r0064 r0064@mail.ru

Обуздать Windbg

Простые приемы сложного отладчика

Windbg – это мощный отладчик как для юзермодных приложений, так и для драйверов. Множество плагинов, команд, скриптовый язык... Все это может смутить и опытного в отладке человека. Особенно актуальна эта тема для реверсеров, которым пришлось полностью или частично перейти на 64-битные платформы.

ВЕДЬ, ПО СУТИ, НИ ОДИН ИМЕЮЩИЙСЯ X64-ОТЛАДЧИК НЕ СПОСОБЕН СРАВНИТЬСЯ С WINDBG ПО ФУНКЦИОНАЛУ, ПОЭТОМУ РЕЧЬ В ОСНОВНОМ ПОЙДЕТ ПРО 64-БИТНЫЕ ВЕРСИИ WINDOWS, НО И ОБЛАДАТЕЛИ 32-БИТНЫХ СМОГУТ ОБНАРУЖИТЬ ДЛЯ СЕБЯ ИНТЕРЕСНУЮ ИНФОРМАЦИЮ.

Патчим

WinDbg славится обилием различных расширений. Загрузка расширения делается командой:

```
.load имя_расширения
```

А выгрузка – командой `.unload`.

К сожалению, довольно удобный плагин для отладки ndis-драйверов, ndiskd, отказался работать в моем Windbg 6.11.1.404 (ndiskd.dll можно найти в WDK или в директории \Debugging Tools for Windows (x64)\winxp). Например, при попытке выполнить команду `!ndiskd.interfaces` ответ был неизменным: «Can't get offset of Link in NDIS_IF_BLOCK!». Аналогичный результат давало выполнение команд opens, protocols и остальных. Перезагрузка символов результата не давала, все структуры были в порядке. Отказываться от такого удобного расширения у меня не было желания, поэтому я решил прибегнуть к патчу, если это возможно (были бы исходники – можно было бы переписать, например). Загружаю плагин в Ida. Начнем исследование с команды interfaces. Все команды, имеющиеся в расширении – это экспортируемые функции (что очень упрощает мою задачу). То есть достаточно найти функцию interfaces в экспорте и проанализировать ее.

Что ж, приступим:

```
.text:0000000180012920   public interfaces
.text:0000000180012920   interfaces proc near

.text:0000000180012920   mov [rsp+arg_18], r9d
.text:0000000180012925   mov [rsp+arg_10], r8
.text:000000018001292A   mov [rsp+arg_8], rdx
.text:000000018001292F   mov [rsp+arg_0], rcx
.text:0000000180012934   push rbx
....
.text:00000001800129AF   lea r8, [rsp+0C8h+var_2C]
.text:00000001800129B7   lea rdx, aLink ; "Link"
.text:00000001800129BE   mov rcx, cs:NDIS_IF_BLOCK_NAME
; получаем смещение поля Link в структуре NDIS_IF_BLOCK
.text:00000001800129C5   call GetFieldOffset
.text:00000001800129CA   test eax, eax
```

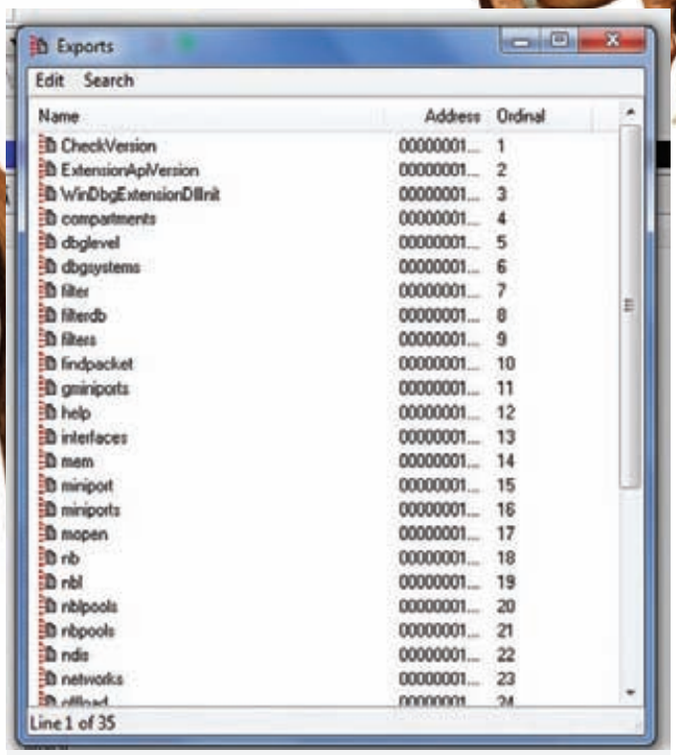
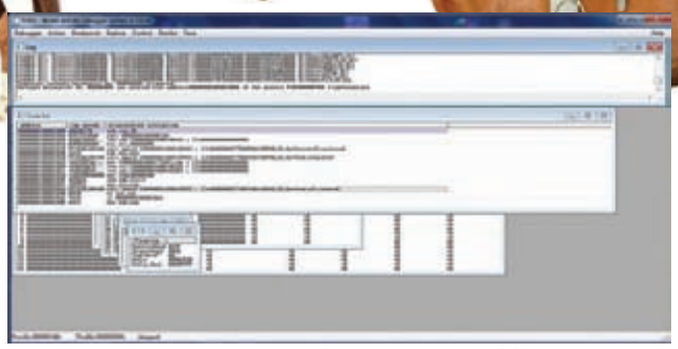


Таблица экспорта исследуемого плагина



Одна из альтернатив WinDbg – отладчик fdbg

```
.text:00000001800129CC   jz short loc_1800129E0
.text:00000001800129CE  lea rcx, aCanTGetOffs_22
      ; "Can't get offset of Link in NDIS_IF_BLO"...
.text:00000001800129D5
      call cs:ExtensionApis.lpOutputRoutine+4
.text:00000001800129DB  jmp loc_180012BF1
```

Видим вывод сообщения о том, что невозможно получить смещение поля Link в структуре NDIS_IF_BLOCK_NAME. Посмотрим, что за строка