000> bl
 0 e 012aa597     0001 (0001)  0:**** calc!WinMainCRTStartup+0xf
 1 e 012aa5ab     0001 (0001)  0:**** calc!WinMainCRTStartup+0x23
0:000> ? @$bp0
Evaluate expression: 19572119 = 012aa597
0:000> ? @$bp1
Evaluate expression: 19572139 = 012aa5ab
```

• $exp - значение последнего вычисленного выражения:

```
0:000> r $t0 = 1 + 4
0:000> ? @$exp
Evaluate expression: 5 = 00000005
```

или:

```
0:000> ? Esp
Evaluate expression: 1637096 = 0018fae8
0:000> ? @$exp
Evaluate expression: 1637096 = 0018fae8
```

В первом примере после вычисления значение присваивается псевдорегистру. Можно увидеть, как `$exp` возвращает последнее значение. То же справедливо для второго примера, в котором вычисляется значение регистра `esp`.

Пользовательские псевдорегистры

Помимо предопределенных псевдорегистров, DbgEng позволяет пользователям определять собственный набор псевдорегистров. DbgEng предоставляет 20 пользовательских псевдорегистров (user-defined pseudo-registers, UDPR) для использования и хранения целочисленных значений. Это регистры от `$t0` до `$t19`. Для присваивания им значений применяется команда `r`:

```
0:000> r $t0 = 1234
0:000> ? @$t0
Evaluate expression: 4660 = 00001234
```

Поскольку числа могут быть указателями, с помощью команды `r?` в этих псевдорегистрах можно хранить типизированные указатели:

```
(1)
0:000> ? poi(@$ip)
Evaluate expression: 409491562 = 1868586a
(2)
0:000> r? $t0 = @@c++((unsigned long *)@$ip)
(3)
0:000> ? @@c++(*@$t0)
Evaluate expression: 409491562 = 1868586a
```

На метке 1 мы разыменовываем и вычисляем значение, на которое указывает `$ip`. На метке 2 командой `r?` присваиваем `$t0` выражение C++; оператор приведения используется для возврата в `$t0` типизированного указателя (типа `unsigned long *`). Наконец, на метке 3 оператор разыменования C++ применяется для разыменования `$t0`. (Без предварительно типизированного `$t0` или без предшествующего выражению приведения это было бы невозможно.)

Вот еще один пример:

```
0:000> r? $t0 = @@c++(@$peb->ProcessParameters->ImagePathName)
0:000> ? $t0

Evaluate expression: 0 = 00000000
0:000> ?? @$t0
struct _UNICODE_STRING
 "c:\windows\syswow64\calc.exe"
   +0x000 Length           : 0x38
   +0x002 MaximumLength    : 0x3a
   +0x004 Buffer           : 0x0098189e  "c:\windows\syswow64\calc.exe"
```

Обратите внимание: при вычислении `$t0` с помощью `?` получается ноль. Однако при использовании синтаксиса вычисления C++ `??` получается фактическое типизированное значение.

В выражениях также можно использовать символы, любые виды псевдорегистров и псевдонимы.

Псевдонимы

Псевдоним - это механизм, позволяющий установить соответствие между значением и символическим именем. При вычислении псевдонима получается присвоенное ему значение.

DbgEng поддерживает три вида псевдонимов:

• Псевдонимы с пользовательскими именами - как следует из названия, их имена выбирает пользователь.

• Псевдонимы с фиксированными именами - их десять, от `$u0` до `$u9`.

• Автоматические псевдонимы - предопределенные псевдонимы, раскрывающиеся в определенные значения.

Псевдонимы с пользовательскими именами

В этом разделе описывается создание пользовательских псевдонимов и управление ими, а также объясняется их интерпретация.

Создание псевдонимов с пользовательскими именами и управление ими

Для создания псевдонимов с пользовательскими именами используются следующие команды:

• `as AliasName Alias_Equivalence` - создает однострочное соответствие для заданного псевдонима:

```
as MyAlias lm;vertarget
```

Эта команда создает псевдоним для двух команд: `lm`, а затем `vertarget`. Обе команды можно выполнить, вызвав `MyAlias`.

• `aS AliasName Alias_Equivalence` - создает фразовое соответствие для заданного псевдонима. Это означает, что точка с запятой завершает соответствие псевдонима (если только соответствие не заключено в кавычки) и начинает новую команду.

```
aS MyAlias lm;vertarget
aS MyAlias "lm;vertarget"
```

Первая строка выполнит два действия: создаст псевдоним со значением `lm`, а затем выполнит команду `vertarget`. Вторая строка (поскольку соответствие заключено в кавычки) определит псевдоним со значением `lm;vertarget`.

> **Примечание.** Имена пользовательских псевдонимов не могут содержать пробел.

К другим командам работы с псевдонимами относятся:

• `al` - выводит уже определенные псевдонимы.

• `ad [/q] AliasName|*` - удаляет псевдоним по имени либо все псевдонимы. Параметр `/q` не выводит сообщения об ошибке, если имя псевдонима не найдено.

Команду aS можно использовать для создания псевдонимов, соответствующих значениям переменных среды, выражениям, содержимому файлов, выводу команд и даже содержимому строк из памяти отлаживаемой программы:

• aS /f AliasName FileName - присваивает псевдониму содержимое файла:

```
0:000> aS /f AliasName c:\temp\lines.txt
0:000> al
  Alias            Value
 -------          -------
 AliasName        line1
line2
line3
line4
line5
```

• aS /x AliasName Expression64 - присваивает псевдониму 64-разрядное значение выражения. Это полезно во многих случаях, особенно при присваивании значения автоматического псевдонима псевдониму с пользовательским именем:

```
0:000> r $t0 = 0x123
0:000> as /x AliasName @$t0
0:000> al
  Alias            Value
 -------          -------
 AliasName        0x123
0:000> as IncorrectAlias @$t0
0:000> al
  Alias            Value
 -------          -------
 AliasName        0x123
 IncorrectAlias   @$t0
```

Обратите внимание: первое применение as /x правильно присвоило псевдониму значение 0x123, тогда как второе присваивание as взяло буквальное значение @$t0 (из-за отсутствующего параметра /x).

• as /e AliasName EnvVarName - задает псевдониму AliasName значение переменной среды с именем EnvVarName:

```
0:000> as /e CmdPath COMSPEC
0:000> al
  Alias            Value
 -------          -------
 CmdPath          C:\Windows\system32\cmd.exe
```

• as /ma AliasName Address - задает псевдониму содержимое строки ASCII с нулевым завершителем, на которую указывает адрес:

```
0:000> db 0x40600C
0040600c  54 6f 6f 6c 62 61 72 57-69 6e 64 6f 77 33 32 00  ToolbarWin-
dow32.
0:000> as /ma Str1 0x40600C
0:000> al
  Alias            Value
 -------          -------
 Str1             ToolbarWindow32
```

• as /mu AliasName Address - задает псевдониму содержимое строки Unicode с нулевым завершителем, на которую указывает адрес.

• `as /ms[a|u] AliasName Address` - задает псевдониму содержимое `ASCII_STRING` (структуры, определенной в DDK) или `UNICODE_STRING`:

```
(1)
0:000> dt _UNICODE_STRING
ntdll!_UNICODuE_STRING
   +0x000 Length           : Uint2B
   +0x002 MaximumLength    : Uint2B
   +0x004 Buffer           : Ptr32 Uint2B

(2)
0:000> ?? sizeof(_UNICODE_STRING)
unsigned int 8

(3)
0:000> ?? @@c++(@$peb->ProcessParameters->DllPath)
struct _UNICODE_STRING
 "C:\Windows\system32\NV"
   +0x000 Length           : 0x2c
   +0x002 MaximumLength    : 0x2e
   +0x004 Buffer           : 0x001f1880  "C:\Windows\system32\NV"

(4)
0:000> dd @@c++(&(@$peb->ProcessParameters->DllPath)) L2
001f1408  002e002c 001f1880

(5)
0:000> db 001f1880 L2e
001f1880  43 00 3a 00 5c 00 57 00-69 00 6e 00 64 00 6f 00
C.:.\.W.i.n.d.o.
001f1890  77 00 73 00 5c 00 73 00-79 00 73 00 74 00 65 00
w.s.\.s.y.s.t.e.
001f18a0  6d 00 33 00 32 00 5c 00-4e 00 56 00 00 00        m.3.2.\.N.V...

(6)
0:000> as /msu DllPath @@c++(&(@$peb->ProcessParameters->DllPath))
0:000> al
  Alias            Value
 -------          -------
 DllPath          C:\Windows\system32\NV
```

На метке 1 выводятся поля структуры `_UNICODE_STRING`, а на метке 2 - размер структуры с помощью вычислителя C++. Аналогично, на метке 3 типизированное вычисление C++ используется для вывода значения поля `DllPath`. На метке 4 оператор & применяется для вывода содержимого поля `_UNICODE_STRING`, а метка 5 выводит адрес `Buffer`. Наконец, на метке 6 команда `as` создает псевдоним, считывая его содержимое через указатель `_UNICODE_STRING`.

Интерпретация псевдонимов с пользовательскими именами

Псевдонимы с пользовательскими именами можно интерпретировать с помощью базового синтаксиса `${AliasName}` или просто введя имя псевдонима. Первую форму следует использовать, когда псевдоним встроен в строку и не окружен пробелами:

```
0:000> aS AliasName "Alias value"
0:000> .printf "The value is >${AliasName}<\n"
The value is >Alias value<
```

Если псевдоним не определен, синтаксис вычисления псевдонима остается невычисленным:

```
0:000> .printf "The value is >${UnkAliasName}<\n"
The value is >${UnkAliasName}<
```

Следующие параметры управляют интерпретацией псевдонимов:

• `${/d:AliasName}` - вычисляется в 1, если псевдоним определен, и в 0, если он не определен. Этот параметр удобен в сценарии для определения того, задан ли псевдоним:

```
0:000> .printf ">${/d:AliasName}<\n"
>1<

0:000> .printf ">${/d:UnkAliasName}<\n"
>0<
```

• `${/f:AliasName}` - при использовании этого параметра неопределенный псевдоним вычисляется в пустую строку, а определенный - в свое фактическое значение:

```
0:000> .printf ">${/f:DefinedAliasName}<\n"
>Alias value<
0:000> .printf ">${/f:UndefinedAliasName}<\n"
><
```

• `${/n:AliasName}` - вычисляется в имя псевдонима или остается невычисленным, если псевдоним не определен:

```
0:000> .printf ">${/n:AliasName}<\n"
>AliasName<
0:000> .printf ">${/n:AliasName2}<\n"
>${/n:AliasName2}<
0:000> .printf ">${/n:UnkAliasName}<\n"
>${/n:UnkAliasName}<
```

• `${/v:AliasName}` - этот параметр предотвращает любое вычисление псевдонима:

```
0:000> .printf ">${/v:AliasName}<\n"
>${/v:AliasName}<
0:000> .printf ">${/v:UnkAliasName}<\n"
>${/v:UnkAliasName}<
```

После определения псевдоним можно использовать в любой последующей команде (как команду или как параметр команды):

```
0:000> aS my_printf .printf
0:000> al
  Alias            Value
 -------          -------
 my_printf        .printf
```

При использовании в качестве команды:

```
0:000> ${my_printf} "Hello world\n"
Hello world
0:000> my_printf "Hello world\n"
Hello world
```

При использовании в качестве параметра команды:

```
0:000> .printf "The command to display strings is >${my_printf}<\n"
The command to display strings is >.printf

0:000> .printf "The command to display strings is my_printf \n"
The command to display strings is  printf
```

При повторном присваивании значений пользовательским псевдонимам учтите следующее:

• Использование команды aS в следующем виде приводит к ошибке:

```
0:000> aS MyVar 0n123;.printf "v=%d", ${MyVar}
v=Couldn't resolve error at '${MyVar}'
```

Причина ошибки в том, что псевдонимы раскрываются только в новых блоках. Исправить это можно следующим образом:

```
0:000> aS MyVar 0n123;.block { .printf "v=%d", ${MyVar}; }
v=123
```

• Параметр `/v:` ведет себя как `/n:` при использовании с aS, as и ad. Причина, по которой мы это упоминаем, показана в следующем примере:

```
0:000> aS MyVar 0n123;.block { aS /x MyVar ${MyVar}+1 }
0:000> al
  Alias            Value
 -------          -------
 0n123            0x7c
 MyVar            0n123
```

Первая команда создает псевдоним MyVar и увеличивает его значение на единицу, однако при этом создается новый псевдоним с именем 0n123. Причина в том, что псевдоним MyVar был заменен своим эквивалентом, а не использован как имя псевдонима.

Вместо этого нужно сообщить команде aS, что MyVar является именем псевдонима и его значение не следует раскрывать или вычислять. Именно здесь нужно использовать параметр `/v:` с командой as или aS:

```
0:000> aS MyVar 123;.block { aS /x ${/v:MyVar} ${MyVar}+1 };al
  Alias            Value
 -------          -------
 MyVar            0x124
```

Обратите внимание: теперь при использовании `${/v:MyVar}` вместе с aS оно вычисляется в имя псевдонима (как это делало бы `${/n:AliasName}`).

Псевдонимы с фиксированными именами

Как упоминалось ранее, существует 10 псевдонимов с фиксированными именами от $u0 до $u9. Хотя такие псевдонимы выглядят как регистры или псевдорегистры, ими не являются. Чтобы присвоить им значения, используйте команду r, после которой укажите `$.` и имя псевдонима:

```
(1)
0:000> r $.u0 = .printf
(2)
0:000> r $.u1 = 0x123
(3)
0:000> r $.u2 = Hello world
(4)
0:000> $u0 "$u2\n"
Hello world
(5)
0:000> $u0 "$u2, u1=%x", $u1
Hello world, u1=123
```

Метка 1 сопоставляет `$u0` команде `.printf`. Обратите внимание на префикс `$.` и на то, что команда `.printf` намеренно не заключена в кавычки в соответствующем значении. Метка 2 определяет `$u1` числовым значением, а метка 3 - `$u2` строковым. На метке 4 `$u0` используется как эквивалент команды `.printf` и выводит `$u2`, заключенный в кавычки и разрешающийся в "Hello world". Наконец, метка 5 выводит значение `$u1` аналогично метке 4.

> **Примечание.** При определении псевдонима всегда используйте `$.`, однако при использовании псевдонима применять `$.` или даже знак `@`, необходимый для псевдорегистров или псевдонимов, не требуется.

Замена псевдонимов с фиксированными именами имеет более высокий приоритет, чем замена псевдонимов с пользовательскими именами.

Автоматические псевдонимы

При запуске сеанса отладки DbgEng определяет несколько псевдонимов. Автоматические псевдонимы похожи на предопределенные псевдорегистры, за исключением того, что их также можно использовать с синтаксисом `${}` (как псевдонимы с пользовательскими именами).

Определены следующие регистры:

• `$ntnsym`

• `$ntwsym`

• `$ntsym`

• `$CurrentDumpFile`
• `$CurrentDumpPath`

• `$CurrentDumpArchiveFile`

• `$CurrentDumpArchivePath`

В качестве иллюстрации следующая команда вызывает консольный отладчик CDB с параметром `-z`, чтобы открыть файл аварийного дампа, и использует `-cf script.wds` для выполнения последовательности команд из текстового файла:

```
c:\Tools\dbg>cdb -cf av.wds -z m:\xp_kmem.dmp
```

Содержимое файла сценария выглядит следующим образом:

```
.printf "Script started\n"
.logopen @"${$CurrentDumpFile}.log"
!analyze -v
.logclose
.printf "Script finished, quitting\n"
q
```

При запуске отладчик интерпретирует каждую строку в `av.wds`:

1. Вывести сообщение о запуске.

2. Открыть файл журнала, имя которого совпадает с именем текущего файла аварийного дампа с добавленным суффиксом `.log`. Обратите внимание, как автоматический псевдоним раскрывается с помощью синтаксиса `${}`.

3. Выполнить команду `!analyze -v`.

4. Закрыть файл журнала, вывести сообщение о завершении и выйти из отладчика командой `q`.

> **Примечание.** Знак @ применяется для определения буквальной (или необработанной) строки. См. следующий раздел "Символы и строки".

Язык

В этом разделе рассматриваются язык сценариев, лексемы и команды.

Комментарии

Для задания комментариев используется команда $$. Например:

```
$$ Это комментарий
$$ Это еще один комментарий
```

Чтобы использовать несколько комментариев в строке с несколькими операторами, завершайте комментарий точкой с запятой:

```
r eax = 0; $$ очистить EAX ; r ebx = ebx + 1; $$ увеличить EBX;
```

Для создания комментариев также можно использовать звездочку (*), однако в этом случае вся строка после звездочки будет проигнорирована, даже если применен разделитель в виде точки с запятой:

```
r eax = 0; * очистить EAX ; r ebx = ebx + 1;
```

Предыдущая команда только очистит EAX; она не увеличит EBX на единицу.

Между спецификатором комментария $$ и командой .echo есть небольшое различие. Команда .echo выводит строку, а не просто игнорирует ее.

Символы и строки

Символы задаются в одиночных кавычках:

```
0:000> @dvalloc 1
0:000> eb @$t0 'a' 'b' 'c' 'd' 'f' 'g'
0:000> db @$t0 L 6
02250000  61 62 63 64 66 67
```

Строки задаются в двойных кавычках:

```
0:000> ea @$t0 "Practical reverse engineering";
0:000> db @$t0 L20
02250000  50 72 61 63 74 69 63 61-6c 20 72 65 76 65 72 73  Practical revers
02250010  65 20 65 6e 67 69 6e 65-65 72 69 6e 67 00 00 00  e engineering...
```

Как и в C, строка может содержать управляющие последовательности; поэтому, чтобы получить правильный результат, последовательность нужно экранировать:

```
(1)
0:000> .printf "c:\\tools\\dbg\\windbg.exe\n"
c:\tools\dbg\windbg.exe
(2)
0:000> .printf "a\tb\tc\n1\t2\t3\n"
a       b       c
1       2       3
```

Первая команда экранирует обратную косую черту с помощью управляющего символа. Во втором примере используется управляющая последовательность горизонтальной табуляции (\t).

DbgEng позволяет использовать необработанные строки; такие строки интерпретируются буквально, без учета управляющих последовательностей. Чтобы задать буквальную строку, поставьте перед ней знак @:

```
(1)
0:000> .printf @"c:\tools\dbg\windbg.exe\n";.printf "\n";
c:\tools\dbg\windbg.exe\n
(2)
0:000> .printf @"a\tb\tc\n1\t2\t3\n"
a\tb\tc\n1\t2\t3\n
```

Обратите внимание: управляющие последовательности остались в заданном виде и не были интерпретированы. Аналогично, если у вас есть псевдоним с пользовательским именем, созданный из содержимого памяти, и вы хотите вычислить его буквально, также добавьте префикс @ перед ${}:

```
(1)
0:000> aS /mu STR 0x3cba030
0:000> al
  Alias            Value
 -------          -------
STR              C:\Temp\file.txt
(2)
0:000> .printf "${STR}\n";
C:Tempfile.txt
(3)
0:000> .printf @"${STR}";.printf "\n";
C:\Temp\file.txt
```

Метка 1 создает псевдоним с пользовательским именем из строки Unicode с нулевым завершителем по указанному адресу памяти и выводит список псевдонимов. Метка 2 выводит значение псевдонима. (Обратите внимание: результат не соответствует ожидаемому.) На метке 3 после добавления к строке префикса @ вывод становится правильным.

Блоки

Блок можно создать командой `.block`, за которой следуют открывающая и закрывающая фигурные скобки ({ }):

```
.block
{
       $$ Внутри блока...
       .block
       {
             $$ Вложенный блок...
       }
}
```

Когда псевдоним с пользовательским именем создается в сценарии, его значение не вычисляется/интерпретируется должным образом, если не создать новый блок:

```
aS MyAlias (@eax + @edx)
.block
{
       $$ Внутри блока...

       .printf "Значение моего псевдонима: %X\n", ${MyAlias}
}
```

Условные операторы

Для написания условных операторов используются командные лексемы `.if`, `.elsif` и `.else`.

Применение `.if` и `.elsif` аналогично другим языкам: они принимают условие. Условием может быть любое выражение, вычисляющееся в ноль (считается ложью) или ненулевое значение (считается истиной):

```
r $t0 = 3;
.if (@$t0==1)
{
	.printf "one\n";
}
.elsif @$t0==2
{
	.printf "two\n";
}
.elsif (@$t0==3)
{
	.printf "three\n";
}
.else
{
	.printf "unknown\n";
}
```

> **Примечание.** Скобки вокруг условия необязательны.

Все встроенные конструкции повторения и условные операторы требуют фигурных скобок ({ и }) и тем самым создают блок, что обеспечивает правильное вычисление псевдонимов:

```
aS MyAlias (@eax + @edx)
.if (1)
{
	$$ Внутри блока...
	.printf "Значение моего псевдонима: %X\n", ${MyAlias}
}
```

Строки также можно сравнивать с помощью `.if` несколькими способами:

```
$$ Заключив сравниваемые строки в одиночные кавычки:
.if '${my_alias}'=='value'
{
	.printf "equal\n";
}

.else
{
	.printf "not equal!\n";
}

$$ С помощью оператора MASM scmp (или sicmp):
.if $scmp("${my_alias}", "value")
{
	.printf "equal\n";
}

$$ С помощью оператора MASM spat:
.if $spat("${my_alias}", "value")
{
	.printf "equal\n";
}
.else
{
	.printf "not equal!\n";
}
```

DbgEng также предоставляет команду `j`, которую можно сравнить с тернарным оператором C (`cond ? true-expr: false-expr`), за исключением того, что она выполняет команды, а не возвращает выражения:

```
j Expression [']Command-True['] ; [']Command-False[']
```

Ниже приведен очень простой пример, в котором в обоих случаях (истина или ложь) выполняется одна команда:

```
0:000> r $t0 = -1
0:000> j (@$t0 < 0) r $t0 = @$t0-1 ; r $t0 = @$t0+1
0:000> ? $t0
Evaluate expression: -2 = fffffffe
```

В большинстве случаев одиночные кавычки необязательны; укажите их, если требуется выполнить несколько команд:

```
0:000> r $t0 = 2
0:000> j (@$t0 < 0) 'r $t0 = @$t0-1;.echo Negative value' ;
                    'r $t0 = @$t0+1;.echo Positive value'
Positive value
0:000> ? $t0
Evaluate expression: 3 = 00000003
```

Команда `j` часто используется как часть команд точки останова для создания условных точек останова.

Следующий пример приостанавливает отладчик (обратите внимание на пустые одиночные кавычки, задающие отсутствие выполняемой команды, когда выражение истинно) только в том случае, когда адрес возврата совпадает с определенным значением:

```
0:000> bp user32!MessageBoxA "j (@$ra=0x401058) '';'gc;'"
0:000> g
user32!MessageBoxA:
756e22c2 8bff            mov     edi,edi
0:000> ? $ra
Evaluate expression: 4198488 = 00401058
```

Следующий пример приостанавливает отладчик всякий раз, когда вызывается функция `GetLastError` и она возвращает `ACCESS_DENIED` (значение 5):

```
0:014> bp kernelbase!GetLastError "g @$ra;j @eax==5 '';'gc'"
0:014> g
uxtheme!ThemePreWndProc+0xd8:
00007ff8`484915e8 33c9            xor     ecx,ecx
0:000> !gle
LastErrorValue: (Win32) 0x5 (5) - Access is denied.
LastStatusValue: (NTSTATUS) 0xc0000034 - Object Name not found.
```

Это не оптимальный способ добиться результата. Загруженные открытые символы NTDLL предоставляют символ `g_dwLastErrorToBreakOn`. Лучший подход - изменить это значение в памяти, передав нужное значение ошибки, при котором следует остановиться:

```
0:000> ep ntdll!g_dwLastErrorToBreakOn 5
0:000> g
(2a0.2228): Break instruction exception - code 80000003 (first chance)
ntdll!RtlSetLastWin32Error+0x21:
00007ff8'4c444df1 cc              int     3
0:000> !gle
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0xc0000034 - Object Name not found.
```

Ошибки сценария

Если во время выполнения сценария отладчика возникает ошибка, после вывода сообщения об ошибке прерывается весь сценарий. Рассмотрим файл сценария со следующим содержимым:

```
.printf "Script started\n";
invalid command;
.printf "Script ending\n";
```

При выполнении этого сценария возникает ошибка:

```
Script started
Script started

   ^ Syntax error in '.printf "Script started
'
0:000>
```

Чтобы сценарий не прерывался, можно использовать командную лексему `.catch`:

```
.printf "Script started\n";
.catch
{
       invalid command;
       .printf "!! will not be reached !!\n";
}
.printf "After catch\n";
```

Из-за ошибки сценарий выйдет из блока `.catch` и выведет сообщение об ошибке, но продолжит выполнение после этого блока:

```
Script started
                ^ Syntax error in ';    invalid command; '
After catch
```

Находясь внутри блока `.catch`, можно явно выйти из него с помощью командной лексемы `.leave`.

Что интересно, `.leave` можно использовать для имитации `break`, как в цикле:

```
r $t0=0;
.catch
{
      .if (by(@$ip) == 0xb9)
      {
             .printf "найдено MOV ECX, ...\n";
             r $t0 = dwo(@$ip+1);
             .leave;
      }
      .elsif (by(@$ip) == 0xb8)
      {
             .printf "найдено MOV EAX, ...\n";
             r $t0 = dwo(@$ip+1);
             .leave;
      }
      $$ выполнить другой анализ...
      .printf "Не удалось найти нужный код операции\n";
      $$ выполнить другие действия...
}
$$ Достигнуто после завершения блока catch, возникновения
$$ ошибки или применения .leave
```

Конструкции повторения

DbgEng поддерживает четыре конструкции повторения, описанные в следующих разделах.

Команду `.break` можно использовать для выхода из цикла. Аналогично команда `.continue` позволяет перейти к следующей итерации внутри охватывающей конструкции повторения.

> **Примечание.** При ошибочном условии повторения (сценарий или команда выполняются бесконечно) выполнение можно прервать, нажав `Ctrl+C` в любом консольном отладчике (KD, CDB, NTSD) или `Ctrl+PauseBreak` в WinDbg.

Цикл for

Командная лексема `.for` имеет следующий синтаксис:

```
.for (InitialCommand ; Condition ; IncrementCommands) { Commands }
```

Следующий пример сценария выводит обработчики таблицы дескрипторов прерываний (IDT) с помощью цикла `for`. Сначала в сеансе отладки режима ядра 32-разрядной системы выполняется команда `dt`, чтобы исследовать структуру `IDTENTRY`:

```
kd> dt _KIDTENTRY
ntdll!_KIDTENTRY
   +0x000 Offset           : Uint2B
   +0x002 Selector         : Uint2B
   +0x004 Access           : Uint2B
   +0x006 ExtendedOffset   : Uint2B
```

Сценарий выглядит следующим образом:

```
.for (r $t0=0; 1;r $t0=@$t0+1)
{
       $$ Получить типизированный указатель на следующую запись IDT
       r? $t1 = @@c++(((_KIDTENTRY *)@idtr) + @$t0);

       $$ Последняя запись?
       .if (@@c++(@$t1->Selector) == 0)
       {
             $$ Выйти из цикла
             .break;
       }

       $$ Разрешить полный адрес
       r $t2  = @@c++((long)(((unsigned long)@$t1->ExtendedOffset << 0x10) +
                             (unsigned long)@$t1->Offset));

       .printf "IDT[%02x] @ %p\n", @$t0, @$t2
       $$ .printf "IDT[%02x] @ %p\n", @$t2
}
```

Некоторые важные аспекты сценария, на которые следует обратить внимание:

• Условие цикла `for` задано равным 1, поэтому цикл выполняется бесконечно. Мы условно выйдем из тела цикла командой `.break`.

• `r?` используется для присваивания `$t1` типизированного значения.

• Псевдорегистр `$t1` является указателем на `_KIDTENTRY`. При прибавлении к нему `$t0` происходит переход к соответствующей позиции памяти (с учетом размера `_KIDTENTRY`).

• Конец записей IDT определяется путем исследования поля `Selector` и соответствующего выхода из цикла.

• Полный базовый адрес обработчика IDT вычисляется объединением полей `ExtendedOffset` и `Offset`.

• `$t2` приводится к типу `long`, чтобы значение правильно расширялось со знаком (поскольку псевдорегистры всегда являются 64-разрядными значениями).

• Выводится результат.

Если использование псевдорегистра вроде `$t0` в качестве счетчика цикла `for` кажется вам немного необычным и вы предпочитаете имя вроде `i`, `j` или `k`, создайте псевдоним с пользовательским именем `i`, эквивалентный `@$t0`:

```
aS i @$t0;

.block
{
        .for (r ${i} = 1; ${i} <= 5; r ${i} = ${i}+1)
        {
                .printf "i=%d\n", ${i}
        }
}
```

Цикл while

Цикл `while` - упрощенная форма цикла `for`, в которой нет ни начальной команды, ни команды приращения:

```
.while (Condition) { Commands }
```

В зависимости от выражения условия тело цикла `while` может не выполниться ни разу. Ниже приведен пример сценария, трассирующего 200 команд во вновь запущенном процессе:

```
$$ Перейти к точке входа (пропустить инициализацию процесса NT)
.printf "Переход к точке входа\n";

g @$exentry;

.printf "Трассировка начата...\n";

$$ Сбросить счетчик
r $t0 = 0;

.while (@$t0 <= 0n200)
{
        .printf "ip -> %p; ntrace=%d\n", @$ip, @$t0;
        r $t0 = @$t0 + 1;
        tr;
}

.printf "Условие выполнено\n";
u @$ip L1;
```

Обратите внимание: это не идеальный способ условной трассировки. Лучше совместно использовать команды `t` и `j`.

Цикл do-while

Цикл `do-while` имеет следующий синтаксис:

```
.do { Commands } (Condition)
```

В отличие от цикла `while`, тело цикла `do` выполняется как минимум один раз до вычисления условия:

```
.do
{
       .if (by(@$ip) == 0xb8)
       {
              .printf "Найдено MOV EAX, ...\n";
              .break;
       }
       $$ выполнить другие действия
       $$ ....
       $$ ....
} (0);
.printf "Продолжить выполнять что-то еще...\n";
```

DbgEng также предоставляет команду `z`, которая выполняет команды, пока определенное условие остается истинным:

```
Command [ Command ; [Command ...;] ]; z( Expression )
```

В следующем примере `$t0` используется как счетчик для трассировки пяти (5) команд ветвления:

```
0:000> r $t0=1
0:000> th;r $t0=@$t0 + 1; z (@$t0 <= 5);
redo [1] th;r $t0=@$t0 + 1; z (@$t0 <= 5);
redo [1] th;r $t0=@$t0 + 1; z (@$t0 <= 5);
redo [1] th;r $t0=@$t0 + 1; z (@$t0 <= 5);
redo [1] th;r $t0=@$t0 + 1; z (@$t0 <= 5);
0:000> ? @$t0
Evaluate expression: 6 = 00000006
```

Как и в предыдущем примере, слева от команды `z` можно указать одну или несколько команд.

Цикл foreach

Цикл `foreach` очень полезен: его можно применять для перебора лексем, прочитанных из файла, вывода команды или строки, предоставленной пользователем.

Командной лексеме `.foreach` можно передать два распространенных параметра (по отдельности или вместе) в качестве первых параметров:

• `/pS ExpressionValue` - начальное количество лексем, которые следует пропустить при запуске цикла. Это эквивалентно инициализации счетчика ненулевым значением в цикле `for`.

• `/ps ExpressionValue` - количество лексем, которые следует пропускать после каждой итерации. Это эквивалентно части приращения цикла `for`, где программист может задать величину приращения счетчика.

Разбиение строки на лексемы

Общий синтаксис выглядит следующим образом:

```
.foreach [Options] /s (TokenVariableName "InString" ) { OutCommands }
```

Например, предположим, что вы ищете символы, связанные с `*CreateFile*`, в следующих трех модулях: NTDLL, KERNELBASE и KERNEL32. Это можно сделать так:

```
.foreach /s (token "ntdll kernel32 kernelbase") { x ${token}!*CreateFile*; }
```

В следующем примере предположим, что требуется разбить на лексемы содержимое заданной строки ASCII в памяти:

```
aS /mu STR 0x8905e8
r $t0 = 0;
.block
{
      .foreach /s (token "${STR}")
      {

            .printf "token_i=%d, token_val=${token}\n", @$t0;
            r $t0 = @$t0 + 1;
      }
}
```

Для вычисления значения переменной лексемы используется `${}`. Это необходимо лишь в том случае, если во время вычисления лексема не окружена пробелами. `.block` использован, чтобы вызвать вычисление псевдонима `STR`.

Разбиение вывода команды на лексемы

Общий синтаксис выглядит так:

```
.foreach [Options] ( Variable  { InCommands } ) { OutCommands }
```

Это наиболее распространенное применение `.foreach`, поскольку оно позволяет извлекать сведения из вывода команды и использовать их в сценарии.

Для демонстрации представим сценарий, которому требуется выделить память в пространстве процесса отлаживаемой программы, а затем использовать эту память для считывания в нее содержимого файла.

Сначала рассмотрим вывод команды выделения памяти `.dvalloc`:

```
0:000> .dvalloc 0n4096
Allocated 1000 bytes starting at 00620000
```

Вывод можно разбить на шесть лексем; следовательно, цикл `foreach` должен применить флаг `/pS`, чтобы пропустить первые пять лексем и сразу начать с последней (то есть со вновь выделенного адреса памяти):

```
0:000> .foreach /pS 5 (token {.dvalloc 0x1000 }) { r $t0 = ${token}; .break; }
0:000> ? @$t0
Evaluate expression: 8323072 = 007f0000
```

Полный сценарий принимает следующий вид:

```
$$ Задать имя файла образа
aS fileName @"c:\temp\shellcode.bin"
.catch
{
       $$ Задать размер выделения равным размеру файла, который нужно прочитать
       r $t0 = 0n880;

       .foreach /pS 5 (token {.dvalloc @$t0; })
       {
             r $t1 = token;
             .break;
       }

       $$ Прочитать файл
       .readmem "${fileName}" @$t1 L@$t0;

       .printf "Загружен ${fileName} @ %p\n", @$t1
}
```

Примечание. Не забудьте освободить память командой `.dvfree`.

В следующем примере разбирается вывод `lm1m` (который по замыслу возвращает упрощенный вывод для использования с `.foreach`):

```
0:000> lm1m
image00400000
SHCORE
KERNEL32
comctl32
user32
Ntdll
```

Цикл `foreach` должен выглядеть так:

```
0:000> .foreach (modulename { lm1m; }) { .printf "Module name: modulename \n";}
```

Разбиение файла на лексемы

Общий синтаксис выглядит следующим образом:

```
.foreach [Options] /f ( Variable  "InFile" ) { OutCommands }
```

Предположим, имеется файл `lines.txt` со следующим содержимым:

```
This is line 1
This is line 2
This is line 3
```

Он будет разбит на лексемы следующим образом:

```
0:000> .foreach /f (line "c:\\temp\\lines.txt") { .printf ">${line}<\n" }
>This<
>is<
>line<
>1<
>This<
>is<
>line<
>2<
>This<
>is<
>line<
>3<
```

Циклы foreach, предоставляемые расширениями

Существуют и другие команды `foreach`, предоставляемые расширениями и не являющиеся частью языка сценариев. Эти команды `foreach` реализованы внутри различных расширений DbgEng:

• `!for_each_frame` - выполняет команду для каждого кадра в стеке текущего потока.

• `!for_each_function` - выполняет команду для каждой функции заданного модуля, совпадающей с шаблоном поиска.

• `!for_each_local` - выполняет команду для каждой локальной переменной в текущем кадре.

• `!for_each_module` - выполняет команду для каждого загруженного модуля.

• `!for_each_process` - выполняет команду для каждого процесса (это расширение работает только при отладке ядра).

• `!for_each_thread` - выполняет команду для каждого потока (только при отладке ядра).

> **Примечание.** Чтобы перечислить все команды расширений `foreach`, используйте команду `.extmatch *for_each*`.

Каждая из этих команд расширения предоставляет выполняемой команде специальные переменные. Чтобы узнать, какие переменные предоставляет каждая конкретная команда расширения, обратитесь к руководству отладчика.

В следующем примере перечисляются все модули и выводятся некоторые сведения о них:

```
!for_each_module .printf /D "%16p %16p: ${@#ModuleName}
@<link cmd=\"u %p\">%p</link>\n", ${@#Base},${@#End},
$iment(0x${@#Base}), $iment(0x${@#Base})
          400000           408000: image00400000 @00406800
        74b70000         74c78000: gdi32 @74b7afc5
        75130000         75270000: KERNEL32 @7514a5cf
        755d0000         75656000: comctl32 @755d1e15
        75670000         757bf000: user32 @75685422
        759b0000         76b28000: shell32 @759b108d
        76c00000         76cbe000: msvcrt @76c0a9ed
        76fa0000         77017000: ADVAPI32 @76fa1005
```

Точка входа была вычислена оператором `$iment()`. Кроме того, язык разметки отладчика (DML) использован, чтобы сделать точку входа доступной для щелчка. При щелчке команды в точке входа будут дизассемблированы.

В следующем примере выполняется поиск всех функций NTDLL, содержащих подстроку `File` в имени:

```
!for_each_function -m:ntdll -p:*File* -c:.echo @#SymbolName
```

> **Примечание.** Чтобы выполнить несколько команд, заключите их в кавычки или просто используйте одну из команд, запускающих файлы сценариев.

Файлы сценариев

Чтобы указать DbgEng выполнить сценарии, можно использовать различные команды. Эти команды делятся на две основные категории:

• Команды, которые открывают файл сценария, заменяют все переводы строк точкой с запятой (разделителем команд) и объединяют все содержимое в один блок команд. Эти команды имеют форму `$><`.

• Команды, которые открывают файл сценария и интерпретируют каждую строку отдельно. Эти команды имеют форму `$<`.

Первая форма очень удобна при использовании команды отладчика, принимающей другие команды как аргументы. Например, команда `bp` принимает действие точки останова, которым может быть простая команда или команда, запускающая файл сценария (содержащий внутри несколько команд).

Вторая форма интерпретирует содержимое файла сценария построчно; каждая строка может содержать несколько команд, разделенных точкой с запятой. Каждая выполненная команда также отражается в выводе отладчика.

Некоторые команды отладчика интерпретируют всю строку, не обращая внимания на наличие точки с запятой (`;`). Это означает, что для таких сценариев команды семейства `$><` не подойдут. Рассмотрим следующий пример сценария:

```
r eax;r ebx
r $.u0 = This is just a line
.printf "$u0"
r ecx;r edx
```

Запуск этого сценария с `$><` работает не так, как задумано:

```
0:000> $><test.wds
eax=00000000
ebx=00000000
0:000> ? $u0

Couldn't resolve error at 'This is just a line;.printf "";r ecx;r edx'
```

И наоборот, запуск именно этого сценария с `$<` работает нормально:

```
0:000> $<test.wds
0:000> r eax;r ebx
eax=00000000
ebx=00000000
0:000> r $.u0 = This is just a line
0:000> .printf "$u0"
This is just a line0:000> r ecx;r edx
ecx=f7fc0000
edx=00000000
```

Причина такого поведения в том, что при присваивании значения псевдониму с фиксированным именем точки с запятой также становятся частью присваивания. Это объясняет, почему в первом выводе сценарий словно внезапно остановился: `$><` объединяет все строки, разделяя их точкой с запятой.

По той же причине, если используется команда, создающая блоки, а фигурные скобки ({ и }) расположены в файле сценария на отдельных строках, $< не будет работать правильно:

```
.if (1 == 2)
{
       .printf "No way!\n";
}
.else
{
       .printf "That's what I thought";
}
```

При выполнении предыдущего сценария возвращается следующая ошибка:

```
(1)
0:000> $<blocktest.wds
0:000> .if (1 == 2)
                  ^ Syntax error in '.if (1 == 2)'
0:000> {
       ^ Syntax error in '{'
0:000>       .printf "No way!";
No way!0:000> }
       ^ Syntax error in '}'
0:000> .else
           ^ Syntax error in '.else'
0:000> {
       ^ Syntax error in '{'
0:000>       .printf "That's what I thought";
That's what I thought0:000> }
       ^ Syntax error in '}'
(2)
0:000> $><p:\book\scripts\t_blocktest.wds
That's what I thought
```

> **Примечание.** Если перед командами запуска сценария стоит дополнительный $, имя/путь файла сценария больше не может содержать точки с запятой. Если после $$>< или $$< найдена точка с запятой, все последующее интерпретируется как другой набор команд.

Чтобы запустить файл сценария, объединив его содержимое в один блок команд, используйте $>< или $$><:

```
$$><path_to\the_script.wds; r eax; al; bl;
```

Поскольку используется $$><, точка с запятой позволяет выполнить последующие команды.

Передача аргументов файлам сценариев

Аргументы можно передавать сценариям с помощью команды $$>a<:

```
$$>a<path_to\the_script.wds arg1 arg2 ...
```

Затем доступ к аргументам в сценарии осуществляется через псевдонимы $argN. Псевдоним $arg0 содержит имя сценария (как argv[0] в C).

Если передать UDPR в качестве аргументов, перед передачей сценарию они не будут раскрыты или вычислены. Это непростая ситуация, которая может привести к различным неожиданным результатам. Например, предположим, что сценарий вызывается так:

```
$$>a<script.wds @$t1 @$t2
```

Предыдущему сценарию в качестве `${$arg1}` и `${$arg2}` будут переданы соответственно значения `@$t1` и `@$t2`. Чтобы решить эту проблему, присвойте псевдорегистры псевдонимам с пользовательскими именами, а затем вызовите сценарий из `.block`. Это гарантирует раскрытие значений псевдонимов до передачи сценарию:

```
aS /x val1 @$t0
aS /x val2 @$t1
.block
{
    $$>a<script.wds ${val1} ${val2}
}
ad /q val1
ad /q val2
```

Чтобы проверить наличие аргумента, используйте `.if` с `${/d:...}`:

```
.catch
{
       .if ${/d:$arg1} == 0 or ${/d:$arg2} == 0
       {
              .printf "Использование: ${$arg0} адрес-памяти длина\n";
              .leave;
       }
       r $t0 = ${$arg1};

       .if $vvalid($t0, 1) == 0
       {
              .printf "Указан недопустимый адрес памяти\n";
              .leave;
       }
       r $t1 = @$t0 + ${$arg2} - 1;
       .printf "Суммирование байтов памяти от %x до %x\n", @$t0, @$t1;
       .for (r $t3 = 0;@$t0 <= @$t1;r $t0 = @$t1 +1)
       {
              r $t3 = @$t3 + by(@$t1);
       }
       .printf "Результат: %x\n", @$t3;
}
```

Мы применили несколько приемов, заслуживающих краткого объяснения:

• `.catch` и `.leave` использовались для имитации начала функции и поведения, подобного `return`.

• `.if` и `${/d:$arg1}` использовались для проверки того, определен ли первый аргумент. Поскольку синтаксис вычислителя явно не переключался, движок сценариев выполняет вычисление с помощью MASM; следовательно, все используемые операторы должны быть допустимы в синтаксисе MASM. Если заключить выражение в `@@c++(expression)`, оно будет вычислено в синтаксисе C++.

• Оператор `$vvalid()` используется для проверки допустимости переданного адреса памяти.

• Командная лексема `.for` используется для перебора содержимого памяти, а каждый байт по соответствующему адресу разыменовывается оператором MASM `by()`.

В следующем выводе сценарию передаются различные аргументы:

```
(1)
0:000> $$>a<script.wds
Usage: script.wds memory-address len
(2)
0:000> $$>a<script.wds 0xbadf00d
Invalid memory address specified
(3)
0:000> $$>a<script.wds @eip 2
Summing memory bytes from 76f83bc5 to 76f83bc6
The result is eb
```

На метке 1 сценарий выполняется без аргументов и успешно показывает способ использования. На метке 2 сценарию передается недопустимый адрес памяти. Наконец, на метке 3 сценарий вызывается правильно и возвращается сумма байтов.

> **Примечание.** Расширение файла `.wds` необязательно. Это лишь соглашение, используемое различными авторами сценариев; оно расшифровывается как файл сценария WinDbg.

Использование сценариев как функций

В языке сценариев DbgEng невозможно определять функции. Однако различные файлы сценариев можно использовать так, словно они являются функциями. Сценарий может рекурсивно вызвать сам себя или вызвать другой сценарий с другим набором аргументов, причем в контексте каждого сценария эти аргументы будут различаться.

UDPR очень удобны при написании сценария. Когда сценарий вызывает другой сценарий, эти UDPR будут общими для всех сценариев, поэтому их нельзя использовать исключительно внутри каждого сценария, не нарушая состояние других вызывающих сценариев, если только сценарий не сохраняет и не восстанавливает их в точках входа и выхода.

> **Примечание.** Необходимость сохранять UDPR можно представить по аналогии с регистрами в программах x86 или AMD64, где компилятор создает код, сохраняющий определенные регистры общего назначения при входе в каждую функцию и выходе из нее, а также (в зависимости от соглашения о вызовах) выделяет определенные регистры для входных/выходных данных функции.

С учетом этого важно разработать механизм, позволяющий легко, незаметно и с минимальным повторением сохранять/восстанавливать определенные UDPR всякий раз, когда сценарий собирается вызвать другой.

Псевдоним файла сценария @call

В предыдущем разделе была обозначена необходимость в способе сохранения/восстановления UDRP. Поэтому мы разработали два простых сценария, выполняющих именно это. В этом разделе показаны оба сценария, `init.wds` и `call.wds`, и объясняется их работа.

Сценарий `init.wds` служит для настройки среды сценариев и создания коротких псевдонимов, действующих как имена функций:

```
(1)
ad /q *;
(2)
aS ${/v:SCRIPT_PATH} @"p:\book\scripts";
.block
{
        $$ Вызываемые сценарии (с помощью @call) (3)

        aS ${/v:#sigma}      @"${SCRIPT_PATH}\sigma";
        aS ${/v:#pi}         @"${SCRIPT_PATH}\pi";
        $$ Псевдонимы вызова сценариев (4)
        aS ${/v:@dvalloc}    @"$$>a<${SCRIPT_PATH}\dvalloc.wds";
        aS ${/v:@call}       @"$$>a<${SCRIPT_PATH}\call.wds";
}

r $t19 = 0; (5)
r $t18 = 1; (6)
```

Сценарий `init.wds` устанавливает два соглашения об именовании псевдонимов с пользовательскими именами:

• Имена с префиксом @ обозначают псевдонимы команды `$$>a<` (запускающей сценарий с аргументами). Обычно это самодостаточные сценарии. (Им не требуется сохранять UDPR и необязательно вызывать себя или другие сценарии.)

• Имена с префиксом # обозначают псевдоним, который можно вызвать с помощью псевдонима `@call`. Такие сценарии могут быть рекурсивными и вправе безопасно предполагать, что все UDPR, кроме назначенных для возвращаемых значений, будут сохранены/восстановлены до/после вызова/возврата сценария.

Метка 1 удаляет все ранее определенные псевдонимы. На метке 2 задается базовый путь сценариев. (Обратите внимание на использование @ для задания буквальной строки.) На метке 3 определяются два псевдонима с пользовательскими именами, снабженных префиксом #. Их можно вызвать через псевдоним `@call`, и они вычисляются в полный путь сценария без расширения `.wds`. (Сценарий вызова добавит расширение.) Для демонстрации определены два вызываемых сценария: `sigma` и `pi`. На метке 4 определяются два псевдонима с пользовательскими именами, снабженных префиксом @. Эти псевдонимы просто разрешаются в `$$>a<`, за которым следует полный путь сценария. Именно псевдоним `@call` делает возможным вызов сценариев как функций. `@dvalloc` является оболочкой команды `.dvalloc`. Метка 5 определяет UDPR `$t19`, который внутри сценария `call.wds` используется для запоминания уровня вложенности вызовов сценариев. Уровень вложенности применяется для формирования псевдонима, сохраняющего все UDPR для каждого уровня вложенности. На метке 6 определяется UDPR `$t18`, который внутри `call.wds` используется для определения числа UDPR начиная с `$t0`, пропускаемых при восстановлении сохраненных UDPR после вызова сценария (подробнее об этом в следующем объяснении).

Вот сценарий `call.wds`:

```
ad /q ${/v:_tn_} (1)
.catch
{
        .if ${/d:$arg1} == 0 (2)
        {
                .printf "Не указан вызываемый сценарий";
                .leave;

        }
        $$ Вычислить имя псевдонима сохраненных регистров предыдущего вызова
        aS /x ${/v:_tn_} @$t19; (3)
        .block
        {
```

```
                $$ Удалить имя псевдонима сохраненных регистров предыдущего запуска
                ad /q _sr_${_tn_};
        }
        r $t19 = @$t19 + 1; $$ Увеличить уровень вложенности (4)

        $$ Вычислить имя псевдонима сохраненных регистров текущего запуска
        aS /x ${/v:_tn_} @$t19; (5)

        $$ Сохранить все псевдорегистры
        .block
        { (6)
                aS /c _sr_${_tn_} "r $t0,$t1,$t2,$t3,$t4,$t5,$t6,$t7,$t8,$t9,
$t10,$t11,$t12,$t13,$t14,$t15,$t16,$t17";
        }

        $$ Вызвать сценарий
        .catch
        {(7)
                $$>a<"${$arg1}.wds" ${/f:$arg2} ${/f:$arg3} ${/f:$arg4}
${/f:$arg5} ${/f:$arg6} ${/f:$arg7} ${/f:$arg8} ${/f:$arg9} ${/f:$arg10}
${/f:$arg11} ${/f:$arg12} ${/f:$arg13} ${/f:$arg14} ${/f:$arg15}
${/f:$arg16} ${/f:$arg17} ${/f:$arg18} ${/f:$arg19} ${/f:$arg20};
        }

        $$ Восстановить регистры после вызова
        .block
        {
                (8)
                $$ Вычислить имя псевдонима сохраненных регистров
                aS /x ${/v:_tn_} @$t19;
                .block
                {
                        $$ Восстановить все регистры, кроме первых,
                        $$ предназначенных для возврата значения
                        .foreach /pS @$t18 /s (X "_sr_${_tn_}" ) (9)
                        {
                                r ${X}; (10)
                        }
                }

                $$ Удалить имя псевдонима сохраненных регистров
                ad /q _sr_${_tn_}; (11)

                $$ Уменьшить уровень вложенности
                r $t19 = @$t19 - 1; (12)
        }
}
ad /q ${/v:_tn_}; (13)
```

Этому сценарию нужны два UDPR специального назначения. Первый - `$t19`, в котором хранится уровень вложенности вызовов. Он увеличивается при каждом использовании `@call` для запуска сценария и уменьшается после завершения выполнения сценария. Поскольку `$t19` увеличивается и уменьшается, для каждого уровня вложенности можно создать псевдоним с уникальным именем, хранящий значения UDPR.

Второй UDPR - `$t18`, обозначающий количество UDPR, используемых для возврата значений (начиная с `$t0`). По умолчанию значение 1 указывает, что единственным регистром для возвращаемого значения является `$t0`. Если сценарий возвращает несколько значений (например, в `$t0` и `$t1`), вызывающий код должен перед вызовом сценария присвоить `$t18` значение 2. Это гарантирует, что ни `$t0`, ни `$t1` не будут возвращены к первоначальным значениям (к значениям до вызова сценария). Сценарий вызова принимает имя вызываемого сценария как первый аргумент, за которым следуют остальные аргументы (от `$arg2` до `$argN`).

Теперь кратко объясним работу остальной части сценария, прежде чем применить его на практике. На метке 1 пользовательский псевдоним `_tn_` (применяемый для вычисления имени псевдонима для каждого уровня вложенности) удаляется перед повторным определением. Метка 2 проверяет,

был ли сценарию передан параметр. На метках 3 и 4 псевдониму _tn_ присваивается числовое значение уровня вложенности (обратите внимание на применение aS /x), после чего UDPR уровня вложенности $t19 увеличивается. На метке 5 создается временный псевдоним со значением текущего уровня вложенности.

На метке 6 UDPR с $t0 по $t17 сохраняются в псевдониме с именем _sr_{_tn_} с помощью aS /c, за которой следует команда r и список UDPR для возврата их значений. Например, если уровень вложенности равен 2, псевдоним сохраненных регистров будет называться _sr_2 и содержать значения всех рассматриваемых UDPR. _sr_0x2 будет эквивалентен $t0=00000003 $t1=00000000 $t2=00000000 ... $t17=00000000.

На метках 7 и 8 сценарий, переданный в $arg1, вызывается с остальными переданными аргументами. После возврата сценария псевдоним _tn_ вычисляется повторно. (Он мог быть перезаписан вызванным сценарием.)

На метках 9 и 10 выполняется перебор текущего псевдонима _sr_NESTING_LEVEL с пропуском $t18 лексем (обратите внимание на параметр /pS), после чего каждый UDPR восстанавливается командой r.

На метках 11-13 очищается псевдоним сохраненных регистров (_sr_NESTING_LEVEL), уменьшается уровень вложенности и удаляется псевдоним временного имени.

Следующий шаг - запустить сценарий init.wds, который создаст соответствующие псевдонимы:

```
0:000> ad /q *; $$><p:\book\scripts\init.wds; al;
  Alias            Value
 -------          -------
 #pi              "p:\book\scripts\pi"
 #sigma           "p:\book\scripts\sigma"
 #test            "p:\book\scripts\test"
 @call            $$>a<"p:\book\scripts\call.wds"
 @dvalloc         $$>a<"p:\book\scripts\dvalloc.wds"
 SCRIPT_PATH      p:\book\scripts
```

Другой способ сделать это - запустить WinDbg (или CDB) с параметром командной строки -c:

```
c:\dbg\windbg.exe -c "ad /q *;$$><p:\book\scripts\init.wds;al;" p:\test.exe
```

Теперь видно, что имеются два псевдонима, @call и @dvalloc, представляющие сценарии, которым не требуется автоматическое сохранение/восстановление UDPR, и еще три сценария, полагающиеся на @call для автоматического сохранения/восстановления UDPR и действующие подобно "функциям".

Сценарий sigma.wds принимает два числовых параметра и возвращает сумму членов между первым и вторым аргументами, помещая результат в $t0:

```
.for (r $t0=0, $t1=${$arg1}, $t2=${$arg2}; @$t1 <= @$t2; r $t1 = @$t1 + 1)
{
       r $t0 = @$t0 + @$t1;
}
```

Чтобы выполнить sigma.wds, используйте @call #sigma start_num end_num, как показано ниже:

```
0:000> @call #sigma 1 4;.printf "The result is %d\n", @$t0
The result is 10
```

Аналогично сценарий `pi.wds` возвращает результат умножения членов между первым и вторым аргументами:

```
.for (r $t0=1, $t1=${$arg1}, $t2=${$arg2}; @$t1 <= @$t2; r $t1 = @$t1 + 1)
{
	r $t0 = @$t0 * @$t1;
}
```

Сценарий `dvalloc.wds` является оболочкой команды `.dvalloc`. При вызове `@dvalloc` результат возвращается в `$t0`, чтобы его можно было использовать в сценариях:

```
.catch
{
	r $t0 = -1; $$ Задать недопустимый результат
	.if ${/d:$arg1} == 0
	{
		.printf "Использование: dvalloc.wds размер-памяти\n";
		.printf "Выделенная память возвращается в t0\n";
		.leave;
	}

	$$ Выделить память и поместить результат в $t0
	.foreach /pS 5 (t {.dvalloc ${$arg1}})
	{
		.if $vvalid(${t}, 1) == 1
		{
			r $t0 = ${t};
			.leave;

		}
	}
}
```

После выделения памяти с помощью `.dvalloc` мы разбиваем результат на лексемы и извлекаем адрес памяти в `$t0`.

И `sigma.wds`, и `pi.wds` являются примерами функций, использующих UDPR `$t1` и `$t2`. Это означает, что, если сценарий вызывает `sigma` или `pi`, после возврата обеих функций к вызывающему коду `$t1` и `$t2` не должны каким-либо образом измениться.

Проверить это поведение можно следующим простым сценарием `test.wds`:

```
r $t1 = 0x123;
r $t2 = 0x456;

.printf "До вызова sigma: t1=%x, t2=%x\n", @$t1, @$t2
@call #sigma 1 3
.printf "После вызова sigma: результат t0=%x, t1=%x, t2=%x\n",
	 @$t0, @$t1, @$t2
```

Предыдущий сценарий присваивает значения UDPR `$t1` и `$t2`, а затем вызывает `#sigma 1 3`, которая изменит `$t1` и `$t2`. Если `@call` работает как ожидается, эти UDPR восстанавливаются сразу после вызова:

```
0:000> @call #test
До вызова sigma: t1=123, t2=456
После вызова sigma: результат t0=6, t1=123, t2=456
```

Примеры сценариев отладки

В этом разделе вы примените различные полезные сценарии, используя на практике все изученное до сих пор.

Получение базового адреса образа заданного модуля

Один из быстрых способов получить базовый адрес образа модуля - использовать команду `lm` (вывести модули) с параметром `m`, чтобы вывести модули, совпадающие с заданным шаблоном:

```
0:000> lm m kernel32
start    end        module name
749e0000 74b20000   KERNEL32   (deferred)
```

Из вывода видно, что пятая лексема содержит базовый адрес образа. Следовательно, базовый адрес образа легко разобрать с помощью `.foreach`: пропустить первые лексемы, извлечь значение пятой лексемы и выйти из цикла:

```
r $t0 = -1;
.foreach /pS 4 ( imgbase { lmm ${$arg1}; } )
{

       r @$t0 = ${imgbase};
       .break;
}
```

Написание базового распаковщика UPX

Написать распаковщик UPX довольно просто, и сделать это можно многими способами. Используемый здесь метод намеренно усложнен, чтобы попрактиковаться с различными командами отладчика. Для правильного понимания сценария предполагается знание основных принципов формата PE-файлов.

Идея сценария такова:

1. UPX упаковывает программу и перемещает исходную точку входа (original entry point, OEP) за пределы секции `.text`, являющейся первой секцией.

2. Сценарий вычисляет границы первой секции и начинает трассировку.

3. Если указатель команд (`EIP`) находится за пределами образа программы, сценарий выполняет `gu`, чтобы вернуться к вызывающему коду.

4. Трассировка продолжается, пока `EIP` не окажется внутри первой секции. В этот момент предполагается, что программа распакована.

Вот сценарий:

```
$$ Получить базовый адрес образа
$$ Получить базовый адрес образа
aS /x IMG_BASE @@c++(@$peb->ImageBaseAddress); (1)
$$ Объявить несколько пользовательских псевдонимов, соответствующих UDPR
aS SEC_START   @$t19; (2)
aS SEC_END   @$t18;
aS IMG_START   @$t17;
aS IMG_END   @$t16;

$$ Перейти к точке входа программы
g @$exentry

.catch
```

```
{
   $$ Получить указатель на заголовки NT
(3)
r $t0 = ${IMG_BASE} + @@c++(((_IMAGE_DOS_HEADER *)${IMG_BASE})->e_lfanew)

   $$ Теперь получить размер необязательного заголовка из
   $$ IMAGE_NT_HEADERS.FileHeader
(4)
r $t1 = @@c++( ((!_IMAGE_NT_HEADERS*)@$t0)->FileHeader.SizeOfOptionalHeader )

   $$ Вычислить адрес первой секции:
   $$ пропустить сигнатуру, размер заголовков файла и необязательных заголовков
r $t2 = @$t0 + 4 + @@c++(sizeof(ole32!_IMAGE_FILE_HEADER)) + @$t1; (5)

$$ (6) Получить границы первой секции
r ${SEC_START} = IMG_BASE +
              @@c++(((_IMAGE_SECTION_HEADER *)@$t2)->VirtualAddress);
r ${SEC_END} = IMG_BASE +
              @@c++(((_IMAGE_SECTION_HEADER *)@$t2)->Misc.VirtualSize);

$$ Вычислить границы образа (7)
r ${IMG_START} = IMG_BASE;
r ${IMG_END} = IMG_START +
              @@c++(((_IMAGE_NT_HEADERS *)@$t0)->OptionalHeader.SizeOfImage);

$$ Логика такова:
$$ 1. Трассировать
$$ 2. Если IP находится за пределами образа, выполнить "gu"
.for (r $t0=0; 1; r $t0 = @$t0 + 1) (8)
{
 $$ Выполнить еще один шаг трассировки, чтобы увидеть, куда он ведет (9)
 t;

 $$ IP за пределами границ образа?
 .if (@$ip < ${IMG_START}) or (@$ip > ${IMG_END}) (10)
 {
    gu;
    .continue;
 }

 $$ IP внутри первой секции?
 .if (@$ip >= ${SEC_START}) and (@$ip <= ${SEC_END}) (11)
 {
    .printf "--- Достигнута первая секция ---\n";
    u;
    .break;
 }
}
}
```

На метке 1 базовый адрес образа выполняющейся в данный момент программы берется из типизированного псевдорегистра `$peb` путем обращения к его полю `ImageBaseAddress` при помощи вычислителя C++, после чего сохраняется в псевдониме `IMG_BASE`.

На метке 2 создается несколько псевдонимов с пользовательскими именами, соответствующих некоторым UDPR. Это удобный прием, позволяющий дать имена таким UDPR. На метке 3 адрес `_IMAGE_NT_HEADERS` присваивается UDPR `$t0` путем сложения базового адреса образа со значением поля `IMAGE_DOS_HEADER.e_lfanew`.

На метке 4 размер необязательных заголовков извлекается в UDPR `$t1`. Это пригодится, чтобы пропустить все заголовки PE и попасть в заголовок первой секции образа.

На метке 5 адрес первой секции образа вычисляется в UDPR `$t2`. На метке 6 из `IMAGE_SECTION_HEADER` разбираются как виртуальный адрес секции (начало секции), так и конец секции (начало секции плюс ее размер).

На метке 7 вычисляются начальный и конечный адреса программы. Начальный адрес - базовый адрес образа, а конечный - базовый адрес образа плюс содержимое поля `IMAGE_OPTIONAL_HEADER.SizeOfImage`.

На метке 8 запускается бесконечный цикл `for`, использующий `$t0` как счетчик и значение 1 как условие.

На метках 9-11 команда `t` используется для трассировки одной команды. Не трассируйте, если `EIP` находится вне границ образа, и остановите трассировку, если `EIP` находится в границах первой секции.

Хотя этот метод слишком длинен, он показывает, как написать более сложный сценарий и логику трассировки на случай более изощренного процесса распаковки.

Ниже приведена более простая версия распаковщика, которая ищет шаблон кода, выполняемый непосредственно перед переходом программы к исходной точке входа (OEP):

```
$$ Распаковка UPX по шаблону
$$ UPX1:0107D7F5 39 C4                cmp     esp, eax
$$ UPX1:0107D7F7 75 FA                jnz     short loc_107D7F3
$$ UPX1:0107D7F9 83 EC 80             sub     esp, -80h
$$ UPX1:0107D7FC E9 ?? ?? ??          jmp     near ptr word_103FC62

$$ Перейти к точке входа программы (не к исходной, а к упакованной)
$$ только если аргументы не указаны
.if ${/d:$arg1} == 0
{
       g @$exentry;
}

$$ Шаблон не найден!
r $t0 = 0; (1)
.foreach (addr { s -[1]b @$ip L200 39 c4 75 fa 83 EC}) (2)
{
       $$ Шаблон найден!
       r $t0 = 1;
       r $t1 = ${addr} + 7; (3)
       .printf /D "Переход JMP к OEP @<link cmd=\"u %x\">%x</link>\n",@$t1,@$t1;
      (4)
       ga @$t1;
      (5)
       t; u;
       .break;
}

.if $t0 == 0
{
   .printf "Не удалось найти шаблон перехода к OEP. Программа упакована
UPX?\n";
}
```

На метке 1 UDPR `$t0` используется как логическая переменная, указывающая, найден ли шаблон.

На метке 2 выполняется поиск шаблона начиная с точки входа и не более чем в 200 байтах с использованием флага 1 команды поиска `s`. Благодаря этому возвращается только адрес совпадения. Если совпадение не найдено, возвращается пустая строка, поэтому `.foreach` нечего разбивать на лексемы.

На метке 3 пропускаются семь байтов после позиции совпавшего шаблона, чтобы указать на длинный относительный переход (возвращающийся к OEP). Этот адрес сохраняется в `$t1`.

На метках 4 и 5 программа выполняется до достижения команды `JMP OEP` (использована команда `ga`, поэтому применяется аппаратная, а не программная точка останова), после чего выполняется один шаг трассировки через команду `JMP OEP`, и таким образом достигается первая команда распакованной программы.

Написание базового монитора файлов

В этом примере создается сценарий, показывающий совместное использование сценариев и условных точек останова для отслеживания всех вызовов ASCII- и Unicode-версий различных функций API файлового ввода-вывода: `CreateFile`, `DeleteFile`, `GetFileAttributes`, `CopyFile` и так далее.

Сценарий рассчитан на однократный вызов с параметром `init` для инициализации, а затем на многократные вызовы в качестве команды точек останова, создаваемых при инициализации.

При вызове сценария из точки останова передаются следующие параметры:

• `ApiName` - используется только для вывода.

• `IsUnicode` - передайте ноль, чтобы указать ASCII-версию API, и единицу, чтобы указать Unicode-версию.

• `FileNamePointerIndex` - номер параметра в стеке, содержащего указатель на буфер имени файла.

• `ApiID` - выбранный вами идентификатор; этот параметр необязателен. Он полезен, если при срабатывании точки останова требуется добавить дополнительную логику. В этом сценарии `CreateFile[A|W]` присваивается идентификатор 5. Позднее проверяется срабатывание этого API, затем имя файла, к которому выполнен доступ, после чего предпринимается соответствующее действие.

Вот содержимое сценария `bp_displayfn.wds`:

```
.catch
{
       .if '${$arg1}' == 'init' (1)
       {

             (2)
              bp kernelbase!CreateFileA @"$$>a<${$arg0} CreateFileA 0 1 5";
              bp kernelbase!CreateFileW @"$$>a<${$arg0} CreateFileW 1 1 5";

             (3)
              bp kernelbase!DeleteFileA @"$$>a<${$arg0} DeleteFileW 0 1";
              bp kernelbase!DeleteFileW @"$$>a<${$arg0} DeleteFileW 1 1";
              bp kernelbase!FindFirstFileA @"$$>a<${$arg0} FindFirstFileA 0 1";
              bp kernelbase!FindFirstFileW @"$$>a<${$arg0} FindFirstFileW 1 1";
              bp kernel32!MoveFileA @"$$>a<${$arg0} MoveFileA 0 1";
              bp kernel32!MoveFileW @"$$>a<${$arg0} MoveFileW 1 1";
              bp kernelbase!GetFileAttributesA
                     @"$$>a<${$arg0} GetFileAttributesA 0 1";
              bp kernelbase!GetFileAttributesExA
                     @"$$>a<${$arg0} GetFileAttributesExA 0 1";
              bp kernelbase!GetFileAttributesExW
                     @"$$>a<${$arg0} GetFileAttributesExW 1 1";
              bp kernel32!CopyFileA @"$$>a<${$arg0} CopyFileA 0 1";
              bp kernel32!CopyFileW @"$$>a<${$arg0} CopyFileW 1 1";

              $$ Игнорировать некоторые события отладки (чтобы уменьшить засорение вывода)
              sxi ld;

              $$ Вывести список вновь установленных точек останова
              bl;
             (4)
              .leave;
       }
      (5)
       $$ Вывести имя API
       .printf "${$arg1}: >";
      (6)
```

```
        $$ Получить указатель на имя файла
        r $t0 = poi(@$csp + 4 * ${$arg3});
       (7)
        $$ Это указатель на строку Unicode?
        .if ${$arg2} == 1
        {
              (8)
               .printf "%mu<\n", @$t0;
        }
        .else
        {
               $$ Вывести как ASCII SZ (9)
               .printf "%ma<\n", @$t0;
        }

        $$ Параметр ApiID задан? (10)
        .if ${/d:$arg4} == 1
        {
               $$ Это идентификатор API CreateFile? (11)
               .if ${$arg4} == 5

        {
               $$ Получить имя файла, чтобы сравнить его
               aS /mu ${/v:FILE_NAME} @$t0; (12)
               .block
               {
                    (13)
                      .if $sicmp(@"${FILE_NAME}", @"c:\temp\eb.txt") == 0
                      {
                             .leave; (14)
                      }
               }
               ad /q ${/v:FILE_NAME};
        }
}

$$ Продолжить после точки останова
gc; (15)
}
```

На метке 1 проверяется, вызван ли сценарий с параметром `init`; если да, сценарий инициализируется (метки 2-4) и завершается. На метке 2 создаются две точки останова для `CreateFileA/W`, условием назначается сам сценарий и передается `ApiID = 5`.

На метках 3 и 4 добавляются точки останова для остальных API без передачи аргумента `ApiID`, после чего сценарий завершается. На метках 5 и 6 выводится имя API, затем `$t0` присваивается указатель на имя файла (с использованием переданного индекса параметра). На метках 7-9 проверяется, вызван ли сценарий для ASCII- или Unicode-версии API, после чего соответствующим образом применяется спецификатор формата `%mu` или `%ma`. На метках 10-12 проверяется, был ли передан `ApiID` и является ли он `ApiID` функции `CreateFile`.

На метках 12-14 имя файла извлекается в псевдоним `FILE_NAME`, создается блок, чтобы псевдоним правильно раскрылся, после чего псевдоним `FILE_NAME` сравнивается с нужным путем к файлу. (Обратите внимание на использование `@` для указания буквального раскрытия строки.) Если путь совпадает с искомым, сценарий завершается и приостанавливает выполнение. Наконец, на метке 15 сценарий возобновляет выполнение после достижения любой из определенных точек останова.

Чтобы использовать этот сценарий, сначала запустите его с параметром `init`:

```
0:000> $$>a<P:\book\scripts\bp_displayfn.wds init; g;
```

Написание базового дескремблера строк

Этот сценарий реализует простую процедуру дескремблирования. Представим, что процедура скремблирования на C выглядит следующим образом:

```
void descramble(unsigned char *p, size_t sz)
{
  for (size_t i=0;i<sz;i++, ++p)

  {
    *p = *p ^ (235 + (i & 1));
  }
}
```

> **Примечание.** Процедура дескремблирования может быть более сложной. Если она использует таблицы и тому подобное, помните, что эти таблицы вам доступны, поскольку сценарий имеет полный доступ к памяти отлаживаемой программы.

Ниже та же процедура реализована на языке сценариев DbgEng. Обратите внимание, как для простой имитации исходного алгоритма используется вычислитель @@c++:

```
.catch
{
       $$ Получить источник
       r $t0 = ${$arg1};

       $$ Получить назначение
       r $t1 = ${$arg2};

       $$ Получить размер
       r $t2 = ${$arg3};

       .for (r $t3=0; @$t3<@$t2; r $t3 = @$t3 + 1, $t0 = @$t0+1, $t1=@$t1+1)
       {
              r $t4 = @@c++((*(unsigned char *)@$t0) ^ (235 + (@$t3 & 1)));
              eb @$t1 @$t4;
       }
       $$ Вывести дескремблированный результат
       db ${$arg2} L ${$arg3};
}
```

Скремблированное содержимое памяти выглядит так:

```
0:000> db 0x4180a4 L 30
004180a4  bb 9e 8a 8f 9f 85 88 8d-87 cc 99 89 9d 89 99 9f  ................
004180b4  8e cc 8e 82 8c 85 85 89-8e 9e 82 82 8c ec eb ec  ................
004180c4  eb ec eb ec eb ec eb ec-eb ec eb ec eb ec eb ec  ................
```

Для дескремблирования запустите сценарий:

```
0:000> @dvalloc 1; ? $t0
Evaluate expression: 131072 = 00020000
0:000> $$>a<descramble.wds 0x4180a4 0x20000 30
00020000  50 72 61 63 74 69 63 61-6c 20 72 65 76 65 72 73  Practical revers
00020010  65 20 65 6e 67 69 6e 65-65 72 69 6e 67 00 00 00  e engineering...
00020020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Использование SDK

До сих пор мы рассматривали автоматизацию задач при помощи средств написания сценариев, предоставляемых инструментами отладки. SDK, поставляемый со средствами отладки, предлагает еще один способ автоматизировать или расширить отладчик. В его состав входят заголовочные файлы, библиотечные файлы для компоновки расширения и различные примеры, показывающие программное использование DbgEng.

SDK находится в подкаталоге `sdk` каталога установки средств отладки. Он имеет следующую структуру каталогов:

• Help - содержит справочные материалы по библиотеке DbgHelp.

• Inc - содержит включаемые файлы, необходимые при использовании SDK.

• Lib - содержит соответствующие библиотечные файлы, используемые на этапе компоновки сборки. В нем имеются библиотеки для WOA (Windows on ARM), AMD64 и i386.

• Samples - содержит примеры, написанные с использованием различных каркасов, которые можно применять для создания расширений отладчика. Есть также примеры использования DbgEng вместо написания для него расширения.

Хотя полное рассмотрение SDK выходит за рамки этой главы, в следующих разделах кратко обсуждается его применение для создания расширений DbgEng для отладчика. Рассмотренного материала должно быть достаточно для хорошего старта, чтобы вы могли легко понять примеры расширений и начать учиться и писать собственные.

Для начала следует знать, что SDK предоставляет три каркаса, с помощью которых можно создавать расширения:

• Каркас расширений WdbgExts - первоначальные расширения WinDbg. Для взаимодействия с DbgEng им необходимо экспортировать несколько обратных вызовов, чтобы работать с API расширений WinDbg, а не с интерфейсом клиента отладчика. Позднее программист может по требованию получить интерфейс клиента отладчика или другие интерфейсы, если нужна дополнительная функциональность.

• Каркас расширений DbgEng - эти более новые типы расширений могут предоставлять автору расширения дополнительную функциональность. Команды расширения имеют доступ к экземпляру интерфейса клиента отладчика, позволяющему получать другие интерфейсы и глубже взаимодействовать с DbgEng.

• Расширения EngExtCpp - построенные поверх каркаса расширений DbgEng, эти расширения создаются путем наследования от базового класса `ExtExtension`. Класс `ExtExtension` предоставляет множество вспомогательных функций, позволяющих расширению выполнять сложные задачи.

В следующих разделах кратко показано создание расширений с помощью каркаса расширений WdbgExts. Обратите внимание: создание расширений с использованием любого из двух других каркасов довольно прямолинейно и может быть освоено по примерам SDK, поставляемым с пакетом Debugging Tools.

Концепции

В этом разделе описаны два способа доступа к API DbgEng:

• Через интерфейсы отладчика, которые можно получить с помощью экземпляра объекта клиента отладки.

• Через структуру, передаваемую обратному вызову инициализации расширения WbgExts. Структура содержит набор указателей на функции API, которыми может пользоваться расширение.

Интерфейсы отладчика

DbgEng предоставляет программисту семь базовых интерфейсов. Со временем добавлялась новая функциональность, и для сохранения обратной совместимости появлялись новые версии этих интерфейсов. Например, на момент написания `IDebugControl` является первой версией интерфейса, а `IDebugControl4` - последней.

Ниже приведен список интерфейсов, краткое объяснение их назначения и некоторые предоставляемые ими функции:

• `IDebugClient5` - этот интерфейс предоставляет различные полезные функции для запуска или остановки сеанса отладки и задания необходимых обратных вызовов DbgEng (ввод/вывод/события). Кроме того, его метод `QueryInterface` используется для получения остальных интерфейсов.

• `CreateProcess`/`AttachProcess` - создает новый процесс или присоединяется к существующему.

• `AttachKernel` - присоединяется к живому отладчику ядра.

• `GetExitCode` - возвращает код завершения процесса.

• `OpenDumpFile` - начинает сеанс отладки из файла дампа.

• `SetInputCallbacks`/`SetEventCallbacks` - задает обратные вызовы ввода/вывода.

• `IDebugControl4` - этот интерфейс предоставляет функции, связанные с управлением процессом:

• `AddBreakpoint` - добавляет точку останова.

• `Execute` - выполняет команду отладчика.
• `SetInterrupt` - подает DbgEng сигнал передать управление от целевой системы.

• `WaitForEvent` - ожидает возникновения события отладчика. Аналогична API Win32 `WaitForDebugEvent()`.

• `SetExecutionStatus` - задает состояние DbgEng. Позволяет программисту возобновить выполнение, запросить шаг с заходом или обходом и т. д.

• `IDebugDataSpaces4` - этот интерфейс предоставляет функциональность для работы с памятью и данными:

• `ReadVirtual` - читает память из виртуальной памяти целевой системы.

• `QueryVirtual` - аналогичная Win32-функции `VirtualQuery()`, эта функция запрашивает виртуальную память в виртуальном адресном пространстве целевой системы.

• `ReadMsr` - читает значение регистра, специфичного для модели.

• `WritePhysical` - записывает физическую память.

• IDebugRegisters2 - предоставляет исследование регистров (перечисление, запрос сведений) и функции установки/получения. DbgEng назначает регистрам индексы. Чтобы работать с регистром по имени, сначала нужно определить его индекс:

• GetDescription - возвращает описание регистра (размер, имя, тип и т. д.).

• SetValue/GetValue - задает/получает значение регистра.

• GetIndexByName - находит индекс регистра по его имени.

• IDebugSymbols3 - предоставляет функциональность для работы с отладочными символами, сведениями о строках исходного кода, запросами типов и т. д.:

• GetImagePath - возвращает путь к исполняемому образу.

• GetFieldName - возвращает имя поля внутри структуры.

• IDebugSystemObjects4 - предоставляет функциональность для запроса сведений из отлаживаемой цели (или целей) и системы, в которой она работает:

• GetCurrentProcessId - возвращает идентификатор DbgEng отлаживаемого в данный момент процесса.

• GetCurrentProcessHandle - возвращает системный дескриптор текущего процесса.

• SetCurrentThreadId - переключает текущий поток по его идентификатору DbgEng. Эквивалентна команде ~Nk.

• IDebugAdvanced4 - предоставляет дополнительную функциональность, необязательно присутствующую в других интерфейсах:

• GetThreadContext/SetThreadContext - получает/задает контекст потока.

• GetSystemObjectInformation - возвращает сведения о нужном системном объекте.

Чтобы использовать API через интерфейсы, необходим экземпляр интерфейса IDebugClient (клиента отладчика) или одного из производных интерфейсов. В следующем фрагменте кода экземпляр интерфейса IDebugClient5 передается вспомогательной функции CreateInterfaces. Затем она многократно вызывает QueryInterface, чтобы получить необходимые интерфейсы:

```
bool CreateInterfaces(IDebugClient5 *Client)
{
  // Интерфейсы уже созданы?
  if (Control != NULL)
    return true;

  // Получить интерфейс клиента отладки
  if (Client == NULL)
  {
    m_LastHr=m_pDebugCreate(__uuidof(IDebugClient5),(void**)&Client);
    if (m_LastHr != S_OK)
      return false;
  }

  // Запросить некоторые другие интерфейсы, которые нам понадобятся.
  do
  {
    m_LastHr = Client->QueryInterface(
      __uuidof(IDebugControl4),
      (void**)&Control);
    if (m_LastHr != S_OK)
      break;
    m_LastHr = Client->QueryInterface(
      __uuidof(IDebugSymbols3),
      (void**)&Symbols);
    if (m_LastHr != S_OK)
      break;
    m_LastHr = Client->QueryInterface(
```

```
      __uuidof(IDebugRegisters2),
      (void**)&Registers);
    if (m_LastHr != S_OK)
      break;
    m_LastHr = Client->QueryInterface(
      __uuidof(IDebugSystemObjects4),
      (void**)&SystemObjects);
    if (m_LastHr != S_OK)
      break;
    m_LastHr = Client->QueryInterface(
      __uuidof(IDebugAdvanced3),
      (void**)&Advanced);
    if (m_LastHr != S_OK)
      break;
    m_LastHr = Client->QueryInterface(
      __uuidof(IDebugDataSpaces4),
      (void**)&DataSpace);
  } while ( false);

  return SUCCEEDED(m_LastHr);
}
```

Переменные интерфейсов определяются так:

```
IDebugDataSpaces4    *DataSpace;
IDebugRegisters2     *Registers;
IDebugSymbols3       *Symbols;
IDebugControl4       *Control;
IDebugSystemObjects4 *SystemObjects;
IDebugAdvanced3      *Advanced;
```

Чтобы получить интерфейс клиента отладчика (`IDebugClient`), используйте функцию `DebugCreate` или функцию `DebugConnect` (подключение к удаленному узлу). Следующий пример получает интерфейс клиента отладчика с помощью `DebugCreate`:

```
HRESULT Status;
IDebugClient *Client;
if ((Status = DebugCreate(__uuidof(IDebugClient),
                          (void**)&Client)) != S_OK)
{
    printf("DebugCreate failed, 0x%X\n", Status);
    return -1;
}
// Теперь все готово для запроса других интерфейсов...
```

API расширений WinDbg

Расширения отладчика получают указатель на структуру `WINDBG_EXTENSION_APIS` через процедуру обратного вызова инициализации расширения `WinDbgExtensionDllInit`. Структура содержит следующие указатели API:

```
// wdbgexts.h
typedef struct _WINDBG_EXTENSION_APIS {
    ULONG                                nSize;
    PWINDBG_OUTPUT_ROUTINE               lpOutputRoutine;
    PWINDBG_GET_EXPRESSION               lpGetExpressionRoutine;
    PWINDBG_GET_SYMBOL                   lpGetSymbolRoutine;
    PWINDBG_DISASM                       lpDisasmRoutine;
    PWINDBG_CHECK_CONTROL_C              lpCheckControlCRoutine;
    PWINDBG_READ_PROCESS_MEMORY_ROUTINE  lpReadProcessMemoryRoutine;
    PWINDBG_WRITE_PROCESS_MEMORY_ROUTINE lpWriteProcessMemoryRoutine;
    PWINDBG_GET_THREAD_CONTEXT_ROUTINE   lpGetThreadContextRoutine;
    PWINDBG_SET_THREAD_CONTEXT_ROUTINE   lpSetThreadContextRoutine;
    PWINDBG_IOCTL_ROUTINE                lpIoctlRoutine;
    PWINDBG_STACKTRACE_ROUTINE           lpStackTraceRoutine;
} WINDBG_EXTENSION_APIS, *PWINDBG_EXTENSION_APIS;
```

Получив эту структуру, расширение должно скопировать и сохранить ее в глобальной переменной, желательно с именем `ExtensionApis`. Такое имя выбирается потому, что заголовочный файл `wdbgexts.h` определяет несколько макросов, обращающихся к `ExtensionApis` для доступа к указателям API:

```
extern WINDBG_EXTENSION_APIS ExtensionApis;

#define dprintf          (ExtensionApis.lpOutputRoutine)
#define GetExpression    (ExtensionApis.lpGetExpressionRoutine)
#define CheckControlC    (ExtensionApis.lpCheckControlCRoutine)
#define GetContext       (ExtensionApis.lpGetThreadContextRoutine)
...
#define ReadMemory       (ExtensionApis.lpReadProcessMemoryRoutine)
#define WriteMemory      (ExtensionApis.lpWriteProcessMemoryRoutine)
#define StackTrace       (ExtensionApis.lpStackTraceRoutine)
```

Эти макросы позволяют авторам расширений напрямую вызывать, например, `StackTrace` или `WriteMemory`, вместо использования `pExtension.lpStackTraceRoutine` или `pExtension.WriteMemory`.

Помимо функций, объявленных в структуре `WINDBG_EXTENSION_APIS`, можно использовать целый набор других функций, основанных на функции `ExtensionApis.lpIoctlRoutine`. Например, `ReadPhysical()` - встроенная функция, которая вызывает `IoCtl()` с управляющим кодом `IG_READ_PHYSICAL`, передавая ему соответствующие параметры.

Список функций, которые можно использовать внутри расширений WdbgExts, приведен в файле справки DbgEng.

Создание расширений Debugging Tools

В предыдущем разделе вы познакомились с концепциями SDK; теперь можно подробнее рассмотреть, как выглядит расширение WdbgExts и как написать самое простое расширение.

Расширение отладчика - это обычная DLL Microsoft Windows. DLL должна экспортировать две обязательные функции, необходимые DbgEng, а затем столько функций, сколько команд расширение предоставляет отладчику.

Первая экспортируемая функция - `WinDbgExtensionDllInit`. Она вызывается, когда отладчик загружает расширение:

```
VOID WinDbgExtensionDllInit(
    PWINDBG_EXTENSION_APIS lpExtensionApis,
    USHORT MajorVersion,
    USHORT MinorVersion)
{
    ExtensionApis = *lpExtensionApis; // Получить копию

    // При желании также сохранить сведения о версии
    SavedMajorVersion = MajorVersion;

    SavedMinorVersion = MinorVersion;

    return;
}
```

Обратите внимание: сохраняется содержимое переданного указателя `lpExtensionApis`. Переданные переменные сведений о версии обозначают соответственно тип сборки Microsoft Windows и номер сборки. При желании сохраните эти переменные, если позднее захотите проверять их значения в командах расширения.

Вторая экспортируемая функция - `ExtensionApiVersion`. DbgEng вызывает ее, когда хочет запросить сведения о версии расширения:

```
EXT_API_VERSION ApiVersion =
{
  5, // Старшая версия
  1, // Младшая версия
  EXT_API_VERSION_NUMBER64,  // Ревизия
  0 // Зарезервировано
};

LPEXT_API_VERSION ExtensionApiVersion(VOID)
{
  return &ApiVersion;
}
```

После определения обязательных функций (или обратных вызовов) можно переходить к объявлению команд расширения.

Команда расширения имеет следующее объявление:

```
CPPMOD VOID myextension(

  HANDLE                hCurrentProcess,
  HANDLE                hCurrentThread,
  ULONG                 dwCurrentPc,
  ULONG                 dwProcessor,
  PCSTR                 args)
```

Наиболее примечательны следующие передаваемые аргументы:

• `dwProcessor` - индекс текущего процессора.

• `dwCurrentPc` - текущий указатель команд.

• `args` - переданные аргументы (если есть).

Еще один предпочтительный способ объявить функцию расширения - использовать макрос `DECLARE_API(api_s)`:

```
DECLARE_API( test )
{
  dprintf("This is a test extension routine");
}
```

> **Примечание.** В любой момент любая команда расширения может вызвать `DebugCreate()`, а затем получить любой нужный интерфейс для приобретения дополнительной функциональности.

Последний шаг - экспортировать две обязательные функции и команды расширения, которые вы собираетесь предоставить DbgEng. Обычный способ - создать файл `.def` и вызвать компоновщик с дополнительным параметром `/DEF:filename.def`. Вот как выглядит DEF-файл написанного нами тестового расширения:

```
EXPORTS

  ; Обратные вызовы, предоставляемые отладчику
  WinDbgExtensionDllInit
  ExtensionApiVersion

  ; Обратные вызовы команд
  test
```

Поместите полученную DLL в каталог средств отладки (либо в его подкаталог `winext`) или в системный каталог Windows. Команда `!load extname` загружает скомпилированное расширение, а `!extension_command` или `!extname.ext_command` выполняет команду расширения.

Полезные расширения, инструменты и ресурсы

Ниже приведен краткий список полезных расширений, инструментов и ресурсов, способных улучшить работу с отладчиком:

• narly (https://code.google.com/p/narly/) - удобное расширение, перечисляющее обработчики `/SAFESEH`, выводящее сведения о `/GS` и DEP, выполняющее поиск ROP-гаджетов и предоставляющее другие разнородные команды.

• SOS - это расширение, поставляемое с Windows Driver Kit (WDK), облегчает отладку управляемого кода.

• `!analyze` - очень полезное расширение (поставляется с DbgEng), выводящее сведения о текущем исключении или аварийной остановке.

• VirtualKd (http://virtualkd.sysprogs.org/) - инструмент, повышающий скорость отладки ядра при использовании с VMWare или VirtualBox.

• windbg.info - этот веб-сайт предоставляет очень подробный справочник по командам WinDbg/DbgEng и форум для обсуждения пользователей.

• kdext.com - этот веб-сайт предоставляет пару расширений DbgEng. Среди них примечательно расширение для подсветки синтаксиса ассемблера и улучшения пользовательского интерфейса.
• SysecLabs WinDbg Scripts (www.laboskopia.com/download/SysecLabs-Windbg-Script.zip) - набор сценариев, помогающих исследовать ядро. Особенно полезен при поиске руткитов.

• `!exploitable` (http://msecdbg.codeplex.com/) - расширение, выполняющее автоматизированный анализ аварий и оценку риска безопасности.

• Qb-Sync (https://github.com/quarkslab/qb-sync) - удобное расширение WinDbg от Quarkslab, позволяющее синхронизировать дизассемблерное представление или граф IDA Pro с WinDbg.

• Pykd (http://pykd.codeplex.com/) - расширение Python для доступа к DbgEng.

Глава 5. Обфускация

Обратная разработка кода, сгенерированного компилятором, - сложный и трудоемкий процесс. Ситуация становится еще хуже, когда код укреплен, то есть намеренно построен так, чтобы противостоять анализу. Такие методы укрепления программ мы объединяем под общим названием обфускации. Ниже приведены некоторые примеры ситуаций, в которых может применяться обфускация:

• Вредоносное ПО - стремление избежать внимания как механизмов обнаружения антивирусов, так и специалистов по обратной разработке является главным мотивом преступников, применяющих вредоносное ПО в своих операциях, поэтому на протяжении многих лет это было традиционной областью применения обфускации.

• Защита интеллектуальной собственности - многие коммерческие программы имеют ту или иную защиту от несанкционированного копирования. Некоторые системы применяют дополнительную обфускацию, чтобы скрыть детали реализации определенных частей системы. Хорошими примерами являются Skype, IMessage компании Apple и даже клиент Dropbox, защищающие форматы своих протоколов связи с помощью обфускации и криптографии.
• Управление цифровыми правами - схемы DRM обычно защищают определенные критически важные сведения (например, криптографические ключи и протоколы) с помощью обфускации. FairPlay компании Apple, Media Foundation Platform компании Microsoft и ее DRM PlayReader - если назвать лишь два примера - являются примерами применения обфускации. В настоящее время это ведущая современная область применения обфускации.

Если говорить абстрактно, "обфускацию" можно рассматривать в терминах преобразований программ. Цель таких методов - принять на вход программу и создать на выходе новую программу, имеющую тот же вычислительный эффект, что и исходная (формально это свойство называется семантической эквивалентностью или вычислительной эквивалентностью), но при этом "более трудную" для анализа.

Понятие "трудности анализа" долгое время определялось неформально, без какой-либо математической строгости. Например, широко распространено мнение, что - в том, что касается человека-аналитика, - размер программы является показателем трудности ее анализа. Программу, затрачивающую 20 000 команд на выполнение одной операции, можно счесть "более трудной" для анализа, чем программу, выполняющую ту же операцию одной командой. Такие предположения сомнительны и привлекли внимание теоретиков (например, Милы Далла Преда[17] и Барака с соавторами[2]).

Для представления обфускатора и, двойственным образом, деобфускатора было предложено несколько моделей. Эти модели полезны для совершенствования проектирования средств обфускации и рассуждений об их устойчивости с помощью адаптированных критериев. Среди них особый интерес представляют две модели.

Первая модель подходит для анализа криптографических механизмов в контексте так называемой атаки белого ящика. В этой модели атакующий определяется как вероятностный алгоритм, пытающийся вывести значимое свойство защищенной программы. Точнее, он пытается извлечь сведения помимо тех, которые можно тривиально вывести из анализа входных и выходных данных программы. Эти сведения значимы в том смысле, что позволяют атакующему обойти функцию безопасности или сами представляют критические данные защищенной программы. Двойственным образом обфускатор определяется в этой модели как вероятностный генератор виртуального черного ящика; идеальный обфускатор гарантирует, что анализ защищенной программы не дает больше сведений, чем анализ распределений ее входных и выходных данных.

Другой способ формализации атакующего - определить действие обратной разработки как абстрактную интерпретацию конкретной семантики защищенной программы. Такое определение естественным образом подходит для статического анализа потока данных программы, являющегося первым этапом перед применением оптимизирующих преобразований. Двойственным образом в модели абстрактной интерпретации обфускатор определяется как специализированный компилятор, параметризованный некоторыми семантическими свойствами, которые не сохраняются.

Цель этих попыток моделирования - получить объективные критерии фактической устойчивости преобразований обфускации. Действительно, многие проблемы, которые когда-то считались трудными, можно эффективно атаковать посредством продуманного применения методов анализа кода. Многие методы, возникшие в контексте более традиционных тем теории языков программирования (например, компиляторов и формальной верификации), можно перепрофилировать для преодоления обфускации.

Эта глава начинается с обзора существующих методов обфускации, часто встречающихся в реальных ситуациях. Затем рассматриваются различные доступные методы и инструменты, разработанные для анализа и возможного взлома обфусцированного кода. Наконец, приводится пример сложной современной схемы обфускации и подробно описывается ее обход с помощью передовых методов анализа.

Обзор методов обфускации

Для простоты изложения начнем с разделения обфускаций на две категории: обфускация на основе данных и обфускация на основе управления. Позднее вы увидите, что они объединяются сложными и трудными способами и фактически неразделимы. Однако прежде чем углубляться в эти направления, мы начнем с характерного примера кода, который можно встретить в реальной обфускации. Обратите внимание: пример особенно прост, поскольку в нем присутствует только обфускация на основе данных, но не на основе управления.

Природа обфускации: мотивирующий пример

Ориентируясь на процессор x86, компиляторы обычно генерируют команды из определенного крошечного подмножества доступного набора команд, а структура управления сгенерированной программы следует предсказуемым соглашениям. Со временем специалист по обратной разработке вырабатывает стиль анализа, приспособленный к этим шаблонам структурированного кода. При столкновении с несоответствующим им кодом скорость анализа может катастрофически снизиться.

Это явление легко показать на конкретном примере. Одна из целей оптимизатора компилятора - уменьшить объем вычислительных ресурсов, затрачиваемых на выполнение задачи, и 50 лет исследований наделили оптимизаторы впечатляющими возможностями в этом направлении, поэтому в переводе исходного кода на язык ассемблера редко встречается очевидная неэффективность. Например, если исходный код предписывает увеличить некоторую переменную на пять (например, оператором `x += 5;`), компилятор, скорее всего, сгенерирует ассемблерный код, похожий на один из следующих вариантов:

```
01: add eax, 5
02: add dword ptr [ebp-10h], 5
03: lea ebx, [ecx+5]
```

В обфусцированном коде вместо этого можно встретить следующий код, если предположить, что `EAX` соответствует переменной x, а значение `EBX` можно свободно перезаписать (или "затереть"):

```
01: xor ebx, eax
02: xor eax, ebx
03: xor ebx, eax
04: inc eax
05: neg ebx
06: add ebx, 0A6098326h
07: cmp eax, esp
08: mov eax, 59F67CD5h
09: xor eax, 0FFFFFFFFh
10: sub ebx, eax
11: rcl eax, cl
12: push 0F9CBE47Ah
13: add dword ptr [esp], 6341B86h
14: sbb eax, ebp
15: sub dword [esp], ebx
16: pushf
17: pushad
18: pop eax
19: add esp, 20h
20: test ebx, eax
21: pop eax
```

В этом примере можно увидеть множество действующих методов обфускации:

• В строках 1-3 для обмена содержимым двух мест - в данном случае регистров `EAX` и `EBX` - используется "трюк обмена XOR".

• Строка 4 показывает присваивание регистру `EAX`, которое в действительности является "мусором" (поскольку в строке 8 `EAX` перезаписывается константой).

• В строках 5-6 регистр `EBX` меняет знак, после чего к нему прибавляется константа `0A6098326h`: `EBX = -EAX + 0A6098326h`.

• В строке 7 `EAX` сравнивается с `ESP`. Команда `CMP` изменяет только флаги, а на последующих строках флаги перезаписываются до следующего использования, поэтому этот код является мусором.

• В строках 8-9 константа `59F67CD5h` помещается в регистр `EAX` и подвергается операции XOR с `-1h` (которое в двоичном виде состоит из одних единичных битов). XOR со всеми единицами эквивалентен операции `NOT`; следовательно, результат этой последовательности - помещение константы `0A609832Ah` в `EAX`.

• Строка 10 вычитает константу в `EAX` из `EBX`: `EBX = -EAX + 0A6098326h - 0A609832Ah`, или `EBX = -EAX - 5`, или `EBX = -(EAX + 5)`.

• Строка 11 изменяет `EAX` с помощью команды `RCL`. Эта команда является мусором, поскольку `EAX` перезаписывается в строке 18.

• Строки 12-13 помещают в стек константу `0F9CBE47Ah`, а затем прибавляют к ней константу `6341B86h`, в результате чего в вершине стека оказывается значение `0h`.

• Строка 14 изменяет EAX с помощью команды SBB, задействуя посторонний регистр EBP. Эта команда является мусором, поскольку EAX перезаписывается в строке 18.

• Строка 15 вычитает EBX из значения, находящегося в данный момент на вершине стека (то есть 0h). Следовательно, dword ptr [ESP] = 0 - -(EAX + 5), или dword ptr [ESP] = EAX + 5.

• Строки 16-19 демонстрируют операции со стеком: в него помещаются девять двойных слов, одно извлекается в EAX, после чего указатель стека корректируется так, чтобы указывать на то же место, что и до выполнения последовательности.

• Строка 20 проверяет EBX относительно регистра EAX и соответствующим образом задает флаги. Если флаги переопределяются до следующего использования, эта команда является мертвой.

• Строка 21 извлекает значение с вершины стека (где хранится EAX + 5) в регистр EAX.

Итак, код вычисляет EAX = EAX + 5.

Само собой разумеется, обфусцированный код совершенно не похож на код, сгенерированный компилятором, и определить функциональность фрагмента весьма трудно. В этом примере присутствует несколько методов обфускации:

• обфускация на основе шаблонов;

• развертывание констант;

• вставка мусорного кода;

• обфускация на основе стека;

• использование необычных команд, таких как RCL, SBB, PUSHF и PUSHAD.

Соответственно, чтобы привести код к форме, более близкой к исходной, можно использовать ряд существующих преобразований компилятора:

• локальная оптимизация;

• свертывание констант;

• удаление мертвых операторов;

• оптимизация стека.

Взаимодействие потока данных и потока управления

Рассмотрим следующую последовательность команд:

```
01: mov eax, dword ptr [ebp-10h]
02: jmp eax
```

Предположим, вы хотите построить "правильный", классический граф потока управления для программы, содержащей подобные последовательности. Чтобы определить, какая команда будет следующей после выполнения строки 2, - или, возможно, множество потенциальных последующих команд, - нужно определить множество возможных значений регистра EAX в этой точке. Иными словами, поток управления этого фрагмента зависит от потока данных, относящегося к местоположению [EBP-10h] в точке программы l1 (строка 1). Однако для определения потока данных относительно [EBP-10h] требуется определить поток управления относительно местоположения строки 1: нужно знать все возможные команды передачи управления (и связанный поток данных, ведущий к этим местоположениям), которые способны быть нацелены на местоположение строки 1. Невозможно содержательно говорить о потоке управления, не говоря одновременно о потоке данных, и наоборот.

Ситуация еще сложнее, чем может показаться на первый взгляд. Анализ программы пытается отвечать на вопросы вроде: "Какие значения местоположение `[EBP-10h]` может принимать при любых возможных обстоятельствах?" Для борьбы с вычислительной неразрешимостью и проблемами разрешимости многие формы анализа программ используют приближения пространства состояний. Некоторые приближения являются точными (например, приближение множества `{1,3}` множеством `{1,2,3}`), а некоторые - грубыми (например, приближение того же множества множеством `{0,1,...,2^32 - 1}`). (В этом абзаце "точные" и "грубые" не являются техническими терминами.) Если невозможно точно приблизить множество потенциальных значений местоположения `[EBP-10h]` (например, если приходится предположить, что местоположение может принимать любое возможное значение), то неизвестно, куда будет указывать переход, поэтому приходится считать, что он может быть нацелен на любое место в адресном пространстве. Тогда факты о потоке данных из местоположения строки 2 необходимо распространить на каждое другое местоположение. В практических условиях такое решение серьезно повлияет на анализ и, скорее всего, заставит его консервативно заключить, что во всех местоположениях возможны все состояния; это верно, но бесполезно.

Хуже того, если когда-либо приходится предположить, что переход может быть нацелен на любое местоположение, то из-за кодирования команд переменной длины в x86 многие из этих передач будут направлены в места, не соответствующие началу правильной команды. Такие фиктивные команды, скорее всего, нанесут хаос любому анализу, особенно в сочетании с наблюдениями предыдущего абзаца.

Академические работы в этой области, например Киндера[30] и Такура с соавторами[41], стремятся построить системы, способные возвращать правильные ответы для всех возможных входных данных. Такие системы предпочитают сообщить пользователям, что не могут определить точные сведения, вернуть правильные, но крайне неточные результаты или погибнуть в попытке (например, исчерпав всю доступную память или не завершившись из-за проблем вычислимости), но не выдавать ответ, который не полностью обоснован. Эта цель похвальна, учитывая исходную мотивацию этих дисциплин: обеспечить абсолютную правильность программ и анализов. Однако она не соответствует нашей мотивации как исследователей обфускации.

Деобфускация имеет иную природу, чем формальная верификация или анализ программ, даже если мы предпочитаем использовать методы, разработанные в этих контекстах. Обфускатор преобразует программу `Porig` в программу `Pobf`, а мы ищем либо транслятор из `Pobf` в `Porig`, либо достаточные сведения о `Pobf`, чтобы ответить на вопросы, близкие к некоторой задаче обратной разработки. Мы неохотно прибегаем к некорректным методам, но к концу дня предпочитаем иметь фактические результаты, поэтому можем использовать такие методы, хотя и осознанно и с неудовольствием.

Обфускации на основе данных

Начнем с методов обфускации, которые лучше всего описываются через их влияние на значения данных и вычисления, не связанные с управлением. В частности, предположим, что представленные фрагменты находятся внутри одного базового блока графа потока управления программы. Обсуждение обфускаций на основе управления и их сочетания с обфускациями на основе данных отложено до последующих разделов.

Развертывание констант

Свертывание констант - одна из самых ранних и основных оптимизаций компилятора. Цель этой оптимизации - заменить вычисления, результаты которых известны во время компиляции, этими результатами. Например, в операторе C `x = 4 * 5;` выражение `4 * 5` состоит из бинарного арифметического оператора (*), которому передаются два операнда со статически известными значениями (соответственно 4 и 5). Компилятору было бы расточительно генерировать код, вычисляющий этот результат во время выполнения, поскольку он может вывести результат во время компиляции. Компилятор может просто заменить присваивание на `x = 20;`.

Развертывание констант - обфускация, выполняющая обратную операцию. Имея константное значение, используемое где-либо во входной программе, обфускатор может заменить константу некоторым процессом вычисления, который ее создает. Эта обфускация уже встречалась в мотивирующем примере:

```
01: push 0F9CBE47Ah
02: add dword ptr [esp], 6341B86h
```

Если не учитывать изменения флагов этой последовательностью, было установлено, что она эквивалентна `push 0h`.

Схемы кодирования данных

Принципиальный недостаток этого метода в том, что константы необходимо динамически декодировать (тем самым раскрывая как их самих, так и функцию декодирования) во время выполнения до обработки. Имеется функция кодирования `f(x) = x - 6341B86h`, результат которой `f(x)` помещается в стек, после чего применяется декодирующая функция: `f^-1(x) = x + 6341B86h`.

Эта конструкция тривиальна; деобфускация выполняется простым применением стандартной оптимизации компилятора - свертывания констант.

Предпринимались попытки укрепить такие операторы и предложить более устойчивые схемы кодирования. Некоторые методы, например полиномиальное кодирование и кодирование вычетами, описаны в патенте US6594761 B1 Чоу, Джонсона и Гу[11]. Также часто используются аффинные отображения.

Что, если удастся найти такое кодирование, при котором для манипуляций с переменными их необязательно декодировать (эквивалентную операцию можно определить над закодированными переменными)? Это свойство, называемое гомоморфизмом, обсуждалось с точки зрения обфускации, равно как и усовершенствование метода кодирования вычетами, в таких работах, как труды Чжу и Томборсона[44].

В абстрактной алгебре гомоморфизм - это сохраняющее операции отображение между двумя алгебраическими структурами. Рассмотрим, например, две группы G и H, снабженные соответственно операциями `+g` и `+h`. Мы хотим построить отображение `f` между множествами, лежащими в основе G и H, и хотим, чтобы наше отображение учитывало операции `+g` и `+h`. В частности, должно выполняться `f(x +g y) = f(x) +h f(y)`.

Понятие гомоморфизма можно обобщить за пределы групп до произвольных алгебраических структур. Например, можно рассматривать гомоморфизмы колец, одновременно сохраняющие и операторы сложения, и операторы умножения. В отличие от отображений, сохраняющих только одну операцию кольца, но не другую, или накладывающих ограничения на операторы либо их применение, неограниченные отображения считаются полностью гомоморфными.

Полностью гомоморфные отображения имеют естественное применение в обфускации.

Если исходная алгебра является незакодированной областью, а целевая - закодированной, то гомоморфное отображение позволяет выполнять вычисления непосредственно над закодированными данными, не декодируя их предварительно и не кодируя повторно после.

На момент написания тема гомоморфной криптографии все еще находилась в зачаточном состоянии. Было показано существование полностью гомоморфных криптосистем, позволяющих вычислять зашифрованные программы над зашифрованными данными. Иными словами, можно делать строгие утверждения о трудности определения деталей выполняемой программы и данных, над которыми она работает. В настоящее время эти схемы слишком неэффективны для практического применения, а вопрос о том, как лучше всего применять технологию к произвольным компьютерным программам, остается открытым.

Вставка мертвого кода

Еще одна распространенная оптимизация компилятора называется удалением мертвого кода и отвечает за удаление операторов программы, не влияющих на ее работу. Например, рассмотрим следующую функцию C:

```
int f()
{
  int x, y;

  x = 1;    // это присваивание x является мертвым
  y = 2;    // y больше не используется, поэтому оно мертвое
  x = 3;    // выше этой строки x не является живым
  return x; // x является живым
}
```

В конечном счете функция возвращает число 3. Перед этим она выполняет несколько бессмысленных вычислений, не влияющих на ее выходное значение. Первые присваивания x и y называются мертвыми, поскольку не влияют на живые вычисления.

Обфускаторы выполняют обратную операцию, вставляя мертвый код, чтобы затруднить чтение кода: специалисту по обратной разработке приходится вручную решать, участвует ли данная команда в вычислении какого-либо значимого результата. Возможность вставлять "мертвый" код требует от обфускатора знать, какие регистры являются "живыми" в каждой точке программы; например, если EAX содержит важное значение (он живой), а EBX - нет (он мертвый), можно вставить операторы, изменяющие EBX.

Деобфускация этой конструкции выполняется простым применением стандартной оптимизации компилятора - удаления мертвых операторов, которое можно выполнить либо в одном базовом блоке, либо по всему графу потока управления.

Арифметическая замена с помощью тождеств

Можно сформулировать математические утверждения, связывающие результаты одних операторов с результатами сочетаний других операторов. Один пример этого общего явления уже встречался в мотивирующем примере: команда `XOR EAX, 0FFFFFFFFh` (двоичное представление `0FFFFFFFFh` состоит из одних единичных битов). Поскольку `0 XOR 1 = 1`, а `1 XOR 1 = 0`, эта команда в действительности инвертирует каждый бит EAX; иными словами, она синонимична оператору NOT. Аналогично можно сформулировать следующие утверждения:

• `-x = ~x + 1` (по определению дополнительного кода);

- `rotate left(x,y) = (x << y) | (x >> (bits(x)-y));`
- `rotate right(x,y) = (x >> y) | (x << (bits(x)-y));`
- `x-1 = ~-x;`
- `x+1 = -x`.

Обфускация на основе шаблонов

В основе обфускации на основе шаблонов, являющейся неотъемлемой частью многих современных защит, лежит простая концепция. Автор защиты вручную строит преобразования, отображающие одну или несколько соседних команд в более сложную последовательность команд с тем же семантическим эффектом. Например, шаблон может преобразовать последовательность

```
01: push reg32
```

в следующую последовательность (назовем ее #1):

```
01: push imm32
02: mov dword ptr [esp], reg32
```

Или он может преобразовать ту же последовательность в следующую (#2):

```
01: lea esp, [esp-4]
02: mov dword ptr [esp], reg32
```

Либо в эту (#3):

```
01: sub esp, 4
02: mov dword ptr [esp], reg32
```

Шаблоны могут быть сколь угодно сложными. Более сложный пример может заменить шаблон

```
01: sub esp, 4
```

следующим шаблоном (#4):

```
01: push reg32
02: mov reg32, esp
03: xchg [esp], reg32
04: pop esp
```

Некоторые средства защиты имеют сотни шаблонов. Большинство защит применяет шаблоны к входной последовательности случайным образом, поэтому две обфускации одного и того же фрагмента кода дают разный результат. Кроме того, шаблоны применяются итеративно. Рассмотрим следующий вход:

```
01: push ecx
```

Представим, что он преобразуется с помощью замены #3:

```
01: sub esp, 4
02: mov dword ptr [esp], ecx
```

Теперь предположим, что обфускатор запускается второй раз и первая команда заменяется согласно шаблону #4:

```
01: push ebx
02: mov ebx, esp
03: xchg [esp], ebx
04: pop esp
05: mov dword ptr [esp], ecx
```

Этот процесс можно продолжать бесконечно, получая выходную последовательность произвольно большого размера. При достаточном количестве шаблонов одну команду можно преобразовать в миллионы команд.

Обратите внимание на несколько особенностей этих замен. #1 и #2 сохраняют семантическую эквивалентность: после выполнения этих последовательностей процессор будет находиться в том же состоянии, в котором он оказался бы после выполнения исходной команды. #3 не сохраняет семантическую эквивалентность, поскольку использует команду `sub`, изменяющую флаги, тогда как исходная `push` их не изменяет. Что касается последовательности #4, исходная команда изменяет флаги, а замена - нет; кроме того, исходная команда вообще не изменяет память, тогда как замена записывает значение `ESP` на вершину стека (поэтому ее также можно считать эквивалентной команде `PUSH ESP`).

Эти соображения показывают трудность обфускации ассемблерного кода после компиляции. Защита может безопасно выполнить замену #3 лишь в том случае, если известно, что измененные командой флаги не используются до следующего изменения этих флагов. Замена #4 аналогично безопасна, если флаги мертвы и результирующий код безразличен к содержимому `[ESP]` после исходной операции `SUB ESP, 4`. Обеспечение живости флагов требует построения графа потока управления функции, что может быть трудно из-за косвенных ветвлений. Убедиться в безопасности изменения памяти стека было бы чрезвычайно трудно из-за наложения памяти. Эти конкретные опасения вряд ли повлияют на обычные функции, сгенерированные компилятором, для которых можно построить графы потока управления, но мы надеемся, что они показывают риски применения к скомпилированному коду преобразований, не являющихся семантически эквивалентными.

Из-за сложности обфускации скомпилированного языка ассемблера средства защиты чаще всего применяют эти преобразования к коду, соответствующему самой защите, а не к коду целевой программы. Так авторы защиты могут гарантировать, что входной код будет безразличен к преобразованиям, не сохраняющим строгую семантическую эквивалентность.

Деобфускация этого типа обфускации проста, хотя написание деобфускатора может занять много времени. Можно построить обратные замены шаблонов, отображающие целевые последовательности в исходные. Фактически это соответствует обычной оптимизации компилятора, называемой локальной оптимизацией. В академических работах, например Джейкоба с соавторами[25] или Бансала[1], обсуждалось автоматическое построение как шаблонных обфускаторов, так и локальных оптимизаторов.

Это возвращает нас к вопросу о практических и академических результатах. Предположим, вы имеете дело с обфускатором на основе шаблонов, содержащим ошибки (например, ошибочные замены шаблонов, не сохраняющие семантическую эквивалентность). Далее предположим, что вы как исследователь деобфускации знаете об ошибках и можете исправить их во время деобфускации. Это означает, что ваш деобфускатор аналогично не будет сохранять семантическую эквивалентность и, следовательно, окажется "неправильным" в абсолютном смысле с точки зрения преобразования, но фактически выдаст "правильные" результаты относительно кода до обфускации. Следует ли выполнить замену? Сторонники формальной корректности ответили бы отрицательно; мы ответим утвердительно.

Обфускация на основе управления

При обратной разработке кода, сгенерированного компилятором, специалисты могут полагаться на предсказуемость того, как компилятор переводит конструкции потока управления. Благодаря этому они быстро определяют структуру потока управления исходного кода на уровне абстракции выше языка ассемблера. В процессе специалист по обратной разработке опирается на множество предположений о том, как компиляторы генерируют код. В чисто скомпилированной программе весь код базового блока чаще всего расположен последовательно (интенсивные оптимизации компилятора могут сделать эту основную предпосылку недействительной). Связанные во времени блоки обычно также расположены рядом. Команда `CALL` всегда соответствует вызову некоторой функции. Команда `RET` тоже почти всегда означает конец некоторой функции и возврат к вызывающему коду. Косвенные переходы, например реализующие операторы `switch`, встречаются редко и следуют стандартным схемам.

Обфускация на основе управления атакует эти опоры стандартной обратной разработки, усложняя как статический, так и динамический анализ. Стандартные инструменты статического анализа делают те же предположения, что и специалисты по обратной разработке, в частности:

• Команда `CALL` используется только для вызова функций, а функция начинается по адресу, на который нацелен вызов.

• Большинство вызовов возвращает управление, и, если они это делают, возврат происходит в местоположение непосредственно после команды `CALL`; операторы `ret` и `RETN` обозначают границы функций.

• Встретив условный переход, дизассемблеры предполагают, что он помещен в код "добросовестно", в частности:

• обе ветви перехода могут быть практически выбраны;

• по обеим ветвям перехода расположен код, а не данные.

• Они смогут легко определить цели косвенных переходов.

• Косвенные переходы и вызовы будут генерироваться только для стандартных конструкций, например `switch` и вызовов через указатели на функции.

• Все передачи управления нацелены на местоположения кода, а не данных.

• Исключения будут использоваться предсказуемыми способами.

В отношении передач управления дизассемблеры предполагают модель "нормальности", основанную на шаблонах стандартного скомпилированного кода. Они явно создают функции по целям вызовов, завершают их по операторам возврата, продолжают дизассемблирование после команды вызова, проходят обе ветви всех условных переходов, считают все цели ветвления кодом, используют синтаксическое сопоставление с шаблоном для разрешения схем косвенных переходов и в целом игнорируют исключительный поток управления. Нарушение изложенных выше предположений приводит к очень плохому дизассемблированию. Это постоянная заноза для исследователей обфускации и открытая тема исследований (как обсуждалось ранее) в области верификации.

Динамическому анализу легче иметь дело с косвенными передачами управления, поскольку он может явно следовать за потоком выполнения. Однако атакующий все равно сталкивается с вопросами определения целей косвенных передач и страдает от отсутствия последовательной локальности, вызванного так называемым спагетти-кодом. В следующих разделах подробно рассматривается, что происходит, когда эти предположения нарушаются.

Встраивание функций и вынесение кода в функции

Граф вызовов программы несет значительную часть ее высокоуровневой логики. Манипуляции с понятием функции могут нарушить некоторые предположения специалиста по обратной разработке. Можно:

• встраивать функции - код подфункции объединяется с кодом вызывающей функции. Если подфункция вызывается несколько раз, размер кода может быстро вырасти;

• выносить код в функции - часть функции извлекается, преобразуется в независимую функцию и заменяется вызовом вновь созданных функций.

Сочетание этих двух операций по всей программе приводит к вырожденному графу вызовов без видимой логики. Само собой разумеется, прототипами функций также можно манипулировать: менять порядок аргументов, добавлять лишние, фиктивные аргументы и так далее, чтобы еще сильнее скрыть логику.

Уничтожение последовательной и временной локальности

Как уже говорилось и как интуитивно понимают специалисты по обратной разработке скомпилированного кода, команды внутри одного скомпилированного базового блока лежат в одной прямолинейной последовательности. Это свойство называется последовательной локальностью. Кроме того, оптимизаторы компилятора стараются помещать связанные друг с другом базовые блоки (например, блок и его последователей) рядом, чтобы максимально повысить локальность кеша команд и уменьшить число ветвлений в скомпилированном выводе. Это свойство мы называем последовательной локальностью связанного во времени кода. При обратной разработке скомпилированного кода эти свойства обычно выполняются. Анализируя такой код, человек привыкает к тому, что весь код, отвечающий за одну единицу функциональности, аккуратно содержится в одной области, а ближайшие соседи по потоку управления расположены рядом и также последовательно.

Очень старый метод обфускации программ - ввод безусловных ветвлений, разрушающих эту привычность, которую специалисты по обратной разработке естественным образом приобретают в обычной работе. Вот простой пример:

```
01: instr_1:
02:   push offset caption
03:    jmp instr_4

04:
05: instr_2:
06:    call MessageBoxA
07:    jmp instr_5
08:
09: instr_3:
10:    push 0
11:    jmp instr_2
12:
13: start:
14:    push 0
```

```
15:    jmp instr_1
16:
17: instr_4:
18:    push offset dlgtxt
19:   jmp instr_3
20:
21: instr_5:
22: ; ...
```

Этот пример показывает отсутствие последовательной локальности команд внутри базового блока, но не временной локальности нескольких базовых блоков. На практике большие объемы кода программы будут переплетены таким образом (обычно в каждом базовом блоке более одной команды, в отличие от предыдущего примера).

С формальной точки зрения этот метод не заслуживает даже названия "тривиального", поскольку вообще не влияет на семантику программы. Построение графа потока управления и удаление ложных безусловных ветвлений полностью побеждает эту схему. Однако с точки зрения анализа, выполняемого человеком вручную, возможность следовать за кодом резко замедляется.

Косвенное управление на основе процессора

Для большинства процессоров двумя важнейшими примитивами перемещения являются ветвление наподобие `JMP` и сохранение указателя команд с ветвлением наподобие `CALL`. Эти примитивы можно обфусцировать с помощью динамически вычисляемых адресов ветвления или их эмуляции. Один из самых простых методов - пара `PUSH-RET`, используемая как команда `JMP`:

```
01: push target_addr
02: ret
```

Она (почти) семантически эквивалентна следующему:

```
01: jmp target_addr
```

Команда `CALL` является легкой целью для обфускаторов, поскольку большинство дизассемблеров предполагает следующее о ее высокоуровневой семантике:

• целевой адрес является точкой входа подфункции;

• вызов возвращает управление (то есть выполняется команда после `CALL`).

На самом деле эти предположения легко нарушить. Рассмотрим следующий пример:

```
01: call target_addr
02: <junk code>
03: target_addr:
04: add esp, 4
```

`CALL` используется как `JMP`; управление никогда не вернется к строке 2. Стек исправляется (адрес возврата удаляется из стека) в строке 3. Теперь рассмотрим следующие два элемента:

```
01: basic_block_a:
02: add [esp], 9
03: ret
```

и:

```
01: basic_block_b:
02: call basic_block_a
03: <junk code>
04: true_return_addr:
05: nop
```

Команда CALL в строке 2 basic_block_b указывает на basic_block_a, который в действительности является лишь заглушкой, обновляющей (см. строку 2 basic_block_a) адрес возврата, сохраненный на вершине стека, прежде чем команда RET использует его (строка 3 basic_block_a). В этих двух примерах результатом является промежуток между естественным (ожидаемым) и фактическим адресами возврата CALL; обфускатор может (и будет) использовать его для вставки кода, который мешает дизассемблерам и создает путаницу.

Следующий пример является интересным развитием стандартной пары PUSH-RET, ранее использованной как JMP:

```
01: push addr_branch_default
02: push ebx
03: push edx
04: mov ebx, [esp+8]
05: mov edx, addr_branch_jmp
06: cmovz ebx, edx
07: mov [esp+8], ebx
08: pop edx
09: pop ebx
10: ret
```

Основой этой конструкции в действительности является PUSH-RET. Строка 7 записывает целевой адрес в стек; он используется командой RET в строке 10. Помещаемый в стек адрес берется из EBX (строка 7), который условно обновляется командой CMOVZX в строке 6. Если условие выполняется (проверяется флаг Z), команда действует как стандартная MOV (EBX перезаписывается значением EDX, содержащим целевой адрес ветвления); в противном случае она действует как NOP (поэтому EBX содержит стандартный адрес ветвления). В итоге ясно видно, что этот шаблон представляет условный переход (JZ).

Косвенное управление на основе операционной системы

Программа может использовать примитивы операционной системы (хотя это может означать потерю переносимости). В Windows для обфускации потока управления часто применяются Structured Exception Handler (SEH), Vectored Exception Handler (VEH) и Unhandled Exception Handler, а в Unix - обработчики сигналов и функции setjmp/longjmp.

Основной алгоритм можно разложить на следующие этапы:

1. Обфусцированный код вызывает исключение (используя недопустимый указатель, недопустимую операцию, недопустимую команду и т. д.).

2. Операционная система вызывает зарегистрированный обработчик (или обработчики) исключения.

3. Обработчик исключения распределяет поток команд согласно своей внутренней логике и возвращает программу в чистое состояние.

Следующий пример миллиарды раз встречался в двоичных файлах x86:

```
01: push addr_seh_handler
02: push fs:[0]
03: mov fs:[0], esp
04: xor eax, eax
05: mov [eax], 1234h
06: <junk code>
07: addr_seh_handler:
08: <continue execution here>
09: pop fs:[0]
10: add esp, 4
```

Строки 1-3 настраивают SEH. Затем в виде нарушения доступа вызывается исключение: строка 5 пытается выполнить запись по адресу `0x0`. Если программа не отлаживается, операционная система передаст выполнение обработчику SEH. Обратите также внимание: при вызове обработчика SEH он получает в качестве одного из аргументов копию контекста потока, а значение регистра указателя команд можно изменить, чтобы еще сильнее обфусцировать перенаправление потока управления.

> **Примечание.** Этот метод также эффективно действует как средство против отладчика. По сути, задача отладчика - обрабатывать исключения. Эти исключения необходимо передавать целевой отлаживаемой программе; в противном случае ее поведение изменится и вмешательство будет обнаружено.

Что еще интереснее, эту концепцию можно обратить. Что, если защита вставляет исключения в исходную программу и перехватывает их собственным подключенным отладчиком? Защищенная программа состоит из отлаживаемой программы и отладчика. Известный пример - возможность `namomites` в Armadillo. `Namomites` фактически заменяют (условные) переходы командой `INT 3`. Исключение перехватывается отладчиком защиты, который соответствующим образом обновляет контекст отлаживаемой программы для эмуляции (условных) переходов. Нельзя просто отсоединить отладчик от отлаживаемой программы: в противном случае исключения не будут обработаны и программа аварийно завершится. Реализация этой концепции была предложена Дероко[19].

Непрозрачные предикаты

Непрозрачный предикат (введенный Коллбергом в работах "A Taxonomy of Obfuscating Transformations"[12] и "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs"[13]) - это специальная условная конструкция (логическое выражение), которая всегда вычисляется либо в истину, либо в ложь (обозначаются соответственно `PT` и `PF`). Его значение известно только во время компиляции/обфускации и должно быть неизвестно атакующему, а его доказательство должно быть вычислительно трудным, чтобы обеспечить достаточную степень устойчивости.

В сочетании с командой условного перехода он вводит дополнительную ложную ветвь, то есть дополнительное ребро в графе потока управления (control-flow graph, CFG). Эту мертвую ветвь можно использовать для вставки мусорного кода или специальных свойств, например циклов в CFG, чтобы усложнить анализ. Однако ложная ветвь должна выглядеть достаточно правдоподобно, чтобы избежать простого обнаружения человеком-атакующим (например, только одна из двух ветвей содержит необходимые инициализации переменных).

Он имеет вид условного перехода, но семантику безусловного. Для реализации непрозрачных предикатов можно применять вычислительно сложные математические задачи. Можно также использовать некоторые переменные среды, значения которых постоянны и известны во время компиляции/обфускации. Последний метод может быть менее устойчивым, поскольку множество переменных-кандидатов ограничено и конечно, что ограничивает потенциальное разнообразие.

Проектирование устойчивых непрозрачных предикатов - трудная задача. Это лишние фрагменты кода, смешанные с существующим кодом, имеющим собственную логику/стиль; без особой осторожности их легко обнаружить. Хорошая практика - создавать зависимости между предикатом и состоянием/переменными программы. Человек-атакующий (вы) обычно весьма эффективно обнаруживает сомнительные шаблоны. Абсурдно сложный предикат может эффективно помешать инструменту статического анализа, но, скорее всего, будет легко обнаружен человеком-атакующим.

Интересная разновидность исходной концепции использует предикат, который случайным образом возвращает либо истину, либо ложь (обозначается `P?`). Поскольку во время выполнения потенциально исполняются обе ветви, они должны быть семантически эквивалентны. В большинстве случаев это сводится к клонированию (и, возможно, диверсификации) базового блока (или более крупного фрагмента кода), создающему "ромбовидную" конструкцию.

Одновременная обфускация потока управления и потока данных

Для ясности до сих пор мы разделяли обфускацию потока управления и потока данных. Однако на практике они тесно связаны. В этом разделе представлены методы, основанные на их взаимодействии.

Вставка мусорного кода

Этот метод тесно связан с обфускацией потока управления. По сути, он состоит во вставке мертвого (то есть никогда не выполняемого) блока кода между двумя допустимыми блоками кода. Цель - полностью сбить с толку дизассемблер, который уже обманом заставили следовать по недопустимому пути (типичный случай непрозрачных предикатов). Команды внутри мусорного кода могут быть частично недопустимыми или создавать ветвления на недопустимые адреса (например, в середину допустимых команд), чтобы чрезмерно усложнить CFG.

Самый тривиальный пример вставки мусорного кода может выглядеть так:

```
01: jmp label
02: <junk>
03: label:
04: <real code>
```

Вот нечто чуть более сложное, использующее фиктивный непрозрачный предикат:

```
01: push eax
02: xor eax, eax
03: jz 9
04: <junk code start>
05: jg 4
06: inc esp
07: ret
08: <junk code end>
09: pop eax
```

Условный переход в строке (адресе) 3 всегда истинен, поскольку регистр `EAX` обнуляется командой `XOR` в строке 1. Это означает, что имеется шесть байтов мусорного кода. Этот мусорный блок использует команды, которые повлияют на дизассемблер, создавая новую ветвь и, по всей видимости, вставляя конец функции (команду `RET` в строке 9).

При правильной генерации блоки мусорного кода бывает довольно трудно заметить с первого взгляда. Чаще всего они удаляются из области досягаемости дизассемблера как побочный эффект деобфускации потока управления (см. Control Flow Deobfuscation via Abstract Interpretation). В последнем примере, если непрозрачный предикат распознан как таковой, к блоку мусорного кода больше не ведет ни одного пути. Как и все остальные методы, если он недостаточно диверсифицирован - например, использует ограниченную базу статических шаблонов, - его устойчивость и сила обычно минимальны.

Выравнивание графа потока управления

Основная идея выравнивания графа - заменить все структуры управления одним оператором `switch`, называемым диспетчером. Выбирается подграф графа потока управления программы (реализации часто работают на уровне функций) и преобразуется; при этом базовые блоки могут перерабатываться (разделяться или объединяться). Затем каждый базовый блок отвечает за обновление контекста диспетчера (то есть состояния подпрограммы), чтобы диспетчер мог связаться со следующим базовым блоком (см. рисунок 5-1). Связи между базовыми блоками теперь "скрыты" внутри операций манипулирования контекстом диспетчера. Условные переходы (как в блоке d) можно легко эмулировать с помощью проверки флагов и команд `IMUL` либо простых команд `CMOV`.

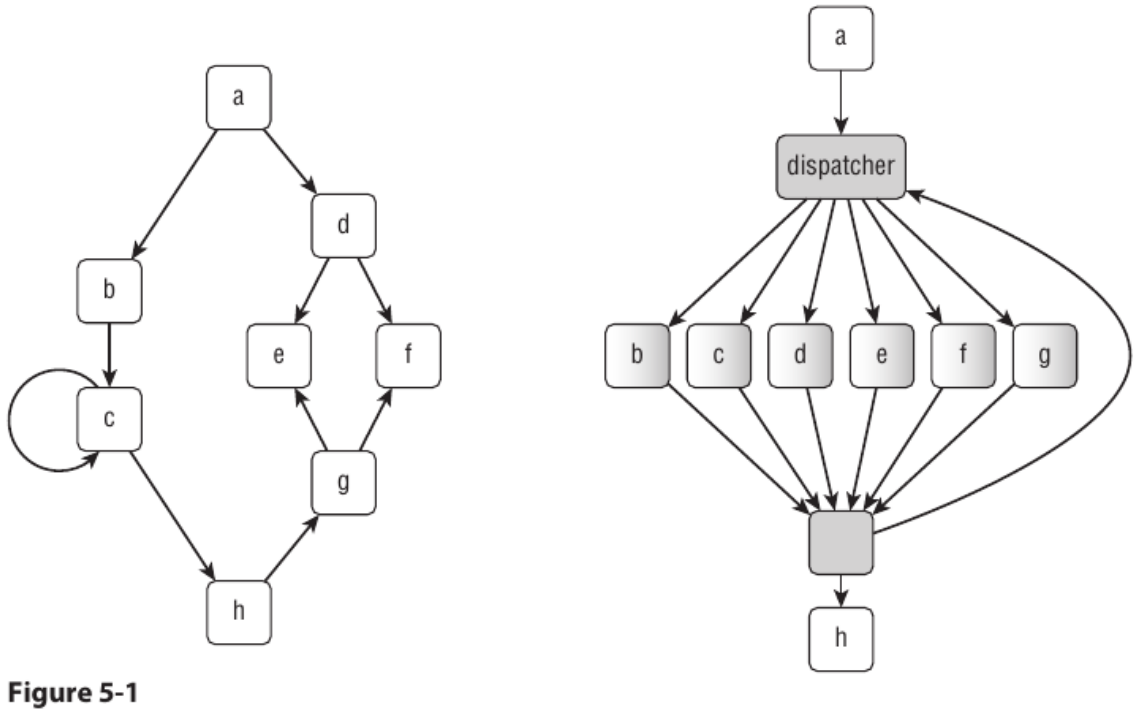

Figure 5-1

Рисунок 5-1. Исходный и выровненный графы потока управления

Само собой разумеется, значительная часть устойчивости этого метода к статическому анализу зависит от способности обфусцировать манипуляции с контекстом и переходы. Для усложнения задачи можно реализовать различные возможности, например межпроцедурные связи, наложение указателей, вставку фиктивных состояний и так далее.

Подобно непрозрачным предикатам, выравнивание CFG можно также использовать для вставки путей мертвого кода и ложных базовых блоков. О выравнивании графа и способах укрепления его реализации можно сказать многое. Результирующий граф не дает никаких подсказок о структуре алгоритма, а код диспетчеризации и манипулирования контекстом также создает накладные расходы, способствующие сокрытию защищенного кода. Концептуально этот метод совпадает с виртуализацией кода (виртуальной машиной); его можно рассматривать как частичную виртуализацию, нацеленную (виртуализирующую) только поток управления, но не поток данных.

Если вы хотите самостоятельно увидеть выровненный код, просто возьмите копию подключаемого модуля Flash (например, `NPSWF32.dll`), дизассемблируйте файл и найдите функции самого большого размера. Выровненные функции легко распознать.

Виртуальные машины

Виртуальные машины (VM) - мощный класс программных защит и особенно сложное преобразование. VM по сути состоит из интерпретатора и некоторого байт-кода. Язык, поддерживаемый интерпретатором, определяется защитой. Во время компиляции выбранные части кода компилируются для целевой архитектуры VM (перенацеливаются), а затем вставляются в защищенную программу вместе с соответствующим интерпретатором. Во время выполнения интерпретатор берет на себя исполнение байт-кода (то есть перевод с целевой архитектуры на исходную). VM обычно влекут значительные накладные расходы на производительность (особенно процессорное время), поэтому обычно виртуализируются только конкретные выбранные части исходной программы.

К примерам известных защит, основанных на VM, относятся VMProtect и CodeVirtualizer. Позднее мы углубимся в восхитительное занятие анализа VM. Пока достаточно сказать, что атакующему необходимо понять интерпретатор, чтобы проанализировать байт-код и в конечном счете создать компилятор с целевой архитектуры на родную (девиртуализация).

Криптография белого ящика

Когда защищаемое приложение не может основывать свою безопасность на использовании аппаратного компонента или сетевого сервера, приходится предположить наличие атакующего, способного выполнять приложение в полностью контролируемой им или ею среде. Соответствующая этой ситуации модель атакующего, называемая контекстом атаки белого ящика (white-box attack context, WBAC), требует особой программной реализации классических криптографических примитивов.

Такие механизмы специально создаются для обеспечения конфиденциальности секретного ключа внутри алгоритма. Подобное преобразование (сокрытие ключа в алгоритме шифрования, с помощью взаимодействия со средой или без него) можно формализовать как преобразование обфускации.

В этом разделе описаны некоторые отрицательные и положительные результаты, касающиеся обфускации кода, и их влияние на проблему управления ключами.

Вероятностный алгоритм O является обфускатором, если он удовлетворяет следующим свойствам, приведенным Бараком с соавторами[2]:

• P и O(P) вычисляют одну и ту же функцию.

• Рост времени выполнения и пространства O(P) не более чем полиномиален относительно времени выполнения и пространства программы P.
• Для любого вероятностного алгоритма A с полиномиальным временем существует вероятностный алгоритм S с полиномиальным временем и пренебрежимо малая функция m (пренебрежимо малая функция растет намного медленнее обратной величины любого полинома), такие, что для всех программ P:

```
| p[A(O(P))=1] - p[S^P(1^{|P|})=1] | ≤ m(|P|)
```

Свойство виртуального черного ящика выражает тот факт, что распределение выходов любого вероятностного алгоритма анализа A, примененного к обфусцированной программе O(P), почти всюду равно распределению выходов симулятора S, имеющего оракульный доступ к программе P. (Программа S не имеет доступа к описанию программы P, но для любого входа x ей предоставляется доступ к P(x) за время, полиномиальное относительно размера P. Оракульный доступ к программе P эквивалентен доступу только к входным/выходным данным программы P.)

Интуитивно свойство виртуального черного ящика просто требует, чтобы все, что можно вычислить из обфусцированной версии O(P), можно было также вычислить через оракульный доступ к P.

Один из главных фактов о таком идеальном обфускаторе состоит в том, что его не существует. Доказательство основано на построении программы, которую невозможно обфусцировать. Этот результат о невозможности показывает, что генератора виртуального черного ящика, способного защитить код любой программы, не позволяя ему раскрывать больше сведений, чем раскрывают его входные/выходные данные, не существует. Этот результат естественным образом влечет важные последствия для разработчиков механизмов обфускации (адаптированных к контексту WBAC).

Рассмотрим практическое применение обфускации, состоящее в преобразовании симметричного шифрования в асимметричное путем обфускации схемы шифрования с закрытым ключом. Если существует схема шифрования с закрытым ключом, то существует и необфусцируемая схема шифрования с закрытым ключом. Это ясно показывает, что не все схемы шифрования с закрытым ключом хорошо подходят для обфускации.

Обратите внимание: этот результат не доказывает отсутствия некоторой схемы шифрования с закрытым ключом, для которой атакующему можно предоставить схему, вычисляющую алгоритм шифрования без потери безопасности. Однако он доказывает, что не существует общего метода, позволяющего преобразовать любую схему шифрования с закрытым ключом в систему шифрования с открытым ключом путем обфускации алгоритма шифрования.

Проблема построения схемы шифрования с закрытым ключом, удовлетворяющей свойству виртуального черного ящика (и потому устойчивой в контексте WBAC), по-прежнему интересует исследователей криптографии, даже если результат о невозможности общего способа сделать это может казаться обескураживающим. Предложения реализаций DES и AES белого ящика иллюстрируют этот интерес.

Обфускация с помощью сети закодированных таблиц поиска позволяет получить версии алгоритмов DES и AES, более устойчивые в контексте атаки белого ящика. Однако эффективный криптоанализ реализаций белого ящика DES (например, выполненный Губеном[24]) и AES (Бийе[5])

установил, что проблема построения схемы шифрования с закрытым ключом, удовлетворяющей свойству виртуального черного ящика, остается нерешенной.

Идеальную модель обфускатора, способного преобразовать любую программу в виртуальный черный ящик, реализовать невозможно. В частности, не существует общего преобразования, позволяющего, исходя из алгоритма шифрования и ключа, получить обфусцированную версию этого алгоритма, которую можно было бы опубликовать без утечки сведений о содержащемся в ней ключе.

Однако этот формализм не устанавливает невозможность сокрытия ключа в алгоритме с целью преобразования алгоритма с закрытым ключом в шифрование с открытым ключом.

Был опубликован метод (Чоу[9]), затрудняющий извлечение ключа в контексте белого ящика. Принцип состоит в реализации специализированной версии алгоритма DES, в которую встроен ключ K и которая способна выполнять лишь одну из двух операций: шифрование или расшифрование. Эта реализация устойчива в контексте белого ящика, поскольку трудно извлечь ключ K путем наблюдения за выполняемыми программой операциями, а также трудно создать функцию расшифрования на основе реализации функции шифрования и наоборот.

Основная идея - выразить алгоритм как последовательность (или сеть) таблиц поиска и обфусцировать эти таблицы путем кодирования их входов/выходов. Все операции блочного шифра, например сложение по модулю 2 с раундовым ключом, встраиваются в эти таблицы поиска. Таблицы рандомизируются, чтобы обфусцировать их работу.

Обфускация AES (описанная Чоу[10]) выполняется аналогично DES. Цель по-прежнему состоит во встраивании раундовых ключей в код алгоритма, чтобы не хранить ключ в статической памяти и не загружать его в динамическую память во время выполнения. Для безопасного встраивания этих ключей применяется тот же метод, что и для DES: представить AES как сеть таблиц поиска и применить кодирование входов/выходов, чтобы скрыть ключи.

Достижение безопасности за счет неясности

До сих пор вы познакомились с большим числом методов обфускации. Большинство из них - простые преобразования, которые на первый взгляд кажутся довольно слабыми, и по отдельности они действительно слабы. Как построить безопасность или доверие из таких примитивов? Сила системы обфускации (или обфускатора) возникает из итеративного и совместного применения набора этих методов. Каждое последовательное применение простого метода накапливается в сильное неразличимое глобальное преобразование (во всяком случае, такова цель). Якубовский с соавторами[26] предложил интересную аналогию между раундовой криптографией и итеративной обфускацией. Раунд криптографического алгоритма состоит из базовых арифметических операций (сложения, исключающего ИЛИ и т. д.), выполняющих тривиальные преобразования входных данных. Взятый отдельно, раунд слаб и подвержен многим формам атак. Тем не менее многократное применение набора раундов может дать достаточно безопасный алгоритм. Именно это является целью обфускатора. Цель атакующего - различить раунды в глобальной обфусцированной форме и атаковать их в самых слабых местах.

Помните: даже если обфускатор несовершенен, как только он в достаточной степени повышает планку, необходимую для взлома защищенного кода, этого может оказаться достаточно для защищающейся стороны. Например, если на взлом новой версии программного обеспечения требуется несколько недель или месяцев, защищающаяся сторона может использовать этот период для работы над новой защитой, обновлениями протоколов и так далее и тем самым всегда оставаться впереди.

Обзор методов деобфускации

Теперь, когда вы лучше понимаете обфускацию кода, возникает вопрос: как вам, специалисту по обратной разработке, принять этот вызов? Какие средства и инструменты находятся в вашем распоряжении для взлома обфусцированного кода? Ручной анализ обфусцированного кода - утомительная, если не невозможная задача; вам захочется свести проблему к анализу чистого кода.

Поскольку ручной подход с использованием стандартных инструментов анализа программ трудоемок, а аналитик может столкнуться с широким разнообразием механизмов обфускации, необходимо найти модели и критерии для проектирования и оценки алгоритмов деобфускации. В этом разделе приводится краткий обзор проблемы с более теоретической точки зрения и описываются некоторые хорошо изученные формальные методы, с помощью которых можно создавать более универсальные инструменты деобфускации и максимально автоматизировать задачи аналитика.

Природа деобфускации: обращение преобразования

Для отмены преобразования обфускации доступны несколько методов анализа программного обеспечения. В этом разделе рассматриваются:

• понятие разрешимого приближения;

• некоторые статические или динамические методы, которые можно применять, и преимущества гибридных статико-динамических методов (некоторые из них представлены позднее с помощью специализированных инструментов);

• некоторые критерии, которые всегда можно применять для оценки алгоритма анализа и из которых можно вывести критерии безопасности относительно устойчивости обфускации (и, двойственным образом, эффективности преобразования деобфускации);

• открытые проблемы и новые тенденции в области гибридного динамического/статического анализа и формализации деобфускации.

Тема обширна, и в литературе до сих пор нет согласия относительно терминологии различных специализированных направлений исследований. Поэтому цель этого раздела - предоставить читателям некоторые ключевые слова для формирования общего представления и полезные ссылки для заинтересованных читателей, желающих дополнить свои знания в этой области.

В области анализа программ можно наблюдать несколько дихотомий. Некоторые методы анализа описываются как статические или динамические, хотя порой это различие кажется довольно искусственным. (Это различие обсуждается Яннисом Смарагдакисом и Кристофом Цальнером[38].) В других случаях алгоритмы анализа характеризуются как корректные или полные, но эти важные характеристики могут иметь в литературе разные значения. Наконец, анализы программ описываются как приближения сверху или снизу, однако и это различие кажется несколько искусственным, поскольку некоторые методы анализа, по-видимому, используют оба вида приближения.

В оставшейся части этого раздела обсуждаются как "синергия", так и "двойственность" статического и динамического анализа (также рассмотренные Майклом Д. Эрнстом[20]): сначала вводится формальная модель абстрактной интерпретации, а затем приводятся несколько примеров анализа в связи с деобфускацией.

Поиск разрешимого приближения конкретной семантики

Цель любого анализа программы - проверить, удовлетворяет ли программа определенному свойству. К сожалению, для любого нетривиального свойства этот вопрос в общем случае неразрешим, то есть нельзя разработать алгоритм, определяющий, выполняется ли свойство для программы. Чтобы преодолеть эту трудность, одним из решений является абстрагирование конкретных вариантов поведения программы в разрешимое приближение. Цель абстрактной интерпретации - формализовать эту идею приближения в единой системе. (Читатели могут обратиться к статье Патрика Кузо и Радии Кузо[14].)

Семантика программы представляет все возможные конкретные варианты ее поведения, включая взаимодействие с любой возможной средой компьютерной системы. К наиболее точным (конкретным) семантикам относятся так называемые семантики трасс. Эта семантика включает все конечные и бесконечные последовательности состояний и переходов. Если X - множество трасс выполнения (конечных и бесконечных), семантику трасс можно выразить как наименьшее решение (для вычислительного частичного порядка) уравнения неподвижной точки `X=F(X)`.

Абстрактный домен - это абстракция конкретной семантики. Цель абстрактной интерпретации - предоставить вычислимые приближения неподвижной точки абстрактных доменов, тем самым определяя вычислимые абстрактные семантики. Очевидно, чем грубее абстрактная семантика, тем на меньшее число вопросов она способна ответить.

Все абстракции семантики можно организовать в иерархию (описанную Кузо[16]) от самых точных до самых грубых. Точнее, абстрактные семантики можно разместить на решетке, а частичный порядок приближения этой решетки можно использовать для характеристики конкретности (или точности) абстрактных семантик и, следовательно, множеств вопросов, на которые они способны ответить.

Абстрактная интерпретация обычно применяется к статическому анализу посредством приближения конкретной семантики сверху. Можно заметить, что двойственным образом и согласно "принципу двойственности" теории решеток она должна также применяться к динамическому анализу, хотя в настоящее время работ на эту тему немного. В следующем разделе вы увидите, что связи и синергия между статическим и динамическим анализом приводят к практическим гибридным динамико-статическим методам, позволяющим как динамическому подходу выиграть в охвате, так и статическому - в точности.

Динамический и статический анализ образуют континуум

Статический анализ - дисциплина автоматического вывода сведений о компьютерных программах без их запуска (поэтому он применяется к "статическому" представлению программы). Статический анализ пытается вывести свойства (инварианты), выполняющиеся для всех запусков программы, посредством консервативного приближения ее конкретной семантики сверху.

Пример такого статического анализа - алгоритм распространения констант, стремящийся определить для каждой команды программы, имеет ли переменная постоянное значение всякий раз, когда поток управления достигает этой команды. Сведения о константах полезны в контексте компиляции, оптимизации и повторной компиляции программы. Например, они используются для удаления мертвого кода и мертвых путей выполнения (заменяя все случаи использования константных переменных их константными значениями, можно выявить постоянные условные ветвления, обусловленные постоянными предикатами).

Среди множества методов оптимизации в контексте обратной разработки следует помнить о методах частичного вычисления (описанных Бекманом с соавторами[3]). Частичный вычислитель

специализирует программу относительно части ее входных данных. Ожидается, что процесс специализации сохранит конкретную семантику программы, а синтаксическое представление результирующей программы будет оптимизировано для используемого класса входов и в результате станет проще для понимания.

Другой важный класс методов оптимизации включает методы срезов (описанные Вайзером[42]), которые также стремятся упростить рассматриваемую программу, но в данном случае путем удаления частей программы, не относящихся к критерию, предоставленному аналитиком. Критерий статического среза включает множество переменных и выбранную точку интереса. Критерий динамического среза дополняет статический критерий сведениями, соответствующими некоторому конкретному выполнению. Срезы представляют большой интерес в контексте обратной разработки, поскольку они отражают то, как специалист по обратной разработке мысленно создает срез программы, пытаясь понять ее внутреннюю работу.

В отличие от статического анализа, динамический анализ - дисциплина автоматического вывода сведений о выполняющейся компьютерной программе. Динамический анализ выводит свойства, выполняющиеся для одного или нескольких запусков программы, посредством точного приближения снизу.

Распространенный метод динамического анализа - динамическое тестирование, которое выполняет программу с несколькими входными данными и проверяет реакцию программы. Как правило, тестовые случаи исследуют лишь подмножество возможных выполнений программы.

Чтобы расширить охват динамического тестирования, принцип символьного выполнения (описанный Бойером[6]) использует символьные значения вместо конкретных входов. В каждой точке символьного выполнения обновляется символьное состояние программы. Это символьное состояние состоит из символьного хранилища и ограничения пути. Символьное хранилище содержит символьные значения, а ограничение пути - формулу, фиксирующую историю всех условных ветвлений, выбранных до текущей команды.

В заданной команде программы можно использовать решатель ограничений (решатель SMT или SAT) для определения соответствующего ограничения пути. Удовлетворяющее присваивание предоставляет конкретные входные данные, с которыми программа достигает данной команды. Генерируя новые тесты и исследуя новые пути, можно увеличить охват динамического тестирования.

К сожалению, ограничения, сгенерированные во время символьного выполнения, могут оказаться слишком сложными для решателя. Если решатель ограничений не способен вычислить удовлетворяющее присваивание, невозможно определить, осуществим путь или нет.

Конколическое выполнение (описанное Годфруа[23] и Сеном[37]) во многих ситуациях решает эту проблему. Идея заключается в одновременном символьном и конкретном выполнении программы. Когда ограничение пути слишком сложно для решателя, можно использовать конкретные сведения для упрощения ограничения (обычно заменяя некоторые символьные значения конкретными). После этого можно ожидать нахождения удовлетворяющего присваивания упрощенного ограничения.

Поскольку символьное выполнение не способно обрабатывать неограниченный цикл, приводящий к бесконечным путям символьного выполнения, оно должно приближать конкретную семантику программы снизу. Такое упрощение можно выполнить, установив некоторое произвольное ограничение цикла. Другое решение - использовать символьное выполнение вместе со статическим анализом, выводящим инварианты циклов.

По-видимому, динамический и статический подходы к анализу образуют континуум. В качестве иллюстрации динамическое тестирование, символьное выполнение и абстрактная интерпретация являются тремя способами приближения конкретной семантики программы. Динамический анализ использует конкретные значения и исследует подмножество конкретных переходов. Символьное выполнение явно находится между динамическим тестированием и статическим анализом.

Оно опирается на более абстрактную семантику, но также на приближение снизу. Абстрактный интерпретатор приближает конкретную семантику программы сверху.

Однако границу между этими подходами к анализу определить не так просто. Например, символьное выполнение можно определить как логический абстрактный интерпретатор, работающий над абстрактным доменом логических формул.

В заключение, многие методы статического анализа улучшаются посредством уточнения на основе динамического анализа. И наоборот, охват многих методов динамического анализа можно увеличить традиционными методами статического анализа. Таким образом, исследование гибридных динамико-статических подходов представляет большой интерес, особенно в контексте обратной разработки. Для описания этой синергии можно использовать критерии корректности и полноты.

Корректность и полнота

Любую задачу анализа программы можно сформулировать как проверку того, что программа удовлетворяет свойству. Для характеристики алгоритма анализа можно использовать два фундаментальных понятия: его корректность и полноту. Эти понятия, традиционно применяемые к логическим системам, можно также применять к анализу программ. К сожалению, из-за их двойственной природы (корректность и полнота соответствуют обратным импликациям в логике) в литературе до сих пор нет согласия относительно их применения к различным специализированным областям исследований.

Для заданного свойства корректный анализ программы обнаруживает все нарушения свойства. Однако, поскольку он приближает варианты поведения программы сверху, он может также сообщать о нарушениях свойства, которые не способны возникнуть. Например, корректный алгоритм обнаружения ошибок выявляет каждую возможную ошибку, хотя некоторые из них могут никогда не возникнуть во время выполнения.

Корректный алгоритм частичного вычисления сохраняет конкретную семантику исходной программы в том смысле, что специализированная программа не выдает ни одного выходного значения, не выдаваемого исходной программой (хотя она может оказаться неспособной выдать их все).

Корректное символьное выполнение гарантирует: если символьный путь ограничения выполним, должен существовать конкретный путь выполнения, достигающий соответствующего конкретного состояния (хотя у некоторого достижимого конкретного состояния может не быть соответствующего символьного состояния).

Корректный абстрактный интерпретатор сохраняет конкретную семантику программы. Если он утверждает, что для программы возможно оптимизирующее преобразование, то оптимизацию можно применить, не нарушив семантику программы. Однако обратите внимание: для некоторых оптимизаций он может оказаться неспособен ответить на вопрос. Он может утверждать, что оптимизация небезопасна, даже если преобразование на самом деле можно применить (без какого-либо разрушительного эффекта). Некоторые потенциальные оптимизации не будут применены. Корректность абстрактного интерпретатора относится к тому, на какие вопросы он способен ответить правильно, несмотря на потерю сведений. В этом смысле он консервативен.

Технически наименьшие неподвижные точки, вычисленные абстрактным интерпретатором, представляют по меньшей мере все конкретные состояния, возникающие во время выполнения.

Например, алгоритм распространения констант корректен, когда любая обнаруженная им константа действительно является константой. Однако некоторые константы могут остаться необнаруженными.

Для заданного свойства полный алгоритм анализа сообщает о нарушении свойства лишь при наличии конкретного нарушения свойства. Однако, поскольку он приближает варианты поведения программы снизу, о некоторых конкретных нарушениях свойства может не сообщаться.

Полный алгоритм частичного вычисления приводит к созданию специализированной программы, способной выдавать те же выходные значения, что и исходная программа для предполагаемых входных значений. Если он некорректен, он может выдавать неожиданные выходные значения (то есть не выдаваемые исходной программой).

Полное символьное выполнение охватывает все конкретные переходы. Оно гарантирует: если конкретное состояние выполнения достижимо, должно существовать соответствующее символьное состояние. Поскольку символьное выполнение не способно обрабатывать неограниченный цикл, приводящий к бесконечным путям символьного выполнения, оно должно приближать конкретную семантику программы снизу (обычно посредством некоторого ограничения цикла). Поэтому алгоритмы символьного выполнения чаще всего неполны.

Полный абстрактный интерпретатор является наиболее точным для ответа на заданное множество вопросов. Технически это означает, что каждое состояние, представленное наименьшей неподвижной точкой, достижимо для некоторых конкретных входных данных. Например, полный алгоритм распространения констант смог бы обнаружить каждую константу в программе.

Мы представили некоторые критерии (корректность и полноту), которые всегда можно применять для оценки алгоритма анализа. Из них можно вывести некоторые критерии безопасности относительно устойчивости обфускации (и, двойственным образом, эффективности преобразования деобфускации).

Абстрактную интерпретацию можно использовать для моделирования любого преобразования программы (см. статью Патрика и Радии Кузо[15]). Рассматривая синтаксис программы как абстракцию ее конкретной семантики, мы можем формализовать любое синтаксическое преобразование программы как абстрактную интерпретацию соответствующего семантического преобразования.

Одно из частных применений относится к моделированию преобразований обфускации и деобфускации. Мила Далла Преда и Роберто Джакобацци[18] исследуют семантические преобразования, соответствующие вставке непрозрачных предикатов. Моделируя деобфускацию как абстрактную интерпретацию, они отмечают, что взлом непрозрачных предикатов соответствует наличию полной абстракции. Критерий полноты оказывается особенно интересным для характеристики как эффективности деобфускатора, так и устойчивости непрозрачного предиката.

В заключение, многие методы, уже используемые в анализе программ и компиляции, представляют интерес в контексте обратной разработки. Как было показано ранее, граница между статическим и динамическим анализом не столь очевидна. В настоящее время модель абстрактной интерпретации кажется достаточно общей для применения к обоим типам анализа. Критерии корректности и полноты особенно интересны при моделировании преобразований обфускации и деобфускации в системе абстрактной интерпретации. Вы увидели, что и корректность, и полноту алгоритма можно определить для статического и динамического анализа (анализа потока данных, частичного вычисления, срезов, символьного выполнения), которые хорошо подходят для представления действий специалистов по обратной разработке при попытке упростить представление обфусцированной программы. В модели абстрактной интерпретации статический и

динамический анализ имеют двойственную природу. Эта двойственность и выгода, которую можно получить из синергии статических и динамических методов, открывают новые возможности, подлежащие исследованию в будущем посредством изучения гибридных методов.

В этом разделе были представлены некоторые академические модели и критерии, а также методы динамического и статического анализа, которые могут быть полезны для проектирования и оценки алгоритмов деобфускации. Также была подчеркнута важность гибридных методов. В следующем разделе представлены некоторые доступные в настоящее время инструменты, помогающие отменять преобразования обфускации.

Инструменты деобфускации

В этом разделе обсуждаются некоторые инструменты, которые можно применять для обратной разработки обфусцированного кода, и особенно возможности, облегчающие вашу работу. Обратите внимание: этот список ни в коей мере не претендует на полноту; он основан на опыте некоторых авторов и стремится представить разные категории инструментов.

IDA

IDA - передовой инструмент обратной разработки двоичного кода. Загрузка анализируемого двоичного файла в IDA - обычный рефлекс, поэтому, вероятно, здесь не нужно представлять этот инструмент; в противном случае читатели могут обратиться к книге Криса Игла *The IDA Pro Book* (No Starch Press, 2011). Что касается интересующей нас конкретной темы, работа с обфусцированным кодом в IDA проблематична (хотя и не невозможна) по нескольким причинам:

• Это не основное назначение IDA. Обфусцированный код - очень особый случай, а обработка каждой конкретной ситуации/уловки стала бы бесконечной работой; поэтому лучше не вступать на этот путь.

• У нас очень мало контроля над дизассемблером, что сильно мешает при встрече со схемами обфускации, ломающими/нарушающими/уничтожающими граф потока управления. Дизассемблер IDA очень легко запутать, и часто возникает проблема курицы и яйца: для восстановления потока управления нужно очистить поток данных, но для очистки потока данных нужен поток управления.

• Сама IDA не предлагает никакого промежуточного представления (intermediate representation, IR) или хотя бы семантики команд, поэтому расширенный анализ ее вывода нетривиален.

В 2008 году на семинаре ICAR (презентация) Ильфак Гильфанов предложил несколько полезных советов по использованию конкретных возможностей IDA:

• объединять блоки на уровне графа для упрощения CFG;

• изменять граф динамически по событиям с помощью перехватчиков вроде `grcode_changed_graph` (см. `graph.hpp` в SDK);

• разрабатывать специальные подключаемые модули.

IDA можно расширять с помощью сценариев (IDC или IDAPython) или подключаемых модулей (см. SDK IDA). Именно там можно было бы взаимодействовать с IDA при реализации расширенного анализа.

Для этого были разработаны некоторые подключаемые модули в качестве каркасов деобфускации (например, модуль Optimice Бранко Спасоевича, http://optimice.googlecode.com). Пытаясь решить некоторые из упомянутых ранее проблем, включая семантику команд на основе *x86 Opcode and Instruction Reference* (http://ref.x86asm.net/), этот модуль предлагает сокращение CFG, локальные оптимизации и удаление мертвого кода.

Metasm

Metasm (http://metasm.cr0.org) - каркас с открытым исходным кодом (выпущенный под GNU Lesser GPL v2), разработанный Йоанном Гийо. Он определяет себя как набор средств манипулирования ассемблерным кодом. Написанный на Ruby каркас фактически предоставляет функции кросс-архитектурного ассемблера, дизассемблера, компилятора, компоновщика и отладчика. В настоящее время поддерживаются процессоры Intel x86/x64, MIPS, PPC, Sh4 и ARCompact. Также поддерживается большинство распространенных форматов файлов, включая MZ, PE/COFF, ELF, Mach-O и другие.

Обратные вызовы дизассемблера

Поведение дизассемблера можно динамически изменять с помощью набора экспортируемых обратных вызовов класса `Disassembler`. Два наиболее полезных для деобфускации:

• `callback_newaddr` - вызывается каждый раз, когда обнаруживается путь, который собираются дизассемблировать. В этот момент можно исследовать путь назад или вперед на предмет несообразностей; что особенно важно, можно изменить поведение движка дизассемблирования, удалив ложную передачу управления, сорвав ловушку дизассемблера и т. д.

• `callback_newinstr` - как следует из названия, обратный вызов выполняется при дизассемблировании каждой новой команды.

Семантика команд

Одна из ключевых возможностей каркаса - обратная трассировка (ее можно представить как создание срезов программы). Эта возможность лежит в основе движка дизассемблирования. Она позволяет очень точно восстанавливать поток управления за счет производительности. Построенный поверх этой возможности API каркаса также предлагает метод вычисления семантики базового блока. Однако Metasm не использует строгий промежуточный язык; он опирается на описание семантики каждой команды. Связанный с этим термин в каркасе - `binding`. Metasm разделяет кодирование семантики потока управления и потока данных. Для описания семантики команды используются четыре типа:

• числовое значение;

• символ - все, что не является числовым значением, на основе типа символа Ruby;

• выражение: `Expression[operand1, (operator), (operand2)]` - операндом может быть любой из четырех типов;

• косвенность - косвенное обращение к памяти `Indirection[target, size, origin]`.

Следующий фрагмент познакомит вас с `binding` команд Metasm:

```
# encoding: ASCII-8BIT
#!/usr/bin/env ruby
require "metasm"
include Metasm

# создать код x86
sc = Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOS)
add eax, 0x1234
mov [eax], 0x1234
ret
EOS

dasm = sc.init_disassembler

# дизассемблировать код обработчика
dasm.disassemble(0)

# получить декодированную команду по адресу 0,
# затем ее базовый блок
bb = dasm.di_at(0).block

# вывести дизассемблированный код
puts "\n[+] generated code:"
puts bb.list

# пройти по списку декодированных команд базового блока
bb.list.each{|di|
    puts "\n[+] #{di.instruction}"
    sem = di.backtrace_binding()

    puts " data flow:"
    sem.each{|key, value| puts "  * #{key} => #{value}"}

    # изменяет ли команда указатель команд?
    if di.opcode.props[:setip]
        puts " control flow:"
        # затем вывести семантику потока управления
        puts "  * #{dasm.get_xrefs_x(di)}"
    end
}
```

Для каждого объекта `DecodedInstruction` вызывается метод `backtrace_binding`. Он возвращает хеш. Каждая пара ключ/значение представляет присваивание ключа согласно значению и выражает выходы через входы. Выполнение сценария дает следующий результат:

```
[+] generated code:
0 add eax, 1234h
5 mov dword ptr [eax], 1234h
0bh ret ; endsub entrypoint_0

[+] add eax, 1234h
  data flow:
    * eax => eax+1234h
    * eflag_z => ((eax+1234h)&0ffffffffh)==0
    * eflag_s => (((eax+1234h)>>1fh)&1)!=0
    * eflag_c => ((eax&0ffffffffh)+1234h)>0ffffffffh
    * eflag_o => (((eax>>1fh)&1)==0)&&((((eax>>1fh)&1)!=0)!=
                 ((((eax+1234h)>>1fh)&1)!=0))

[+] mov dword ptr [eax], 1234h
  data flow:
    * dword ptr [eax] => 4660

[+] ret
  data flow:
    * esp => esp+4+0
  control flow:
    * [Indirection[Expression[:esp], 4, 0xb]]
```

Команда RET хорошо показывает различие между потоком данных и потоком управления. Метод get_xrefs_x, предоставляемый объектом дизассемблера, возвращает список (объект Ruby Array) возможных значений указателя команд. Для этой конкретной команды это косвенное обращение, целью которого является регистр ESP, а размер равен 4 (для архитектуры Ia32), то есть dword ptr [ESP]; 0xb - адрес в программе, по которому происходит косвенное обращение.

Обратная трассировка и срезы

До сих пор вы видели, как описывается семантика каждой изолированной команды. Теперь рассмотрим команды внутри потока управления и то, как можно использовать binding команды. Для этого следующий пример демонстрирует типичный шаблон вычисления динамического перехода:

```
# encoding: ASCII-8BIT
#!/usr/bin/env ruby
require "metasm"
include Metasm

# создать код x86 обработчика
sc = Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOS)
entry:
    mov ecx, 1
    shl ecx, 0xA
    add edx, 0xBADC0FFE
    mov eax, 0x100000
    lea eax, [ecx+eax]
    add ecx, 0xBADC0FFE
    jmp eax
EOS

# дизассемблировать код обработчика
dasm = sc.init_disassembler
dasm.disassemble(0)

# получить базовый блок
bb = dasm.block_at(0)
target = dasm.get_xrefs_x(bb.list.last).first
puts "[+] jmp target: #{target}"

# обратная трассировка
values = dasm.backtrace(target, bb.list.last.address,
    {:log => bt_log = [], :include_start => true})
```

get_xrefs_x сообщает, какая цель является последней командой перехода. Затем метод backtrace используется для обратного прохода по потоку управления с отслеживанием зависимостей переменных, пока он не достигнет присваиваний переменных или просто не упрется в свой предел сложности. Каждый шаг обратной трассировки сохраняется в массиве bt_log. Следующие несколько строк позволяют аккуратно вывести журнал:

```
bt_log.each{|entry|
    case type = entry.first
    when :start
        entry, expr, addr = entry
        puts "[start] backtacking expr #{expr} from 0x#{addr.to_s(16)}"

    when :di
        entry, to, from, instr = entry
        puts "[update] instr #{instr},\n  -> update expr from #{from} to
#{to}\n"

    when :found
       entry, final = entry

       puts "[found] possible value: #{final.first}\n"
```

```
    when :up
        entry, to, from, addr_down, addr_up = entry
        puts "[up] addr 0x#{addr_down.to_s(16)} -> 0x#{addr_up.to_s(16)}"
    end
}
```

Вот вывод примера:

```
[+] jmp target: eax
[start] backtacking expr eax from 0x1c
[update] instr 13h lea eax, [ecx+eax],
  -> update expr from eax to ecx+eax
[update] instr 0eh mov eax, 100000h,
  -> update expr from ecx+eax to ecx+100000h
[update] instr 5 shl ecx, 0ah,
  -> update expr from ecx+100000h to (ecx<<0ah)+100000h
[update] instr 0 mov ecx, 1,
  -> update expr from (ecx<<0ah)+100000h to 100400h
[found] possible value: 100400h
```

Движку обратной трассировки удалось пройти назад по потоку команд и вычислить конечное значение отслеживаемого выражения. Движок упрощения позволяет решать (или по крайней мере сокращать) выражения и на символьном, и на числовом уровне.

Из записи журнала можно даже извлечь срез, то есть минимальное подмножество исходной программы, создающее исследуемый эффект (критерий среза). В данном случае срез будет содержать все команды, участвующие в вычислении назначения `JMP`:

```
# Объект DecodedInstruction является 3-м элементом записи :id
slice = bt_log.select{|e| e.first==:di}.map{|e| e[3]}.reverse
puts slice
```

Срез выглядит следующим образом:

```
0 mov ecx, 1
5 shl ecx, 0ah
0eh mov eax, 100000h
13h lea eax, [ecx+eax]
```

Обратите внимание, как несущественные вычисления/присваивания (например, использующие константу `0BADC0FFEh`) были удалены из этого списка.

Этот пример - идеальный случай: выражение можно статически сократить/решить в числовое значение. Теперь представьте, что вы удаляете первую строку ассемблера (`MOV ECX, 1`) - в пределах базового блока `ECX` не определен, - а затем повторяете анализ:

```
[+] jmp target: eax
[start] backtacking expr eax from 0x17
[update] instr 0eh lea eax, [ecx+eax],
  -> update expr from eax to ecx+eax

[update] instr 9 mov eax, 100000h,
  -> update expr from ecx+eax to ecx+100000h
[update] instr 0 shl ecx, 0ah,
  -> update expr from ecx+100000h to (ecx<<0ah)+100000h
```

Возвращаемое значение - объект типа `Expression` со значением `(ECX<<0Ah)+100000h`.

Этот пример довольно тривиален. Возможности движка обратной трассировки значительно шире. Ниже предыдущий пример изменяется так, чтобы включить более сложный граф потока управления:

```
# создать код x86 обработчика
sc = Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOS)
entry:
    mov ecx, 1
    test edx, edx
    jnz label inc cl
label:
    shl ecx, 0xA
    add edx, 0xBADC0FFE
    mov eax, 0x100000
    lea eax, [ecx+eax]
    add ecx, 0xBADC0FFE
    jmp eax
EOS

# дизассемблировать код обработчика
dasm = sc.init_disassembler
dasm.disassemble(0)

# получить последний базовый блок
bblist = dasm.instructionblocks.sort{|b1, b2| b1.address <=> b2.address}
bblist.each{|bb| puts "-\n", bb.list}
bb = bblist.last
```

По сути, здесь вставлена команда (`TEST EDX, EDX`), управляющая условным переходом; в одном случае `ECX` увеличивается, в другом - нет. Обновленный вывод выглядит так:

```
[+] jmp target: eax
[start] backtacking expr eax from 0x21
[update] instr 18h lea eax, [ecx+eax],
  -> update expr from eax to ecx+eax
[update] instr 13h mov eax, 100000h,
  -> update expr from ecx+eax to ecx+100000h
[update] instr 0ah shl ecx, 0ah,
  -> update expr from ecx+100000h to (ecx<<0ah)+100000h
[up]    addr 0xa -> 0x9
[up]    addr 0xa -> 0x7
[update] instr 0 mov ecx, 1,
  -> update expr from (ecx<<0ah)+100000h to 100400h
[found] possible value: 100400h

[update] instr 9 inc ecx,
  -> update expr from (ecx<<0ah)+100000h to ((ecx+1)<<0ah)+100000h
[up]    addr 0x9 -> 0x7
[update] instr 0 mov ecx, 1,
  -> update expr from ((ecx+1)<<0ah)+100000h to 100800h
[found] possible value: 100800h
```

Движок обратной трассировки возвращает массив двух возможных значений: `100400h` или `100800h`. Обратите внимание, как он проследовал за потоком управления по CFG. (Были пройдены обе ветви условного перехода.) Метка `[up]` указывает пересечение базового блока. Обратная трассировка действительно лежит в основе дизассемблера и обеспечивает более точное дизассемблирование. Очевидно, эта возможность влечет серьезные потери производительности (помните о компромиссе между вычислимостью и точностью).

Связывание кода

Вы знаете, как получить семантику изолированной команды, а также как выполнить обратную трассировку значения и вычислить срез для этого конкретного значения. Что, если обобщить этот процесс и вычислить семантику базового блока? Это еще одна очень мощная возможность Metasm: метод `code_binding`, предоставляемый объектом дизассемблера. Он полностью опирается на возможность обратной трассировки. Вот его применение к последнему базовому блоку предыдущего примера:

```
# вычислить семантику базового блока
bbsem = dasm.code_binding(bb.list.first.address, bb.list.last.address)
puts "\n[+] basic block semantics"
bbsem.each{|key, value| puts "    * #{key} => #{value}"}
```

Его вывод выглядит следующим образом:

```
[+] basic block semantics
    * eax => ((ecx<<0ah)+100000h)
    * ecx => ((ecx<<0ah)+badc0ffeh)
    * edx => (edx+badc0ffeh)
```

Miasm

Miasm (http://code.google.com/p/smiasm) - каркас обратной разработки, разработанный Фабрисом Декло и предоставляющий манипулирование PE/ELF, ассемблирование и дизассемблирование (в настоящее время поддерживает Ia32, ARM, PPC и байт-код Java). Как и Metasm, Miasm имеет открытый исходный код и выпущен под GNU Lesser GPL v2, поэтому вы можете углубиться в его движок и настроить его под конкретные потребности. Примеры в этом разделе основаны на последней доступной на момент написания ревизии MIASM (`changeset:270:6ee8e9a58648`).

Каркас опирается на промежуточный язык. Это означает, что семантика большинства распространенных команд закодирована как список выражений. Слово "список" следует понимать в смысле Python (то есть упорядоченное множество объектов).

Грамматика IR Miasm использует девять базовых типов выражений, наиболее важные из которых:

• `ExprInt` - числовое значение;

• `ExprId` - идентификатор/символ, все, что не является числовым значением; например, регистры определяются как `ExprId`;

• `ExprAff` - присваивание `a = b`;

• `ExprCond` - тернарный/условный оператор `a ? b : c`;

• `ExprMem` - косвенное обращение к памяти;

• `ExprOp` - операция `op(a,b,...)`.

Кроме того, он полностью поддерживает срезы (представьте их как объект для представления битовых полей) и композицию срезов. IR допускает символьные вычисления и оснащен движком упрощения выражений.

Для каждого поддерживаемого процессора файл с суффиксом `sem` описывает семантику большинства распространенных команд. Например, вот семантика `ADD`, определенная в `miasm/arch/ia32_sem.py`:

```
def add(info, a, b):
    e= []
    c = ExprOp('+', a, b)
    e+=update_flag_arith(c)
    e+=update_flag_af(c)
    e+=update_flag_add(a, b, c)
    e.append(ExprAff(a, c))
    return e
```

Эта функция строит семантику команды на основе двух ее операндов (`a` и `b`). Эти возможности легко продемонстрировать небольшим сценарием:

```
#! /usr/bin/env python

from miasm.arch.ia32_arch import *
from miasm.tools.emul_helper import *

# ассемблировать команду asm по заданному адресу
def instr_sem(instr, address):
    print "\n[+] instruction %s @ 0x%x" % (instr, address)
    binary = x86_mn.asm(instr)
    di = x86_mn.dis(binary[0])
    semantics = get_instr_expr(di, address)
    for expr in semantics:
        print "  %s" % expr

instr_sem("add eax, 0x1234", 0)
instr_sem("mov [eax], 0x1234", 0)
instr_sem("ret", 0)
instr_sem("je 0x1000", 0)
```

Вот вывод:

```
[+] instruction add eax, 0x1234 @ 0x0
  zf = ((eax + 0x1234) == 0x0)
  nf = ((0x1 == ((eax + 0x1234) >> 0x1F)) & 0x1)
  pf = (parity (eax + 0x1234)) af = (((eax + 0x1234) & 0x10) == 0x10)
  cf = ((0x1 == (((eax ^ 0x1234) ^ (eax + 0x1234)) >> 0x1F)) ^
       (0x1 == ((((eax + 0x1234)) & (! (eax ^ 0x1234))) >> 0x1F)))
  of = (0x1 == (((eax ^ (eax + 0x1234)) & (! (eax ^ 0x1234))) >> 0x1F))
  eax = (eax + 0x1234)

[+] instruction mov [eax], 0x1234 @ 0x0
  @32[eax] = 0x1234

[+] instruction ret @ 0x0
  esp = (esp + (0x4 + 0x0))
  eip = @32[esp]

[+] instruction je 0x1000 @ 0x0
  eip = (zf == 0x1)?(0x1000,0)
```

Семантика команды `ADD` выглядит наиболее сложной из-за обновления флагов. Вот семантика команды `MOV` с явным указанием типа объекта:

```
[+] instruction mov [eax], 0x1234 @ 0x0
  ExprAff( ExprMem(@32[ExprId(eax)]) = ExprInt(0x1234) )
```

Это очень ценная и мощная возможность. Поверх IR построена возможность JIT-компиляции, при которой код сначала дизассемблируется, переводится в IR, а затем повторно генерируется как родной код для выполнения. Документация и примеры показывают случаи использования Miasm для анализа упаковщиков/VM, а также для двоичного инструментирования.

VxStripper

VxStripper - инструмент переписывания двоичных файлов, разработанный Себастьеном Жоссом. Предназначенный для анализа защищенных и потенциально враждебных двоичных программ, он динамически извлекает промежуточное представление двоичного исполняемого файла и все сведения, необходимые для применения определенных упрощений, делая внутреннюю работу двоичного файла более понятной аналитику.

Одна из главных причин выбора вариантов проектирования и реализации этого инструмента - обход текущих ограничений существующих решений для анализа вредоносных и двоичных программ. (Многие инструменты поставляются с собственным промежуточным представлением - неэкспортируемым, иногда проприетарным, - что затрудняет их интеграцию. Более того, многие из них не подходят для анализа враждебного или защищенного кода.) Цель - получить как можно больше сведений из двоичной программы, использующей для защиты этих сведений все доступные методы и инструменты. Идея заключается в ненавязчивом инструментировании виртуального центрального процессора и гостевой операционной системы, чтобы динамически получать сведения, необходимые для восстановления программы и упрощения ее представления. Этот инструмент основан на движке динамической двоичной трансляции QEMU и цепочке компиляции LLVM.

LLVM (Low Level Virtual Machine) - цепочка компиляции с внушительным набором оптимизаций, которые можно применять на всем протяжении жизни программы. LLVM использует строго типизированный RISC-подобный набор команд и представление статического однократного присваивания (static single assignment, SSA; в таком представлении каждой временной переменной значение присваивается только один раз). LLVM включает множество двоичных серверных частей (x86, x86-64, SPARC, PowerPC, ARM, MIPS, CellSPU, XCore, MSP430, MicroBlaze, PTX) и несколько серверных частей для исходного кода (C, C++). Дополнительные сведения об LLVM приведены в статье Латтнера[31].

Динамический двоичный транслятор (Dynamic Binary Translator, DBT) QEMU (Quick EMUlator) применяется для динамической трансляции двоичного кода архитектуры гостевого процессора в архитектуру процессора узла с помощью IR под названием TCG (Tiny Code Generator). Этот язык состоит из простых RISC-подобных команд, называемых микрооперациями. Двоичная трансляция состоит из двух этапов. Сначала гостевой двоичный код переводится в последовательности команд TCG, называемые блоками трансляции (внешний интерфейс DBT). Затем блоки трансляции преобразуются в код, исполняемый процессором узла (серверная часть DBT). DBT QEMU имеет множество двоичных внешних интерфейсов (x86, x86-64, ARM, ETRAX CRIS, MIPS, Micro Blaze, PowerPC, SH4 и SPARC). Дополнительные сведения о QEMU приведены в статье Беллара[4].

VxStripper наследует от QEMU множество двоичных внешних интерфейсов, а от LLVM - множество серверных частей, предоставляя за разумную цену полный каркас переписывания двоичных файлов. Функции переписывания реализованы как проходы LLVM.

Его текущая конструкция опирается на уже выполненную работу по преобразованию TCG IR в LLVM IR (LLVM-QEMU, описанный Шеллером[36], и S2E, описанный Чипуновым[8]), а также на алгоритмы проектирования, представленные Жоссом[27][28].

Одна из целей этого инструмента - взаимодействие с многочисленными средствами анализа программ, основанными на цепочке компиляции LLVM, через "экспортированное" представление вредоносной программы. Этот инструмент двоичного анализа специально предназначен для решения проблемы анализа враждебных программ. Цель - автоматизировать зачастую утомительные и повторяющиеся задачи аналитика.

Эта цепочка компиляции основана на модульной и развивающейся архитектуре, позволяющей применять одни и те же преобразования к широкому разнообразию программных и аппаратных архитектур. Она основана на современной цепочке компиляции, предоставляющей эффективное промежуточное представление и функциональность.

Vellvm (Verified LLVM), описанный Чжао[43], предоставляет формальные инструменты для рассуждений о преобразованиях, работающих с промежуточным представлением LLVM. Vellvm можно использовать для извлечения формально проверенных реализаций проходов деобфускации, реализованных в VxStripper.

Расширение DBT QEMU

Вы увидели, что движок DBT QEMU выполняет динамическую трансляцию двоичного кода из архитектуры гостевого процессора в архитектуру процессора узла с помощью промежуточного представления TCG.

Следующая команда служит простым примером того, как выглядит этот язык:

```
0x0040104c:  push   0xa
```

Предыдущая команда переводится в представлении QEMU TCG следующим образом:

```
(1)    movi_i32 tmp0,$0xa
(2)    mov_i32 tmp2,esp
(3)    movi_i32 tmp13,$0xfffffffc
(4)    add_i32 tmp2,tmp2,tmp13
(5)    qemu_st32 tmp0,tmp2,$0x1
(6)    mov_i32 esp,tmp2
(7)    movi_i32 tmp4,$0x40104e
(8)    st_i32 tmp4,env,$0x30
(9)    exit_tb $0x0
```

Этот блок команд TCG эмулирует выполнение команды `push` на программном процессоре. Выполняются следующие операции: целое число `0xa` сохраняется в переменной `tmp0` (строка 1). Затем эта переменная сохраняется в стеке (строки 2-6). Адрес команды, следующей за текущей, сохраняется в `tmp4` (строка 7), а затем в регистре VPU QEMU `cc_op`. Последняя команда (строка 9) обозначает конец блока TCG.

Инструмент изменяет механизм DBT таким образом, чтобы функция инструментирования виртуального процессора систематически вызывалась перед выполнением блока трансляции. Для этого добавляется дополнительная микрооперация (`op_callback`), принимающая в качестве операнда адрес функции инструментирования (`vpu_callback`). Результирующий код TCG выглядит так:

```
(1)    op_callback @vpu_callback
(2     movi_i32 tmp0,$0xa
(3)    mov_i32 tmp2,esp
(4)    movi_i32 tmp13,$0xfffffffc
(5)    add_i32 tmp2,tmp2,tmp13
(6)    qemu_st32 tmp0,tmp2,$0x1
(7)    mov_i32 esp,tmp2
(8)    movi_i32 tmp4,$0x40104e
(9)    st_i32 tmp4,env,$0x30
(10)   exit_tb $0x0
```

Этот механизм позволяет выполнять код инструментирования на каждом цикле выполнения виртуального процессора. Имея доступ к регистрам VPU и памяти виртуального ПК, можно получить контекст процесса и извлечь сведения о его взаимодействиях с гостевой операционной системой.

Инструментируя также команды загрузки и сохранения TCG, можно извлечь сведения о взаимодействиях целевого процесса с памятью гостевой системы. Благодаря этим сведениям можно восстановить информацию о перемещениях процесса.

Теперь, когда вы увидели, как изменить виртуальный процессор QEMU, чтобы обеспечить систематический вызов функции инструментирования, рассмотрим преобразование промежуточного представления TCG в представление LLVM. Результат преобразования предыдущего блока TCG выглядит так:

```
(1)  %esp_v.i = load i32* @esp_ptr
(2)  %tmp2_v.i = add i32 %esp_v.i, -4
(3)  %4 = inttoptr i32 %tmp2_v.i to i32*
(4)  store i32 10, i32* %4
(5)  store i32 %tmp2_v.i, i32* @esp_ptr
(6)  store i32 4198478, i32* %next.i
(7)  store i32 0, i32* %ret.i
```

Целое число `0xa` сохраняется по адресу, на который указывает переменная `%4`, что эквивалентно сохранению его в стеке (строки 1-4). Адрес команды, следующей за текущей, сохраняется в переменной `%next.i` (строка 6). Последняя команда (строка 7) завершает блок LLVM.

После процесса нормализации этот блок LLVM компилируется в следующий ассемблерный код:

```
401269!  mov dword ptr [esp-14h], 0ah
```

Теперь у вас есть общее представление об основных изменениях, внесенных в эмулятор QEMU и схематично показанных на рис. 5-2. В следующем разделе описывается общая архитектура инструмента.

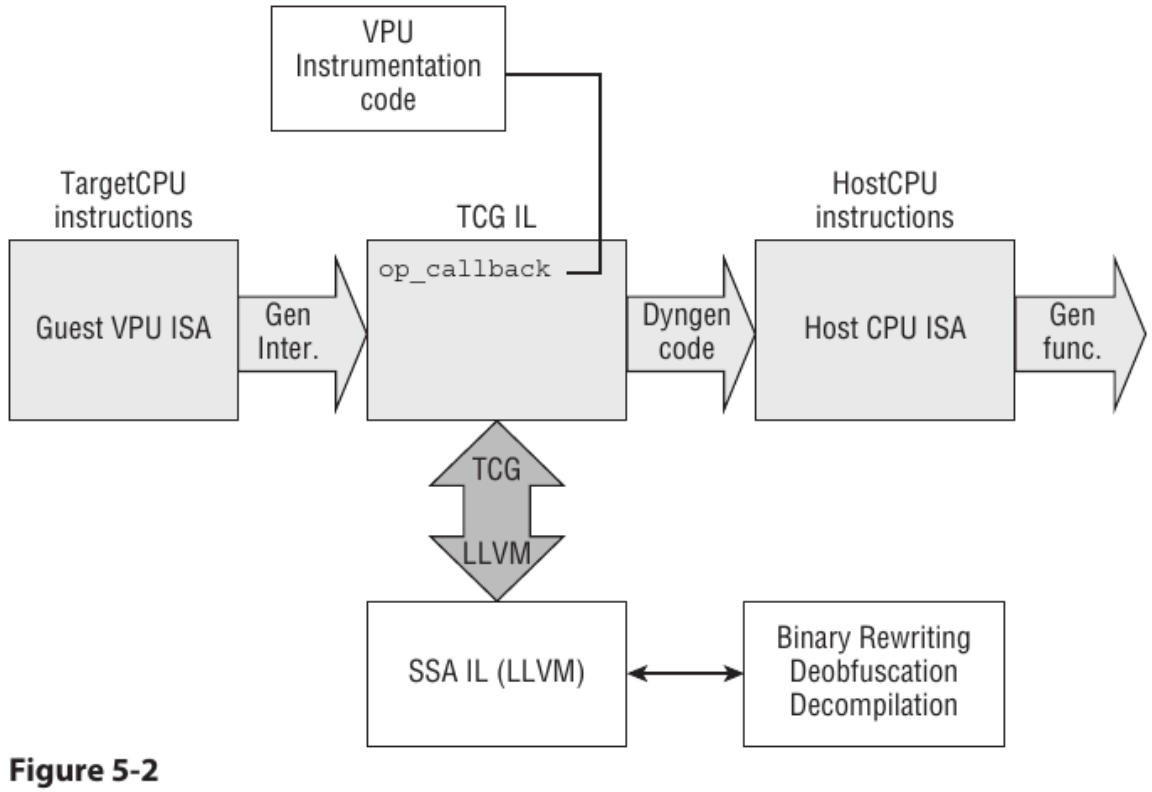

Figure 5-2

Рисунок 5-2. Схема преобразования кода в VxStripper

Архитектура VxStripper

VxStripper реализует расширенный движок DBT и несколько специализированных функций анализа (см. рис. 5-3) для наблюдения за целевой программой и средой ее выполнения.

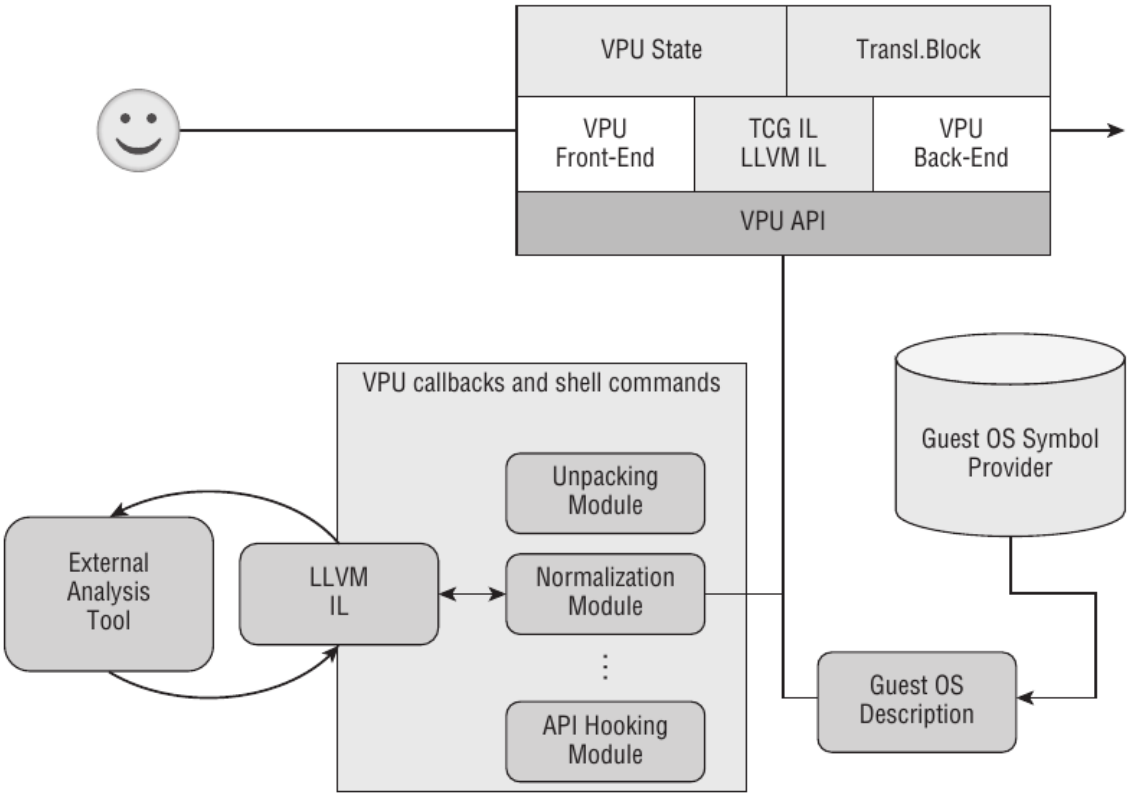

Figure 5-3

Рисунок 5-3. Архитектура VxStripper

Диспетчер модулей управляет активацией этих функций анализа, реализованных как подключаемые модули, и их совместной работой.

Эти функции анализа извлекают семантические сведения из целевой программы. Это могут быть трасса ее взаимодействий с API гостевой операционной системы, способ ее работы с объектами и структурами исполнительной системы или ядра гостевой операционной системы либо, проще говоря, трасса ее машинного кода.

Извлечение этих сведений опирается на описание гостевой операционной системы, которое может предоставляться, например, сервером символов, как в случае семейства операционных систем Windows.

Среди уже реализованных модулей, в частности, имеются:

• модуль перехвата API;

• модуль криминалистического анализа;

• модуль распаковки;

• модуль нормализации.

Перехват API

Модуль перехвата Native API и Windows API в VxStripper основан на криминалистическом анализе памяти гостевой операционной системы без какого-либо взаимодействия с гостевой операционной системой.

Исполнительная система Windows поддерживает набор структур, содержащих сведения о модулях, загруженных для данного процесса в его адресное пространство. Эти структуры данных можно восстановить с помощью блока окружения процесса (process environment block, PEB), к которому, в свою очередь, можно обратиться по смещению относительно сегментного регистра `FS`. Модули перехвата Native API и Windows API в VxStripper используют эти сведения, чтобы находить и инструментировать Windows API.

Криминалистический анализ и анализ руткитов

Модуль криминалистического анализа VxStripper предоставляет дополнительные возможности для наблюдения и проверки целостности множества мест в гостевой платформе, где может быть установлен перехватчик. Он обходит структуры исполнительной системы операционной системы, чтобы выявлять потенциальные цели атаки руткита, и отслеживает аппаратные компоненты, которые могут быть повреждены руткитом. Эти сведения крайне важны для понимания аналитиком низкоуровневых вирусных атак.

Для целей этой главы можно считать, что эти возможности подобны ожидаемым от отладчика ядра. Можно подключиться к процессу, просматривать состояние его ЦП и дизассемблированный код, а также трассировать взаимодействие целевой программы с API операционной системы. Такое исследование выполняется в безопасной и контролируемой среде, без какого-либо навязчивого взаимодействия с гостевой операционной системой.

В следующих двух разделах подробнее рассматривается работа двух важнейших аналитических модулей Vxstripper: модуля распаковки и модуля нормализации.

Модуль распаковки

Модуль распаковки находит исходную точку входа (original entry point, OEP) целевого исполняемого файла, получает сведения о его взаимодействиях с API операционной системы и извлекает информацию о перемещениях.

В основе лежит простая проверка целостности исполняемого кода целевой программы: для каждого блока трансляции программы сравнивается его значение в виртуальной памяти со значением в файловой системе узла. Пока значения совпадают, ничего не делается. Как только обнаруживается различие, текущий блок трансляции записывается в необработанный файл вместо старого блока трансляции. Первая команда вновь созданного блока трансляции определяется как OEP защищенной программы. В конце анализа разделы данных записываются в необработанный файл вместо исходных разделов данных.

Один и тот же алгоритм наблюдения применяется к каждому блоку трансляции. Загрузчик защиты упакованного исполняемого файла может иметь несколько слоев расшифровки. Как только достигнут последний расшифрованный блок трансляции, остается лишь восстановить целевой исполняемый файл. Чтобы восстановить структуру PE (Portable Executable) незащищенного исполняемого файла, необходимо выполнить несколько задач: задать исходную точку входа, перестроить таблицы импорта и перемещений и проверить согласованность заголовка PE.

Метод, который движок распаковки использует для восстановления таблицы адресов импорта (Import Address Table, IAT) и перемещений, основан на перехвате Win32 API и Native API. Во время распаковки трассируются все вызовы API. При загрузке инициализируется отсортированная таблица вызовов API посредством обхода структур исполнительной системы NT.

Затем, после возобновления выполнения процесса, трассируется каждый вызов API. Эта таблица регулярно обновляется во время выполнения целевого процесса и используется для динамического разрешения имен функций API. Наконец, после завершения дампа адресного пространства целевого процесса эта таблица используется для исправления IAT в исполняемом файле PE.

Благодаря инструментированию команд загрузки и сохранения TCG можно динамически извлечь информацию о перемещениях программы, которую также можно добавить в новый раздел исполняемого файла.

Например, ниже приведены (полезные) сведения, извлеченные на этапе распаковки программы, отображающей диалоговое окно (функция `MessageBoxA`):

```
[INFO] eip=0x00401000
[RELOC] value=0x00403000 va=0x00401003
[RELOC] value=0x0040300f va=0x00401008
[RELOC] value=0x00402008 va=0x00401010
[APICALL] api_pc=0x77d8050b api_oep=0x77d8050b
          dll_name=C:\WINDOWS\system32\user32.dll
          func_name=MessageBoxA
          value=0x00402008 va=0x00401010
```

Информация о перемещениях состоит из пар `(va, value)`, содержащих соответственно виртуальный адрес и перемещаемое значение. Обратите внимание: для этого упаковщика пролог функции `MessageBoxA` защитой не эмулируется. В противном случае фактически вызываемый внешний адрес (`api_pc`) отличается от точки входа функции API (`api_oep`).

Нормализация

В большинстве случаев после этапа распаковки можно (автоматически) получить двоичный файл, очищенный от загрузчика защиты и не содержащий никакого перезаписываемого кода. К сожалению, теперь для полного понимания внутренней работы вредоносной программы необходимо обработать некоторые механизмы обфускации (уплощение потока управления, преобразования обфускации на основе VM и т. д.).

Первая попытка решить эти проблемы была реализована в VxStripper с помощью промежуточного представления LLVM. Вместо того чтобы пытаться работать с двоичным файлом после создания дампа его образа памяти, предлагается работать с его промежуточным представлением и увеличивать объем сведений (собранных динамически), внедряя их в модуль LLVM. Такое представление лучше подходит для дальнейшего анализа.

Модуль нормализации использует результаты предыдущих анализов для создания представления LLVM блоков трансляции, к которому применяется несколько оптимизирующих преобразований. Если рассмотреть это подробнее, во время выполнения целевой программы серверная часть LLVM для QEMU TCG выдает представление LLVM преобразованных блоков. Этот код LLVM связывается с инициализационным модулем LLVM (см. рис. 5-4).

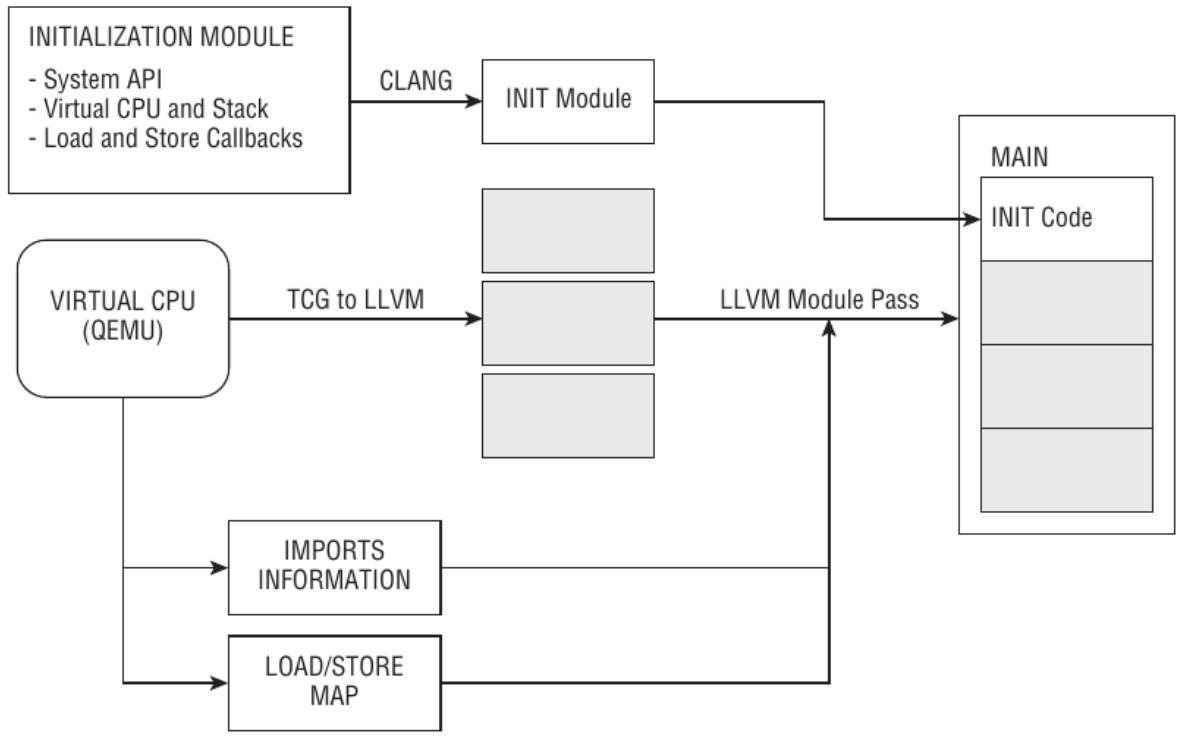

Figure 5-4

Рисунок 5-4. Формирование модуля LLVM при нормализации

Этот инициализационный модуль реализует функции обратного вызова для загрузки и сохранения, объявляет прототипы системных API и настраивает виртуальный процессор и его стек.

Модуль нормализации использует сведения, динамически собранные во время выполнения целевой программы, чтобы разрешать импорт, обрабатывать перемещения и извлекать разделы данных. Сведения таблицы импорта используются для построения команд вызова API в LLVM. Карта обращений к памяти при загрузке и сохранении используется для применения перемещений и внедрения данных целевой программы в модуль LLVM.

После перестроения модуля LLVM к его представлению применяются некоторые дополнительные проходы оптимизации. Затем LLVM можно скомпилировать для выбранной архитектуры с помощью одной из доступных серверных частей LLVM. Его также можно преобразовать в код C или C++.

Первые результаты показывают, что стандартной оптимизации совместно с частичным вычислением, обусловленным динамическим преобразованием целевого кода в его представление LLVM, достаточно для радикального сокращения и упрощения анализируемого кода.

Заключительные соображения

В этом разделе обсуждались лишь четыре инструмента (причем с уклоном в статический анализ). Наряду с каркасами Metasm и Miasm можно было бы, например, упомянуть каркас Radare (http://www.radare.org/y/). Заслуживают внимания и усилия Рольфа Роллеса по расширению IDA с помощью его интерпретатора idaocaml (https://code.google.com/p/idaocaml/). Существует множество других инструментов, которые здесь не упоминались или упоминались лишь кратко, и мы призываем вас опробовать их самостоятельно.

Если взглянуть на эти инструменты в перспективе, IDA - хороший дизассемблер, однако при работе с обфусцированным кодом она мало чем может помочь. Каркасы Metasm и Miasm идут на шаг дальше, предоставляя больше контроля, IR для экспериментов и так далее. Такие инструменты, как VxStripper, идут еще дальше. Вероятно, вы уже ощущаете это: идет гонка вооружений. На разработку обфускаторов затрачиваются огромные усилия, поэтому и наши инструменты должны развиваться.

Для специалиста по обратной разработке создание инструментов - это вложение, которое он делает ради достижения своих целей и от которого ожидает определенной отдачи. Создание наиболее совершенных инструментов может занять недели, если не месяцы, и требует обширных знаний.

Практическая деобфускация

Теперь вы увидите, как некоторые из представленных ранее инструментов можно использовать для практической деобфускации. И вновь следующие разделы не претендуют на исчерпывающую полноту. Их цель - показать несколько распространенных случаев применения методов деобфускации.

Деобфускация на основе шаблонов

Возможно, это простейшая и наименее затратная деобфускация: она работает на синтаксическом уровне и сопоставляет известные шаблоны. Не забывайте, что ранние шаблоны обфускации в основном создавались вручную и защищали код (например, код некоторых упаковщиков), а набор их шаблонов был ограничен; поэтому перечислить их все было "приемлемо".

Этот метод деобфускации сводится к алгоритму поиска и замены на уровне двоичного кода (кодов операций), возможно с использованием поиска по подстановочным знакам. Главный недостаток заключается в том, что в результате остается двоичный файл, усеянный командами `NOP`.

Чтобы проиллюстрировать это, рассмотрим следующий старый сценарий OllyDbg. Упаковщики - классический пример программ, использующих методы обфускации (в данном случае для защиты своих заглушек). На протяжении многих лет OllyDbg был (и, вероятно, остается) любимым инструментом распаковки, и для помощи в этой задаче было выпущено множество сценариев. Этот (случайно выбранный) старый сценарий (2004 год, автор loveboom, http://tuts4you.com/download.php?view.601) предназначен для версий ASProtect 2.0x (широко применявшегося в то время упаковщика). Он использует команды `REPL` языка OllyScript для поиска и замены набора шаблонов. Определение `REPL` выглядит так:

```
repl addr, find, repl, len
Replace find with repl starting att addr for len bytes.
```

Все шаблоны основаны на одном и том же приеме: безусловный переход внутрь следующей за ним команды используется для запутывания дизассемблеров. Разнообразие немного увеличивается с помощью префиксов команд (таких как `REP` или `REPNE`). Команда `repl` используется для замены этих шаблонов командами `NOP`:

```
repl eip,#2EEB01??#,#90909090#,1000
repl eip,#65EB01??#,#90909090#,1000
repl eip,#F2EB01??#,#90909090#,1000
repl eip,#F3EB01??#,#90909090#,1000
repl eip,#EB01??#,#909090#,1000
repl eip,#26EB02????#,#9090909090#,1000
repl eip,#3EEB02????#,#9090909090#,1000
```

Здесь мы работаем только на синтаксическом уровне. Для цели с ограниченным набором шаблонов этот метод деобфускации тривиален, но эффективен:

• затраты на применение ограниченны, если вообще не пренебрежимо малы;

• затраты на разработку также почти равны нулю.

Разумеется, как и в случае сигнатур вирусов и антивирусных движков, полиморфизм и разнообразие делают его бесполезным. Эквивалентный сценарий можно было бы разработать с использованием возможностей сценариев IDA. Именно такой сценарий можно создать при анализе тривиально обфусцированной вредоносной программы и/или упаковщика, когда скорость анализа имеет приоритет.

Деобфускация на основе анализа программ

Теперь рассмотрим следующий образец обфусцированного кода (это лишь очень небольшой фрагмент; обфусцированный код продолжается таким образом на протяжении тысяч команд):

```
.text:00405900  loc_405900:
.text:00405900      add edx, 67E37DA7h
.text:00405906      push    esi
.text:00405907      mov esi, 0D0B763Ah
.text:0040590C      push    eax
.text:0040590D      mov eax, 15983FC8h
.text:00405912      neg eax
.text:00405914      inc eax
.text:00405915      inc eax
.text:00405916      jmp loc_4082AD
.text:004082AD  loc_4082AD:
.text:004082AD      not eax
.text:004082AF      and eax, 1D48516Ch
.text:004082B4      sub eax, 0ACE1B37Ah

.text:004082B9      xor esi, eax
.text:004082BB      pop eax
.text:004082BC      xor edx, esi
.text:004082BE      pop esi
.text:004082BF      and ecx, edx
.text:004082C1  jmp loc_407C54
```

Вероятно, по первым разделам этой главы вы узнаете некоторые использованные здесь методы обфускации, особенно развертывание констант. Обратите также внимание, что для разделения кода на базовые блоки вставляются безусловные переходы, после чего блоки распределяются по двоичному файлу (их порядок изменяется случайным образом). Очевидного шаблона нет - по крайней мере такого, который можно было бы эффективно сопоставлять на синтаксическом уровне с помощью сигнатур.

Необходимо перейти на семантический уровень. Опираясь на вывод дизассемблера, можно начать с потока управления, объединив базовые блоки `405900h` и `4082ADh`. Затем можно заняться потоком данных, рассмотрев две команды:

```
.text:0040590D    mov eax, 15983FC8h
.text:00405912    neg eax
```

Исходя из семантики этих команд, вы знаете, что регистру `EAX` сначала присваивается постоянное значение, а затем это значение меняет знак. Команду `NEG` можно вычислить заранее и переписать фрагмент в более простой форме, присвоив `EAX` значение с уже измененным знаком:

```
.text:0040590D    mov eax, EA67C038h
```

Переписывание программ в более простой форме, предварительное вычисление значений, не зависящих от входных данных программы, удаление бесполезного кода - все это оптимизация программ, которой компиляторы занимаются почти с момента своего появления. Эти методы можно адаптировать и повторно использовать для собственных целей; по данной теме имеется обширная литература. К классическим методам оптимизации компилятора относятся:

• локальная оптимизация;

• свертывание и распространение констант;

• устранение мертвых сохранений;

• свертывание операций;

• устранение мертвого кода;

• и т. д.

Именно такой подход предлагает подключаемый модуль деобфускации Optimice для IDA. Ранее также были представлены работы Газе и Гийо[22] и Жосса[29] - соответственно в виде подключаемого модуля Metasm и подключаемого модуля VxStripper. Идея состоит в нормализации кода для получения сокращенной, оптимизированной, канонической формы, которая проще для анализа и ближе к исходному незащищенному коду.

Эти попытки предоставить каркасы и утилиты деобфускации все еще далеки от совершенства. Кроме того, рядовому специалисту по обратной разработке доступно не так много инструментов для противодействия обфусцированным программам (разумеется, кое-где существуют закрытые совершенные инструменты). Вероятно, будущее деобфускации примет форму развитых инструментов и аналитических платформ, основанных на формальном IR. Рольф Роллес представил некоторые результаты работы собственного каркаса, написанного на OCaml[34]. Среди других популярных каркасов, которые можно было бы использовать, - основанный на LLVM SecondWrite (Смитсон и др., 2007[39]), S2e/revgen (Чипунов, 2001[7]) и BitBlaze (Сонг и др., 2008[40], а также их последующие работы). Усилия в области деобфускации должны будут соответствовать усилиям, вкладываемым в обфускацию.

Сложный анализ

В этом разделе обсуждаются два наиболее действенных метода обфускации: виртуализация кода и уплощение кода. Для них совершенно очевидно требуется работать на семантическом уровне.

Простые реализации VM

Существуют главным образом две формы реализации VM. Самая простая форма - разработка простого эмулятора процессора. С точки зрения алгоритма он включал бы следующие шаги:

1. Цикл:

a. Выборка - прочитать поток байт-кода по указателю команд.

b. Декодирование - декодировать код операции команды и ее операнды.

c. Выполнение - вызвать соответствующий обработчик кода операции.

2. Обновить указатель команд или выйти из цикла.

В такой конфигурации каждый обработчик команды отвечает за обновление контекста VM. Контекст представляет лежащую в основе эмулируемую архитектуру. Вероятно, он состоит из набора регистров и, возможно, области памяти. Каждый обработчик реализует отдельную команду эмулируемого процессора (один обработчик для `ADD`, один для `SUB` и т. д.).

Такая форма часто используется более простыми реализациями. Обработчики совершенно независимы друг от друга, а указатель команд увеличивается на размер команды (кроме команд, которые непосредственно его изменяют).

С точки зрения атакующего этот тип реализации легко распознается. Ниже подробно перечислены шаги, обычно необходимые для анализа VM:

1. Понять, как команда декодируется из необработанного байт-кода: какая часть кодирует операцию (номер обработчика), какая - операнд или операнды и так далее.

2. Вывести архитектуру VM из операндов команд: количество регистров, компоновку памяти, интерфейсы ввода-вывода и т. д.
3. Выполнить анализ обработчиков. Когда способ декодирования операндов известен, можно рассмотреть, как каждый обработчик манипулирует различными операндами, которые он может принимать в качестве аргументов. Этот шаг составляет суть анализа VM: каждому обработчику сопоставляется собственная семантика.

Обладая всеми этими сведениями, наконец можно построить подобный дизассемблеру инструмент, позволяющий дизассемблировать байт-код VM.

В 2006 году Максимус опубликовал две замечательные статьи об обратной разработке VM: "Reversing a simple virtual machine"[32] и "Virtual machines re-building"[33]. Одна из использованных им целей (HyperUnpackme2) в том же году была также подробно рассмотрена Рольфом Роллесом[35].

Хотя это полезные работы талантливых специалистов по обратной разработке, анализ VM остается в некоторой степени ручной и повторяющейся задачей: для каждого нового экземпляра VM приходится разрабатывать новый дизассемблер. Кроме того, авторы защит также отреагировали, усилив свои реализации VM.

Совершенные реализации VM

Более совершенные реализации VM происходят от простого типа, но добавляют важные возможности, усиливающие реализации и повышающие их устойчивость к анализу:

• Развертывание цикла - этот классический метод оптимизации компилятора отдает предпочтение времени (скорости) в компромиссе программы между объемом и временем. Он заменяет структуру цикла последовательными вызовами тела цикла, которое тем самым разворачивается. В применении к VM каждый обработчик отвечает за выборку и декодирование собственных операндов, после чего соответствующим образом обновляет контекст.

• Уплощение кода - главный цикл выполнения VM уплощается. Это означает, что каждый обработчик отвечает за обновление указателя команд (указателя на байт-код). В действительности уплощение кода и обфускация на основе VM по существу являются одним и тем же. Уплощение кода виртуализирует или перенаправляет только поток управления защищенного кода, тогда как обфускация виртуальной машиной виртуализирует или перенаправляет и поток управления, и поток данных. Для отслеживания контекста диспетчера уплощенного кода и контекста VM можно использовать почти одни и те же алгоритмы.

• Кодирование и шифрование байт-кода - каждый вызов VM зависит от ключа шифрования, передаваемого VM как часть инициализации ее контекста. Каждый обработчик обновляет этот ключ, в результате чего получается изменяющийся ключ. Обработчики зависят от изменяющегося ключа при декодировании своих операндов из закодированного байт-кода. Атакующий не может начать анализ VM в выбранной точке, поскольку значение ключа в этой точке было бы неизвестно.

• Обфускация кода - машинный код VM обфусцируется с помощью методов, подобных описанному в начале этой главы. Простое изучение кода обработчика ничего не говорит о его семантике.

Итак, усиленные реализации VM на порядок труднее анализировать статически. Для каждого состояния VM атакующий должен знать по меньшей мере следующие обязательные значения:

• указатель байт-кода;

• указатель команд, то есть значение следующего обработчика, который будет выполнен;

• изменяющийся ключ шифрования.

VM вышли на новый уровень сложности, поэтому требуется и новый уровень атаки. Анализировать обработчики стало сложнее; более того, их нельзя анализировать изолированно (например, значение ключа шифрования было бы неизвестно). Атакующий стремится свести ручной анализ к минимуму. Тем не менее чаще всего требуется ручное исследование, чтобы "уловить" общее поведение VM и тем самым вывести возможные способы ее атаки.

Одно из первых интересующих мест - заглушка вызова VM, то есть переход между машинным (невиртуализированным) кодом и VM/интерпретатором. Инициализация контекста показывает характер отображения между машинной архитектурой и архитектурой VM. Это может быть точная копия машинных регистров в регистры VM или нечто более сложное. Кроме того, в этой точке полезно по возможности различать обязательные переменные инициализации (например, ключ VM, номер обработчика или точку входа), для которых необходимо числовое значение, и дополнительные переменные, которые можно оставить символическими.

Второе важнейшее место - диспетчер VM (если он существует). Чаще всего имеется единственная точка диспетчеризации, которая в основном извлекает следующий обработчик из таблицы обработчиков по индексу, хранящемуся где-либо в контексте VM. Необходимо ответить на вопрос: каково условие прерывания этого цикла выполнения? В такой конфигурации VM можно рассматривать как обобщение уплощения кода. Уплощение кода виртуализирует только поток управления, тогда как VM виртуализирует и поток управления, и поток данных. Кроме того,

уплощение кода чаще всего действует лишь на уровне одной функции, тогда как VM действует на уровне программы. Другая возможность - распределенная диспетчеризация, при которой каждый обработчик отвечает за обновление указателя команд VM и связывание со следующим обработчиком.

Это общие идеи; у каждого специалиста по обратной разработке есть собственные приемы и абстракции этой задачи.

Использование Metasm

Подход, который мы собираемся исследовать, основан на каркасе Metasm. Он опирается на символьное выполнение, заставляющее VM (то есть интерпретатор) обрабатывать байт-код (относительно статических данных) и вычислять остаточную программу. С одной стороны имеется программа (интерпретатор), с другой - ее статические данные (байт-код); мы специализируем программу относительно ее статических данных.

Рассматривать обфусцированную программу и ее данные как единое целое было бы слишком сложно, поэтому сложную задачу следует разбить на несколько более простых подзадач. Уровень обработчиков команд виртуальной машины дает гораздо более подходящую гранулярность. Отныне мы будем считать обработчики команд наименьшей единицей семантики интерпретатора.

На основе этой исходной предпосылки можно применить следующий псевдоалгоритм:

1. Зафиксировать текущий контекст (байт-код VM, статические параметры, известное окружение и т. д.).

2. Дизассемблировать текущий обработчик.

3. При необходимости деобфусцировать код.

4. Вычислить его семантику (то есть передаточную функцию).

5. Сформировать вывод из разрешенной семантики.

6. Вычислить следующее состояние (то есть применить передаточную функцию к текущему контексту).

7. Если диспетчер обработчика не достиг условия прерывания или выхода, повторить с шага 1.

При наличии возможности из передаточной функции, вычисленной на шаге 4, можно дополнительно восстановить машинный код. Как сформулировал Футамура[21], если имеется интерпретатор языка `L_interpreted`, написанный на некотором машинном языке `L_native`, можно автоматически получить компилятор из `L_interpreted` в `L_native`.

С теоретическими понятиями все хорошо. Теперь предположим, что перед вами экземпляр VM. С чего начать? Рассмотрим практический пример.

Следующий сценарий использует Metasm, чтобы скомпилировать, а затем дизассемблировать то, что могло бы быть обработчиком VM. Для простоты в этом примере рассматривается лишь часть VM (поэтому код не обфусцирован). Код обработчика расположен по адресу 10000000h, а раздел данных, содержащий байт-код обработчика, - по адресу 1a000000h:

```
# encoding: ASCII-8BIT
#!/usr/bin/env ruby

require "metasm"
include Metasm

$SPAWN_GUI = false
CODE_BASE_ADDR = 0x10000000
HTABLE_BASE_ADDR = 0x18000000
DATA_BASE_ADDR = 0x1A000000
INJECT_MAX_ITER = 0x20

NATIVE_REGS = [:eax, :edx, :ecx, :ebx, :esp, :ebp, :esi, :edi]

def display(bd)
  bd.each{|key,value| puts "  #{Expression[key]} => #{Expression[value]}"}
end

# создать код x86 обработчика
sc = Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOS)
lodsd
mov ecx, eax
xor ecx, ebp
movzx eax, cl
push eax
mov eax, [edi+eax]

movzx edx, ch
mov edx, [edi+edx]
xor eax, edx

pop edx
mov [edi+edx], eax

lodsd xor ebp, 0x35ef6a14
xor eax, ebp
jmp [#{HTABLE_BASE_ADDR}+eax*4]
EOS

handler = sc.encode_string

# раздел данных в шестнадцатеричном виде
data_section_hex = "\xA3\xCB\xDB\x5F\x60\xBD\x34\x6A"

# добавить раздел кода
dasm = sc.init_disassembler
dasm.add_section(EncodedData.new(handler), CODE_BASE_ADDR)

# добавить раздел данных
dasm.add_section(EncodedData.new(data_section_hex), DATA_BASE_ADDR)

# дизассемблировать код обработчика
dasm.disassemble_fast_deep(CODE_BASE_ADDR)
```

Первое, что нужно сделать, - автоматически получить семантику этого обработчика. Как упоминалось ранее, Metasm предоставляет метод `code_binding`, который вычисляет передаточную функцию (в терминологии Metasm - связывание) набора команд. Таким образом, можно написать следующее:

```
# вычислить семантику обработчика
bb = dasm.di_at(CODE_BASE_ADDR).block
start_addr = bb.list.first.address
end_addr = bb.list.last.address
puts "[+] from 0x#{start_addr.to_s(16)}, to 0x#{end_addr.to_s(16)}"
binding = dasm.code_binding(start_addr, end_addr)
display(binding)
```

Предыдущий код выдает следующий результат:

```
[+] from 0x10000000, to address 10000021
  dword ptr [esp] => (dword ptr [esi]^ebp)&0ffh
  dword ptr [edi+((dword ptr [esi]^ebp)&0ffh)] =>

    dword ptr [edi+((dword ptr[esi]^ebp)&0ffh)] ^
    dword ptr [edi+(((dword ptr[esi]>>8)^(ebp>>8))&0ffh)]
  eax => (dword ptr [esi+4]^(ebp^35ef6a14h))&0ffffffffh
  ecx => (dword ptr [esi]^ebp)&0ffffffffh
  edx => (dword ptr [esi]^ebp)&0ffh
  ebp => (ebp^35ef6a14h)&0ffffffffh
  esi => (esi+8)&0ffffffffh
```

И по ассемблерному коду, и по связыванию о VM можно сказать следующее:

• по-видимому, она использует изменяющийся ключ, хранящийся в `EBP`. Обратите внимание, как он используется для расшифровки операндов команды из байт-кода;

• по-видимому, на ее контекст указывает `EDI`.

Это хорошее начало, но до цели еще далеко. Мы все еще остаемся на уровне ассемблерного кода, поэтому сделаем шаг назад и рассмотрим инициализацию VM, которую мы определили следующим образом:

```
pushad
pop [edi]
pop [edi+0x4]
pop [edi+0x8]
pop [edi+0xC]
pop [edi+0x10]
pop [edi+0x14]
pop [edi+0x18]
pop [edi+0x1C]
```

Сначала машинные регистры помещаются в стек, а затем считываются из стека в область памяти, на которую указывает `EDI`; в свою очередь, `EDI` отвечает за указание на контекст VM. Эти сведения позволяют создать отображение между символическими внутренними элементами VM и ассемблерным выражением:

```
vm_symbolism = {
    :eax => :nhandler,
    :ebp => :vmkey,
    :esi => :bytecode_ptr,
    Indirection[[:edi], 4, nil] => :vm_edi,
    Indirection[[:edi, :+, 4], 4, nil] => :vm_esi,
    Indirection[[:edi, :+, 8], 4, nil] => :vm_ebp,
    Indirection[[:edi, :+, 0xC], 4, nil] => :vm_esp,
    Indirection[[:edi, :+, 0x10], 4, nil] => :vm_ebx,
    Indirection[[:edi, :+, 0x14], 4, nil] => :vm_edx,
    Indirection[[:edi, :+, 0x18], 4, nil] => :vm_ecx,
    Indirection[[:edi, :+, 0x1c], 4, nil] => :vm_eax,
}
```

Эта символика внедряется в связывание: каждое вхождение значения слева заменяется связанным с ним значением справа. У выражений есть специальный метод `bind`, который делает именно это. В следующем примере сначала определяется символическое выражение - сложение двух слагаемых, одно из которых является косвенным обращением; в нем участвуют два символа, `:a` и `:b`. Затем символу `:a` сопоставляется (связывается с ним) значение `1000h`:

```
expr = Expression[[:a, 4], :+, :b]
sym = {:a => 0x1000}

puts expr
>> dword ptr [a]+b

puts expr.bind(sym)
>> dword ptr [1000h]+b
```

Это можно обобщить для каждого выражения связывания. Такое отображение - ключ, позволяющий абстрагировать код VM от уровня его реализации и подняться до уровня семантики VM. Кроме того, положительным побочным эффектом этого шага часто становится значительное снижение сложности связывания. Новое связывание выглядит так:

```
[+] symbolic binding
  dword ptr [esp] => (dword ptr [bytecode_ptr]^vmkey)&0ffh
  dword ptr [edi+((dword ptr [bytecode_ptr]^vmkey)&0ffh)] =>
  dword ptr [edi+dword ptr [bytecode_ptr]^vmkey)&0ffh)]^
  dword ptr[edi+(((dword ptr [bytecode_ptr]>>8)^(vmkey>>8))&0ffh)]
  nhandler => (dword ptr [bytecode_ptr+4]^(vmkey^35ef6a14h))&0ffffffffh
  vmkey => (vmkey^35ef6a14h)&0ffffffffh
  bytecode_ptr => (bytecode_ptr+8)&0ffffffffh
```

Мы продвинулись вперед, но шифрование по-прежнему создает проблему, и двигаться дальше нельзя, если неизвестен контекст VM во время выполнения этого обработчика: обязательными значениями являются указатель байт-кода, изменяющийся ключ и, при необходимости, номер обработчика. Предположив, что эти значения известны (вы находитесь в точке входа VM или динамически протрассировали VM до этой точки), можно определить псевдоконтекст:

```
context = {
    :nhandler => 0x84,
    :vmkey => 0x5fdbd7b7,
    :bytecode_ptr => DATA_BASE_ADDR,
    :virt_eax => 0xffeeffee,
    :virt_ecx => 0,
    :virt_edx => 0x41414141,
    :virt_ebx => 1,
    :virt_edi => :virt_edi,
}
```

Обратите внимание, что контекст содержит как символические, так и числовые значения. Например, `nhandler` определен равным `84h`, тогда как регистр VM `virt_edi` является символическим.

Затем в связывание внедряются контекст и определенная ранее символика. На практике это итеративный процесс, однако не будем перегружать изложение подробностями реализации. Выражения постепенно разрешаются и сокращаются относительно всех известных значений, включая текущий контекст и данные программы (то есть байт-код). В итоге получается разрешенное связывание, которое фактически представляет контекст VM после выполнения обработчика. Этот шаг мы называем символьным выполнением:

```
[+] binding solver
  [+] key: dword ptr [esp]
    => solved key: dword ptr [esp]

  [+] value: (dword ptr [bytecode_ptr]^vmkey)&0ffh
    [+] solved memory read at 0x1a000000, size 4
    [+] value 5fdbcba3h
     => solved value: 14h

  [+] key: dword ptr [edi+((dword ptr [bytecode_ptr]^vmkey)&0ffh)]
    [+] solved memory read at 0x1a000000, size 4
    [+] value 5fdbcba3h
    => solved key: virt_edx

  [+] value: dword ptr [edi+((dword ptr [bytecode_ptr]^vmkey)&0ffh)]^
        dword ptr [edi+(((dword ptr [bytecode_ptr]>>8)^(vmkey>>8))&0ffh)]
    [+] solved memory read at 0x1a000000, size 4
    [+] value 5fdbcba3h
    [+] solved memory read at 0x1a000000, size 4
    [+] value 5fdbcba3h
    => solved value: 0beafbeafh

  [+] key: nhandler
    => solved key: nhandler

  [+] value: (dword ptr [bytecode_ptr+4]^(vmkey^35ef6a14h))&0ffffffffh
    [+] solved memory read at 0x1a000004, size 4
    [+] value 6a34bd60h
    => solved value: 0c3h

  [+] key: vmkey
    => solved key: vmkey
    [+] value: (vmkey^35ef6a14h)&0ffffffffh
    => solved value: 6a34bda3h

  [+] key: bytecode_ptr
     => solved key: bytecode_ptr
    [+] value: (bytecode_ptr+8)&0ffffffffh
    => solved value: 1a000008h

[+] solved binding
  virt_edx => 0beafbeafh
  nhandler => 0c3h
  vmkey => 6a34bda3h
  bytecode_ptr => 1a000008h
```

Решатель выражений помогает вычислить окончательные значения связывания первого обработчика: следующий обработчик, который должен быть выполнен, а также обновленное значение изменяющегося ключа. Обновление контекста - тривиальная операция:

```
updated_context = context.update(solved_binding)

puts "\n[+] updated context"
display(updated_context)
```

```
[+] updated context
  nhandler => 0c3h
  vmkey => 6a34bda3h
  bytecode_ptr => 1a000008h
  virt_eax => 0ffeeffeeh
  virt_ecx => 0
  virt_edx => 0beafbeafh
  virt_ebx => 1
  virt_edi => virt_edi
```

Этот процесс можно повторять, обходя весь граф потока управления VM. Результат весьма существенный; тем не менее чисто числовые значения несколько скрывают семантику обработчика. Кроме того, одна из целей - восстановить машинный ассемблерный код, эквивалентный контекстуализированному выполнению обработчика.

При анализе VM с помощью Metasm мы часто прибегаем к следующему приему: для каждого обработчика выполняем двойное символьное выполнение - одно с полным контекстом (главным образом числовые значения, используемые для обновления контекста), а другое с почти полностью символическим контекстом (используемым для извлечения высокоуровневой семантики). Следующий фрагмент кода демонстрирует использование символического контекста:

```
symbolic_context = {
  :nhandler => 0x84,
  :vmkey => 0x5fdbd7b7,
  :bytecode_ptr => DATA_BASE_ADDR,
  :virt_eax => :virt_eax,
  :virt_ecx => :virt_ecx,
  :virt_edx => :virt_edx,
  :virt_ebx => :virt_ebx,
  :virt_edi => :virt_edi,
}

solved_symolic_binding = sym_exec(symbolic_context,
                                  symbolic_binding,
                                  vm_symbolism)
puts "\n[+] solved binding" display(solved_symbolic_binding)
```

На этот раз вывод выглядит так:

```
[+] binding solver

  [+] key: dword ptr [esp]
    => solved key: dword ptr [esp]

  [+] value: (dword ptr [bytecode_ptr]^vmkey)&0ffh
    [+] solved memory read at 0x1a000000, size 4
    [+] value 5fdbcba3h
    => solved value: 14h

  [+] key: dword ptr [edi+((dword ptr [bytecode_ptr]^vmkey)&0ffh)]
    [+] solved memory read at 0x1a000000, size 4
    [+] value 5fdbcba3h
    => solved key: virt_edx

  [+] value: dword ptr [edi+((dword ptr [bytecode_ptr]^vmkey)&0ffh)]^
      dword ptr edi+(((dword ptr [bytecode_ptr]>>8)^(vmkey>>8))&0ffh)]
     [+] solved memory read at 0x1a000000, size 4
     [+] value 5fdbcba3h
     [+] solved memory read at 0x1a000000, size 4
     [+] value 5fdbcba3h
    => solved value: virt_edx^virt_eax

  [+] key: nhandler
    => solved key: nhandler

  [+] value: (dword ptr [bytecode_ptr+4]^(vmkey^35ef6a14h))&0ffffffffh
     [+] solved memory read at 0x1a000004, size 4
     [+] value 6a34bd60h
    => solved value: 0c3h
```

```
[+] key: vmkey
  => solved key: vmkey

[+] value: (vmkey^35ef6a14h)&0ffffffffh
  => solved value: 6a34bda3h
[+] key: bytecode_ptr
  => solved key: bytecode_ptr

[+] value: (bytecode_ptr+8)&0ffffffffh
  => solved value: 1a000008h

[+] solved binding
  virt_edx => virt_edx^virt_eax
  nhandler => 0c3h
  vmkey => 6a34bda3h
  bytecode_ptr => 1a000008h
```

Наконец, из разрешенного связывания можно просто исключить служебные элементы управления VM, в результате чего останется следующее:

```
vm_edx => vm_edx^vm_eax
```

Из этого конечного результата восстановить машинный код довольно просто. Этот процесс повторяется для каждого обработчика по всему графу потока управления VM. В общем случае при применении такого метода важен выбор момента, когда следует сохранять символические значения, а когда - сводить их к числовым. Первый вариант способствует восстановлению высокоуровневой семантики, тогда как второй облегчает восстановление потока управления VM. В конце вывод также можно подвергнуть дальнейшей обработке. Представьте VM, которая выглядит как стековый интерпретатор:

```
01: push vm_edx
02: add [esp], vm_eax
03: pop vm_edx
```

Классические методы деобфускации, такие как оптимизации компилятора, переписали бы предыдущие три строки следующим образом:

```
01: add vm_eax, vm_eax
```

Обратите внимание, что сам байт-код также мог быть обфусцирован.

Деобфускация уплощенного кода

Как уже говорилось, уплощение кода можно рассматривать как частичную виртуализацию: виртуализируется только поток управления. Поэтому методы, описанные для анализа VM, и особенно символьное выполнение, можно применять и для анализа уплощенного кода.

Однако остаются некоторые трудности, характерные именно для этого метода:

• Преобразование уплощения кода чаще всего применяется на уровне функции, а иногда в одной функции может находиться более одного уплощенного "узла". В целом это означает наличие нескольких экземпляров метода; следовательно, инструмент должен быть безотказным и полностью автоматическим.

• Отличить исходный код функции от добавленного кода диспетчера трудно, если реализация уплощения кода устойчива: потоки данных программы и диспетчера тесно переплетены и взаимозависимы.

• Обратное преобразование реализовать нетривиально.

Использование VxStripper

Чтобы продемонстрировать применение VxStripper к простому "игрушечному" примеру, рассмотрим следующую программу (она отображает y = 22) после распаковки и восстановления:

```
...... ! entrypoint:
...... !   push    ebp
401001 !   mov     ebp, esp
401003 !   sub     esp, 10h
401006 !   mov     dword ptr [ebp-4], 0
40100d !   mov     dword ptr [ebp-0ch], 2
401014 !   mov     dword ptr [ebp-8], 0ah
40101b !
...... ! loc_40101b:
...... !   cmp     dword ptr [ebp-0ch], 6
40101f !   jnl     loc_40108c
401021 !   mov     eax, [ebp-0ch]
401024 !   mov     [ebp-10h], eax
401027 !   mov     ecx, [ebp-10h]
40102a !   sub     ecx, 2

40102d !   mov     [ebp-10h], ecx
401030 !   cmp     dword ptr [ebp-10h], 3
401034 !   ja      loc_40108a
401036 !   mov     edx, [ebp-10h]
401039 !   jmp     dword ptr [edx*4+data_4010a4]
401040     mov     dword ptr [ebp-4], 2
401047     mov     dword ptr [ebp-0ch], 3
40104e     jmp     loc_40108a
401050     cmp     dword ptr [ebp-8], 0
401054     jng     40105fh
401056     mov     dword ptr [ebp-0ch], 4
40105d     jmp     401066h
40105f     mov     dword ptr [ebp-0ch], 6
401066     jmp     loc_40108a
401068     mov     eax, [ebp-4]
40106b     add     eax, 2
40106e     mov     [ebp-4], eax
401071     mov     dword ptr [ebp-0ch], 5
401078     jmp     loc_40108a
40107a     mov     ecx, [ebp-8]
40107d     sub     ecx, 1
401080     mov     [ebp-8], ecx
401083     mov     dword ptr [ebp-0ch], 3
40108a !
...... ! loc_40108a:
...... !   jmp     loc_40101b
40108c !
...... ! loc_40108c:
...... !   mov     edx, [ebp-4]
40108f !   push    edx
401090 !   push    strz_yd_402008
401095 !   call    dword ptr [msvcrt.dll:printf]
40109b !   add     esp, 8
40109e !   xor     eax, eax
4010a0 !   mov     esp, ebp
4010a2 !   pop     ebp
4010a3 !   ret    return 0;
```

Граф потока управления (CFG) такой программы уплощен.

Выполнение модуля нормализации создает следующий код (если применять не все оптимизации):

```
...... !   push    eax
4011f1 !   mov     dword ptr [esp-0ch], 0ah
4011f9 !   mov     dword ptr [esp-8], 4
401201 !   dec     dword ptr [esp-0ch]
401205 !   add     dword ptr [esp-8], 2
40120a !   dec     dword ptr [esp-0ch]
40120e !   add     dword ptr [esp-8], 2
401213 !   dec     dword ptr [esp-0ch]
401217 !   add     dword ptr [esp-8], 2
40121c !   dec     dword ptr [esp-0ch]
401220 !   add     dword ptr [esp-8], 2

401225 !   dec     dword ptr [esp-0ch]
401229 !   add     dword ptr [esp-8], 2
40122e !   dec     dword ptr [esp-0ch]
401232 !   add     dword ptr [esp-8], 2
401237 !   dec     dword ptr [esp-0ch]
40123b !   add     dword ptr [esp-8], 2
401240 !   dec     dword ptr [esp-0ch]
401244 !   add     dword ptr [esp-8], 2
401249 !   dec     dword ptr [esp-0ch]
40124d !   add     dword ptr [esp-8], 2
401252 !   dec     dword ptr [esp-0ch]
401256 !   mov     eax, [esp-8]
40125a !   mov     [esp-18h], eax
40125e !   mov     dword ptr [esp-1ch], strz_yd_402010
401266 !   mov     ebp, esp
401268 !   lea     eax, [esp-1ch]
40126c !   mov     esp, eax
40126e !   call    crtdll.dll:printf_4012d8
401273 !   mov     esp, ebp
401275 !   mov     ebp, esp
401277 !   lea     eax, [esp+8]
40127b !   mov     esp, eax
40127d !   mov     esp, ebp
40127f !   xor     eax, eax
401281 !   pop     edx
401282 !   ret
```

Обратите внимание, что динамическое формирование кода, выполняемое VxStripper, естественным образом устраняет уплощение уплощенного кода. Применение стандартных оптимизирующих преобразований дает программу, очищенную от этой обфускации:

```
...... !   push    eax
4011f1 !   mov     dword ptr [esp-18h], 16h
4011f9 !   mov     dword ptr [esp-1ch], strz_yd_402010
401201 !   mov     ebp, esp
401203 !   lea     eax, [esp-1ch]
401207 !   mov     esp, eax
401209 !   call    crtdll.dll:printf_401268
40120e !   mov     esp, ebp
401210 !   mov     ebp, esp
401212 !   lea     eax, [esp+8]
401216 !   mov     esp, eax
401218 !   mov     esp, ebp
40121a !   xor     eax, eax
40121c !   pop     edx
40121d !   ret
```

Хотя до получения программы, поддерживающей весь набор средств программной защиты, доступных авторам вредоносных программ, еще предстоит работа, эти первые результаты побуждают продолжать исследование универсальных методов распаковки и нормализации с целью максимально возможной автоматизации задач, выполняемых аналитиком.

Этот инструмент предоставляет самодостаточную программу для анализа вредоносного ПО. Однако одна из будущих целей проекта - обеспечить взаимодействие инструмента с другими средствами анализа. По своему устройству этот инструмент может сотрудничать с любым средством анализа программ, основанным на цепочке компиляции LLVM.

Цепочка компиляции LLVM и множество уже существующих инструментов на основе LLVM предоставляют обширную библиотеку видов анализа программ, которые можно совместно применять для преодоления механизмов защиты вредоносного ПО.

Кроме того, Vellvm (Verified LLVM) можно использовать для формального извлечения проверенных реализаций проходов деобфускации, реализованных в VxStripper. Помимо анализа угроз вредоносного ПО можно представить и другие применения этого инструмента, например извлечение схем обнаружения, анализ программных защит и устойчивости антивирусных программ.

Практический пример

Образец, который будет использоваться в этом практическом примере, в действительности представляет собой crackme, первоначально опубликованный пользователем quetz на Crackmes.de в 2007 году. Хотя это "всего лишь" crackme, в нем реализовано большинство понятий, встречающихся в защите профессионального уровня. Среди прочих радостей он содержит:

• уплощение кода;

• кодирование переменных;

• виртуализацию кода.

Вот как автор представляет свою задачу:

> В последнее время защита от статического анализа становится все популярнее. Почти каждое средство защиты применяет какую-либо обфускацию, виртуальную машину и т. д. Этот keygenme - попытка показать, что происходит при злоупотреблении идеей обфускации. Может ли человек эффективно анализировать такой код? Возможно, с помощью инструмента?
>
> - http://crackmes.de/users/quetz/q_keygenme_1.0/

К счастью, у нас есть инструменты, и в этом разделе мы ими воспользуемся. Перед началом рекомендуем не заглядывать в раздел символов, содержащийся в двоичном файле. К слову, предыдущие версии IDA Pro (возможно, ниже 6.2) не загружали эти символы, и автор этих строк с радостью не догадался их поискать.

Первые впечатления

После запуска исполняемого файла можно ввести имя пользователя и пароль. После нажатия кнопки `Check` появляется окно сообщения, показывающее, действительны ли введенные учетные данные.

Если вы внимательно прочитали предыдущие главы этой книги, то, вероятно, уже запустили любимый дизассемблер и нацелились на функцию обратного вызова `DialogProc` графического интерфейса.

Сначала рассмотрим общую архитектуру защищенного кода. Граф потока управления слишком запутан, чтобы быть естественным кодом, созданным компилятором: имеется обфусцированная `DialogProc`, вызывающая две функции с уплощенным кодом (`func1@0x430DB0`, `func2@0x431E00`). Эти две функции, в свою очередь, вызывают то, что похоже на VM (`vm@0x401360`).

Мы уже обсуждали единственную точку диспетчеризации VM; эту точку довольно легко обнаружить: при отсутствии других подсказок ищите крупную таблицу переходов.

```
01: .text:00401F20

02: mov    ebp, [esp+13Ch]
03: cmp    ebp, 3E2Dh; переключатель, 15918 вариантов
04: ja short loc_401F36
05: jmp    ds:off_43D000[ebp*4]; переход переключателя
```

Так мы почти бесплатно получили первые две порции сведений о VM: номер текущего обработчика хранится в `[ESP+13Ch]`, а в диспетчере имеется 15 918 записей (пока нельзя заключить, что все они являются разными обработчиками).

Прежде чем приступать к трудоемкой работе, можно попытаться выполнить анализ функций `func1` и `func2` как черных ящиков. Мы знаем, что `func1` и `func2` вызывают VM; просто регистрируем каждый вызов VM и особенно номер первого вызываемого обработчика, своего рода точки входа в код VM. Это кажется совершенно тривиальным, но никогда не следует пренебрегать легко доступными результатами.

Результаты сразу многое раскрывают: `func2` вызывается перед `func1`, поэтому начнем с `func2`. При первом же взгляде на журналы точки входа VM выделяются следующие шаблоны:

• `546h-0BFFh-7B2h-9A2h-405h-919h-3B9h` - 624 раза;

• `0CF5h-15Eh-184h-39Ch-5B0h-3C0h-0F75h` - 624 раза;

• `0A06h-0xA29h-0x268h-0xCB3h` - 227 раз;

• `736h-13Ah-1EBh-897h` - 396 раз;

• `150h-8ABh-843h-697h-474h` - 200 раз.

На самом деле это уже очень много сведений. Если посмотреть в раздел `.data`, там ожидает дополнительная подсказка:

```
01: .data:0043CA44  dword_43CA4402: .data:0043CA48  dd 9908B0DFh
```

Откуда взялось `9908B0DFh`? А 624? А 227? Что ж, либо вы очень хорошо знакомы с генераторами случайных чисел, либо ищете эти значения: они указывают на алгоритм генератора псевдослучайных чисел "вихрь Мерсенна". 624 и 397 - параметры периода, а `9908B0DFh` - константа, используемая при генерации чисел.

Мы выявили критическую слабость защиты: виртуализированный код раскрывает некоторые сведения о структуре защищенного алгоритма, благодаря чему восстановить число итераций цикла становится тривиально. Тем не менее получили ли мы аккуратно подготовленный виртуализированный код, который обезвреживается одной точкой останова? Еще нет. У нас есть кандидат на алгоритм, но его требуется подтвердить.

И вновь: никогда не следует заниматься обратной разработкой кода, если сведения можно угадать и проверить! В данном случае анализ черного ящика многое раскрывает при простом наблюдении за входами и выходами функций `func1` и `func2`. Такая базовая стратегия или дифференциальный анализ выполнения VM иногда могут оказать огромную помощь. Уточним анализ этих двух функций:

• `func2`

• Вход - два аргумента: адрес в стеке, который, по-видимому, является массивом целых чисел, и 32-разрядное значение, которое, по-видимому, зависит от длины имени.

• Выход - ничего примечательного, кроме обновления массива целых чисел.

• Число вызовов - вызывается один раз, в начале и перед `func1`.

• Предположение - инициализация "вихря Мерсенна"; массив в действительности является состоянием PNRG. 32-разрядное значение - начальное значение. Это можно дополнительно проверить, сопоставив параметры цикла (узнанные из журналов) со стандартной функцией инициализации.

• `func1`

• Вход - два аргумента: адрес (предполагаемого) состояния PRNG и 32-разрядное значение, которое, по-видимому, является буквой имени пользователя.

• Выход - возвращает 32-разрядное случайное значение.

• Число вызовов - вызывается 100 раз.

• Предположение - функция, подобная `rand32`, для "вихря Мерсенна".

Анализ семантики обработчиков

Теперь пора анализировать VM. Главный диспетчер уже найден по адресу `[ESP+13ch]`. Часто полезно вручную проверить несколько обработчиков, чтобы увидеть, как они обращаются к контексту VM, как обновляют счетчик команд и/или указатель байт-кода и так далее.

Этот процесс можно применить к случайному обработчику - например, начинающемуся по адресу `0x41836c`:

```
01: .text:0041836C loc_41836C:
02:         ; ПЕРЕКРЕСТНАЯ ССЫЛКА ДАННЫХ: .rdata:off_43D000
03:         ; таблица переходов 00401F2F, варианты 2815,4091
04: .text:0041836C  movzx ecx, [esp+3D8h+var_2A2]

05: .text:00418374  mov   esi, 97Fh
06: .text:00418379  mov   ebx, [esp+3D8h+var_3C8]
07: .text:0041837D  movzx edi, [esp+3D8h+var_29E]
08: .text:00418385  add   [esp+3D8h+var_3C0], 94Eh
09: .text:0041838D  imul  eax, ecx, 1Ch
10: .text:00418390  sub   [esp+3D8h+var_3C4], 0FF0h
11: .text:00418398  imul  ecx, edi, 5DDh
12: .text:0041839E  mov   [esp+3D8h+var_37C], esi
13: .text:004183A2  lea   edx, [eax+ebx+5C8h]
14: .text:004183A9  mov   ebx, 97Fh
15: .text:004183AE  mov   [esp+3D8h+var_3C8], edx
16: .text:004183B2  sub   ebx, ecx
17: .text:004183B4  lea   edx, [ebp+ebx+29Dh+var_8BC]
18: .text:004183BB  mov   [esp+3D8h+var_378], ebx
19: .text:004183BF  mov   [esp+3D8h+var_380], ebx
20: .text:004183C3  mov   [esp+3D8h+var_29D+1], edx
21: .text:004183CA  jmp   loc_401F20
```

Обратимся за помощью к Metasm. Как было показано ранее, метод `code-binding` можно использовать для вычисления семантики фрагмента кода:

```
dword ptr [esp+10h] => 1ch*byte ptr [esp+136h]+dword ptr [esp+10h]+5c8h
dword ptr [esp+14h] => dword ptr [esp+14h]-0ff0h
dword ptr [esp+18h] => dword ptr [esp+18h]+94eh
dword ptr [esp+58h] => -5ddh*byte ptr [esp+13ah]+97fh
dword ptr [esp+5ch] => 97fh
dword ptr [esp+60h] => -5ddh*byte ptr [esp+13ah]+97fh
dword ptr [esp+13ch] => ebp-5ddh*byte ptr [esp+13ah]+360h
eax => (1ch*byte ptr [esp+136h])&0ffffffffh
ecx => (5ddh*byte ptr [esp+13ah])&0ffffffffh
edx => (ebp-5ddh*byte ptr [esp+13ah]+360h)&0ffffffffh
ebx => (-5ddh*byte ptr [esp+13ah]+97fh)&0ffffffffh
esi => 97fh
edi => byte ptr [esp+13ah]&0ffffffffh
```

Контекст VM хранится в стеке, и никакие значения не передаются между обработчиками через регистры; это означает, что для более ясного представления все изменения регистров можно отбросить:

```
dword ptr [esp+10h] => 1ch*byte ptr [esp+136h]+dword ptr [esp+10h]+5c8h
dword ptr [esp+14h] => dword ptr [esp+14h]-0ff0h
dword ptr [esp+18h] => dword ptr [esp+18h]+94eh
dword ptr [esp+58h] => -5ddh*byte ptr [esp+13ah]+97fh
dword ptr [esp+5ch] => 97fh
dword ptr [esp+60h] => -5ddh*byte ptr [esp+13ah]+97fh
dword ptr [esp+13ch] => ebp-5ddh*byte ptr [esp+13ah]+360h
```

Мы уже знаем, что номер обработчика хранится в `[ESP+13Ch]`. Обработчик обновляет его. Конечное значение зависит от значения `byte ptr [ESP+13ah]`. Проанализировав несколько других обработчиков, можно предположить, что это логическое значение, а в контексте хранятся еще несколько логических значений. Это значение хранится во второй позиции и будет названо `flag2`.

`[ESP+58h]`, `[ESP +5Ch]` и `[ESP +60h]` тесно связаны с вычислением номера обработчика. Они соответственно содержат разность между старым и новым номерами обработчика на случай, если условие (здесь `flag2`) истинно или ложно.

`[ESP +10h]`, `[ESP +14h]` и `[ESP +18h]` также представляют большой интерес. Они обновляются почти каждым обработчиком и, предположительно, расшифровывают байт-код: они обращаются к большой таблице неопределенных констант, хранящейся в разделе `.data`). В действительности они подобны работающему ключу; назовем их соответственно `key_a`, `key_b` и `key_c`.

Ниже приведен пример использования ключа из обработчика `0xa0a` по адресу `0x427b17`:

```
nHandler => dword ptr [4*key_c+436010h]^
            dword ptr [4*key_b+436010h]^
            dword ptr [4*key_a+436010h]
```

Имена можно внедрить в связывание обработчика, сделав его понятнее (даже если здесь это не главная цель) и удобнее для манипулирования:

```
key_a => 1ch*flag6+key_a+5c8h
key_b => key_b-0ff0h
key_c => key_c+94eh
delta_true => -5ddh*flag2+97fh
delta_false => 97fh
delta => -5ddh*flag2+97fh
nHandler => ebp-5ddh*flag2+360h
```

Этот обработчик обладает почти такой же семантикой, как условный переход. Проанализировав несколько других обработчиков, можно восстановить и проверить отображение символических переменных VM, которое будет представлено хеш-объектом:

```
SYMBOLIC_VM = {
    Indirection[Expression[ :esp, :+, 0x10], 4, nil] => :key_a,
    Indirection[Expression[ :esp, :+, 0x14], 4, nil] => :key_b,
    Indirection[Expression[ :esp, :+, 0x18], 4, nil] => :key_c,
    Indirection[Expression[ :esp, :+, 0x58], 4, nil] => :delta,
    Indirection[Expression[ :esp, :+, 0x5c], 4, nil] => :delta_false,
    Indirection[Expression[ :esp, :+, 0x60], 4, nil] => :delta_true,
    Indirection[Expression[ :esp, :+, 0x134], 1, nil] => :flag8,
    Indirection[Expression[ :esp, :+, 0x135], 1, nil] => :flag7,
    Indirection[Expression[ :esp, :+, 0x136], 1, nil] => :flag6,
    Indirection[Expression[ :esp, :+, 0x137], 1, nil] => :flag5,
    Indirection[Expression[ :esp, :+, 0x138], 1, nil] => :flag4,
    Indirection[Expression[ :esp, :+, 0x139], 1, nil] => :flag3,
    Indirection[Expression[ :esp, :+, 0x13a], 1, nil] => :flag2,
    Indirection[Expression[ :esp, :+, 0x13b], 1, nil] => :flag1,
    Indirection[Expression[ :esp, :+, 0x13c], 4, nil] => :nHandler
}
```

У других ячеек памяти, по-видимому, нет специального назначения; их можно считать или отображать как регистры общего назначения. С этим отображением у нас есть все необходимое для символьного выполнения VM, то есть пошагового выполнения семантики обработчиков.

Символьное выполнение

Чтобы выполнить символьное выполнение, необходимо иметь некоторые сведения об исходном контексте инициализации VM: вспомните исходное значение изменяющегося ключа или значение счетчика команд (номер обработчика). В нашем случае сами вызовы VM обфусцированы (вспомните рассмотренные ранее функции с уплощенным графом), поэтому статически восстановить контекст инициализации довольно трудно. В такой ситуации можно просто взять лучшее из двух миров и использовать компромисс между статическим и динамическим анализом, иногда называемый конколическим выполнением.

В сущности, это означает, что вы отлаживаете цель и перехватываете каждое обращение к VM, устанавливая на нем точку останова; внутри функции обратного вызова вы переключаетесь с динамического анализа на статический и выполняете следующие действия:

1. Создать дамп памяти цели.

2. Инициализировать контекст символьного анализа дампом памяти. В действительности можно использовать разновидность отложенной загрузки. Все обращения к неинициализированному контексту будут разрешаться с помощью дампа памяти и кэшироваться.

3. Вычислить символьное выполнение VM.

При использовании конколического выполнения следовать за потоком выполнения VM, то есть последовательностью обработчиков, становится довольно легко. Процесс анализа и трассировки обработчиков полностью автоматизирован.

Ниже приведен пример вывода инструмента для одного обработчика. Хорошо заметно интенсивное использование изменяющегося ключа, состоящего из `key_a`, `key_b` и `key_c`; в данном случае ключ используется для обфускации доступа к контексту VM:

```
[+] disasm handler 2be at 42c2cdh
[+] analyzing handler at 0x42c2cd
[+] considering code from 0x42c2cd to 0x42c3b3
[+] cached handler binding

dword ptr [dword ptr [esp+4*(dword ptr [4*key_b+436000h]^
  (dword ptr [4*key_c+436000h]^dword ptr [4*key_a+436000h]))+140h]] =>
  dword ptr [dword ptr [esp+4*(dword ptr [4*key_b+436004h]^
  (dword ptr [4*key_c+436004h]^dword ptr [4*key_a+436004h]))+140h]]

key_c => key_c+5
key_b => key_b+5
key_a => key_a+5

nHandler => (dword ptr [4*key_b+43600ch]^
  (dword ptr [4*key_c+43600ch]^dword ptr [4*key_a+43600ch]))+
  (((dword ptr [4*key_c+436010h]^(dword ptr [4*key_b+436010h]^
  dword ptr [4*key_a+436010h]))*(byte ptr [dword ptr [esp+4*

  (dword ptr [4*key_b+436008h]^(dword ptr [4*key_c+436008h]^
  dword ptr [4*key_a+436008h]))+140h]]&0ffh))&0ffffffffh)

[+] symbolic binding

dword ptr [esp+0a8h] => dword ptr [esp+0ach]
key_c => 0d6h
key_b => 1f6h
key_a => 126ah
nHandler => ((21eh*(flag4&0ffh))&0ffffffffh)+0ac6h

[+] solved binding
dword ptr [esp+0a8h] => 4f3de0b9h
key_c => 0d6h
key_b => 1f6h
key_a => 126ah n
Handler => 0ac6h
```

Решение задачи

То, что мы разработали до сих пор, эквивалентно инструменту трассировки на уровне VM. Для получения инструмента, ориентированного на дизассемблирование, потребовалась бы обработка ветвящихся команд - условных и безусловных переходов, вызовов. Можно было бы построить более сложный инструмент, своего рода компилятор, основанный на дизассемблере байт-кода и способный восстанавливать машинный код. На предыдущем примере инструмент обрабатывал бы вход:

```
dword ptr [ESP+0a8h] => dword ptr [ESP+0ach]
```

и выдавал бы исходный код, подобный C:

```
vm_ctx.reg_2ah = vm_ctx.reg_2bh;
```

Для этого образца мы будем использовать только возможность трассировки. Стратегия проста. У нас есть хорошее представление об алгоритме, реализованном VM, поэтому мы воспользуемся анализом черного ящика и дифференциальным анализом, чтобы выявить расхождения между стандартным алгоритмом "вихря Мерсенна" (MT) и алгоритмом VM. Когда расхождение будет обнаружено, мы проверим вывод трассировки.

Снова рассмотрим пример: инициализацию состояния алгоритма MT. Инициализация реализована функцией `func2`. Мы будем смотреть лишь на ее входы и выходы. Состояние - это массив из 624 двойных слов. При использовании стандартных реализаций и того же начального значения, которое программа применяет для имени из шести символов (`3961821h`), получаем следующее:

• Стандартная реализация - `state[1] = 0x968bff6d`;

• Реализация VM - `state[1] = 0x968e4c84`.

Ищем эти значения в трассе:

```
[+] disasm handler 2c2 at 41f056h
[+] analyzing handler at 0x41f056
[+] considering code from 0x41f056 to 0x41f165

[+] cached handler binding

byte ptr [esp+0dh] => byte ptr [dword ptr [esp+4*(dword ptr
  [4*key_b+43600ch]^(dword ptr [4*key_c+43600ch]^dword ptr
  [4*key_a+43600ch]))+140h]]&0ffh
dword ptr [dword ptr [esp+4*(dword ptr [4*key_b+436000h]^(dword ptr
  [4*key_c+436000h]^dword ptr [4*key_a+436000h]))+140h]] => dword ptr [dword
ptr
  [esp+4*(dword ptr [4*key_b+436004h]^(dword ptr [4*key_c+436004h]^dword ptr
  [4*key_a+436004h]))+140h]]^dword ptr [dword ptr [esp+4*(dword ptr
  [4*key_b+436008h]^(dword ptr [4*key_c+436008h]^dword ptr
  [4*key_a+436008h]))+140h]]
key_c => key_c+6
key_a => key_a+6
key_b => key_b+6
nHandler => (dword ptr [4*key_b+436010h]^(dword ptr [4*key_c+436010h]^dword
ptr
  [4*key_a+436010h]))+(((dword ptr [4*key_c+436014h]^(dword ptr
  [4*key_b+436014h]^dword ptr [4*key_a+436014h]))*(byte ptr [dword ptr
  [esp+4*(dword ptr [4*key_b+43600ch]^(dword ptr [4*key_c+43600ch]^dword ptr
  [4*key_a+43600ch]))+140h]]&0ffh))&0ffffffffh)

[+] symbolic binding
byte ptr [esp+0dh] => flag2&0ffh
dword ptr [esp+100h] => dword ptr [esp+9ch]^dword ptr [esp+10ch]
key_c => 10e2h
key_a => 12b3h
key_b => 0c8ah
nHandler => ((164h*(flag2&0ffh))&0ffffffffh)+0a53h

[+] solved binding
byte ptr [esp+0dh] => 1
dword ptr [esp+100h] => 968e4c84h
key_c => 10e2h
key_a => 12b3h
key_b => 0c8ah
```

Имеется операция XOR между dword ptr [ESP+9ch] и dword ptr [ESP+10ch]. Их значения можно проверить по контексту:

```
[+] context dump
[...]
dword ptr [esp+9ch] => 968bff6dh
dword ptr [esp+10ch] => 5b3e9h
[...]
```

Этот обработчик обладает семантикой, подобной XOR, и входит в один из ранее выявленных циклов - тот, где выполняются 624 итерации, по размеру состояния MT. Для восстановления полного преобразования требуется еще несколько шагов, однако такого подхода достаточно. Его псевдокод выглядел бы следующим образом:

```
scramble = 0x5b3e9h
for i in (N-1)
  state[i+1] ^= scramble

  scramble = lcg_rand(scramble)
```

где N - размер состояния: 624. lcg_rand - линейный конгруэнтный генератор x_(n+1) ≡ (ax_n + c) (mod m), где a, c и m соответственно равны 0x159b, 0x13e8b и 0xffffffff.

Остальная часть алгоритма "вихря Мерсенна" также немного изменена; в каждом из этих изменений участвуют первая буква имени пользователя и простая операция ADD/SUB/XOR. Больше мы об этих изменениях не скажем; обратитесь к следующему разделу "Упражнения".

• Имя пользователя - "Hell yeah, we have tools!"

• Серийный номер - "117538a51905ddf6"

Заключительные соображения

Этот образец - замечательная площадка, искусно созданная его автором. Для его исследования мы применили интересное сочетание динамического и статического анализа. Защита реализует уплощение кода, виртуализацию кода и кодирование данных - понятия, встречающиеся в большинстве систем защиты профессионального уровня, - но при этом остается доступной. Она предоставляет полезный шаблон для совершенствования инструментов и экспериментов с новыми идеями и/или алгоритмами. Простота схемы защиты и алгоритма позволила нам воспользоваться в этом разделе множеством сокращений пути.

Упражнения

Первое предлагаемое упражнение - создать генератор ключей для двоичного файла из практического примера этой главы. Это замечательная отправная точка:

• двоичный файл уникален, относительно невелик, его легко анализировать, дизассемблировать и инструментировать. Поэтому задача доступна даже начинающим;

• большинство важных реализованных методов описано в практическом примере. Найдите их и убедитесь, что понимаете их внутреннюю работу.

После прочтения этой главы самостоятельная работа с задачей даст бесценный опыт. Ваше задание:

1. На основе предложенной методики (или придуманной вами) создайте собственный инструмент для анализа байт-кода VM.

2. Свяжитесь с любимым демо-подразделением и упакуйте потрясающий генератор ключей для этого прекрасного crackme.

Чтобы познакомиться с Metasm, среди материалов, поставляемых с книгой, вы найдете два сценария упражнений: `symbolic-execution-lvl1.rb` и `symbolic-execution-lvl2.rb`. Ответы на вопросы отправят вас в путешествие по внутренностям Metasm. Сценарии можно найти по адресу www.wiley.com/go/practical-reverseengineering.com.

Примечания

[1] [назад] Бансал, Сорав, и Айкен, Алекс. "Автоматическое создание локальных супероптимизаторов", 2006, http://theory.stanford.edu/~sbansal/pubs/asplos06.pdf.
[2] [назад] Барак, Боаз, и др. "О (не)возможности обфускации программ". Технический отчет, Electronic Colloquium on Computational Complexity, 2001. http://www.eccc.uni-trier.de/eccc.
[3] [назад] Бекман, Леннарт, и др. "Частичный вычислитель и его применение как средства программирования", Artificial Intelligence, том 7, выпуск 4, с. 319-357, 1976.
[4] [назад] Беллар, Фабрис. "QEMU - быстрый и переносимый динамический транслятор". Доклад, представленный в Proceedings of the USENIX Annual Technical Conference, направление FREENIX, 41-46, 2005.
[5] [назад] Бийе, Оливье, Жильбер, Анри, и Эш-Шатби, Шараф. "Криптоанализ реализации AES в белом ящике". В Selected Areas in Cryptography, под редакцией Хелены Хандшух и М. Анвара Хасана, 227-240. Springer, 2004.
[6] [назад] Бойер, Р. С., Элспас, Б., и Левитт, К. Н. SELECT - формальная система для тестирования и отладки программ посредством символьного выполнения. SIGPLAN Not., 10:234-245, 1975.
[7] [назад] Чипунов, В., и Кандеа, Г. "Обеспечение сложных видов анализа двоичных файлов x86 с помощью RevGen". Доклад, представленный на семинарах Dependable Systems and Networks Workshops (DSN-W), 41-я международная конференция IEEE/IFIP, 211-216, 2011.
[8] [назад] Чипунов, В., Кузнецов, В., и Кандеа, Г. "S2e: платформа для многопутевого анализа программных систем в процессе выполнения", ACM SIGARCH Computer Architecture News, том 39, N 1, 265-278, 2011.
[9] [назад] Чоу, С., Айзен, П. А., Джонсон, Х., и ван Орсхот, П. К. Реализация DES в белом ящике для приложений DRM. В Security and Privacy in Digital Rights Management, семинар ACM CCS-9, DRM 2002, Вашингтон, округ Колумбия, США, 18 ноября 2002 г., исправленные статьи, том 2696 серии Lecture Notes in Computer Science, 1-15. Springer, 2002.
[10] [назад] Чоу, С., Айзен, П. А., Джонсон, Х., и ван Орсхот, П. К. Криптография в белом ящике и реализация AES. В Selected Areas in Cryptography, том 2595 серии Lecture Notes in Computer Science, 250-270. Springer, 2002.
[11] [назад] Чоу, Стэнли Т., Джонсон, Гарольд Дж., и Гу, Юань. Устойчивое к вмешательству кодирование программного обеспечения, 2003.
[12] [назад] Коллберг, Кристиан, Томборсон, Кларк, и Лоу, Дуглас. Таксономия обфусцирующих преобразований. Технический отчет, 1997.
[13] [назад] Коллберг, Кристиан С., Томборсон, Кларк Д., и Лоу, Дуглас. "Создание дешевых, устойчивых и скрытных непрозрачных конструкций". В POPL, 184-196, 1998.
[14] [назад] Кузо, П., и Кузо, Р. "Абстрактная интерпретация: единая решеточная модель статического анализа программ посредством построения или приближения неподвижных точек". В материалах 4-го симпозиума ACM по принципам языков программирования (POPL '77), 238-252. ACM Press, Нью-Йорк, 1977.
[15] [назад] Кузо, П., и Кузо, Р. "Систематическое проектирование каркасов преобразования программ посредством абстрактной интерпретации". В материалах двадцать девятого ежегодного симпозиума ACM SIGPLAN-SIGACT по принципам языков программирования, 178-190. Нью-Йорк, 2002. ACM Press.
[16] [назад] Кузо, П. "Конструктивное проектирование иерархии семантик системы переходов посредством абстрактной интерпретации". ENTCS, 6, 1997.
[17] [назад] Далла Преда, Мила. "Обфускация кода и обнаружение вредоносного ПО посредством абстрактной интерпретации", докторская диссертация, http://profs.sci.univr.it/~dallapre/MilaDallaPreda_PhD.pdf.
[18] [назад] Далла Преда, Мила, и Джакобацци, Роберто. "Обфускация управляющего кода посредством абстрактной интерпретации". В Third IEEE International Conference on Software Engineering and Formal Methods, 301-310, 2005.
[19] [назад] Дероко. Nanomites.w32. http://deroko.phearless.org/nanomites.zip.
[20] [назад] Эрнст, Майкл Д. "Статический и динамический анализ: синергия и двойственность". В Proceedings of WODA 2003: Workshop on Dynamic Analysis, Портленд, Орегон, 24-27 мая 2003.
[21] [назад] Футамура, Ёсихико. "Частичное вычисление вычислительного процесса - подход к компилятору компиляторов", 1999. http://cs.au.dk/~hosc/local/HOSC-12-4-pp381-391.pdf.
[22] [назад] Газе, Александр, и Гийо, Йоанн. "Преодоление программной защиты с помощью Metasm". В HITB Malaysia, Куала-Лумпур, 2009. http://metasm.cr0.org/docs/2009-guillot-gazet-hitb-deprotection.pdf.
[23] [назад] Годфруа, П., Кларлунд, Н., и Сен, К. "DART: направленное автоматизированное случайное тестирование". В PLDI '05, июнь 2005.

[24] [назад] Губен, Л., Мазерель, Ж. М., и Кискатер, М. "Криптоанализ реализаций DES в белом ящике". Cryptology ePrint Archive, отчет 2007/035, 2007.
[25] [назад] Якоб, Маттиас, и др. "Супердиверсификатор: локальная индивидуализация для защиты программного обеспечения". 2008. http://research.microsoft.com/apps/pubs/default.aspx?id=77265.
[26] [назад] Якубовский, Мариус Х., и др. "Итеративные преобразования и количественные метрики для защиты программного обеспечения", 2009. http://research.microsoft.com/apps/pubs/default.aspx?id=81560.
[27] [назад] Жосс, С. "Безопасная и совершенная распаковка с использованием компьютерной эмуляции". В Proceedings of the AVAR 2006 Conference, Окленд, Новая Зеландия, 3-5 декабря, 174-190, 2006.
[28] [назад] Жосс, С. "Обнаружение руткитов извне матрицы". Journal in Computer Virology, том 3, 113-123. Springer, 2007.
[29] [назад] Жосс, С. "Динамическая перекомпиляция вредоносного ПО". В IEEE Proceedings of the 47th HICSS Conference, 2014.
[30] [назад] Киндер, Йоханнес, Цулегер, Флориан, и Вайт, Хельмут. "Каркас на основе абстрактной интерпретации для восстановления потока управления из двоичных файлов". 2009. http://pure.rhul.ac.uk/portal/files/17558147/vmcai09.pdf.
[31] [назад] Латтнер, К., и Адве, В. "LLVM: каркас компиляции для анализа и преобразования программ на всем протяжении их жизни". В International Symposium on Code Generation and Optimization, 75-86, 2004.
[32] [назад] Максимус. "Обратная разработка простой виртуальной машины". 2006. http://tuts4you.com/download.php?view.210.
[33] [назад] Максимус. "Перестроение виртуальных машин". 2006. http://tuts4you.com/download.php?view.1229.
[34] [назад] Роллес, Рольф. "Поиск ошибок в VM с помощью средства доказательства теорем, раунд 1". http://www.openrce.org/blog/view/1963/Finding_Bugs_in_VMs_with_a_Theorem_Prover,_Round_1.
[35] [назад] Роллес, Рольф. "Преодоление HyperUnpackMe2 с помощью процессорного модуля IDA", 2006. http://www.openrce.org/articles/full_view/28.
[36] [назад] Шеллер, Т. "Llvm-qemu, серверная часть QEMU, использующая компоненты LLVM", Google Summer of Code 2007. http://code.google.com/p/llvm-qemu/.
[37] [назад] Сен, К., Маринов, Д., и Ага, Г. "CUTE: движок конколического модульного тестирования для C". В ESEC/FSE '05, сентябрь 2005.
[38] [назад] Смарагдакис, Яннис, и Ксаллнер, Кристоф. "Объединение статических и динамических рассуждений для обнаружения ошибок". В Proceedings of International Conference on Tests and Proofs (TAP), LNCS, том 4454, 1-16, Springer, 2007.
[39] [назад] Смитсон, Мэтт, и др. "Система анализа и переписывания двоичных файлов на основе промежуточного представления уровня компилятора". В MALWARE, 47-54, 2010.
[40] [назад] Сонг, Дон, и др. "BitBlaze: новый подход к компьютерной безопасности посредством анализа двоичных файлов". В Proceedings of the 4th International Conference on Information Systems Security, Хайдарабад, Индия, 2008.
[41] [назад] Такур, А., и др. "Направленное создание доказательств для машинного кода". 2010. http://research.cs.wisc.edu/wpis/papers/cav10-mcveto.pdf.
[42] [назад] Вайзер, М. "Срезы программ: формальные, психологические и практические исследования автоматического метода абстрагирования программ" (докторская диссертация, Мичиганский университет, Анн-Арбор, 1979).
[43] [назад] Чжао, Ж., и др. "Формализация промежуточного представления LLVM для проверенных преобразований программ". В Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 427-440, 2012.
[44] [назад] Чжу, Уильям, и Томборсон, Кларк. "Доказуемая схема гомоморфной обфускации в безопасности программного обеспечения". В CNIS, 208-212. ACTA Press, 2005.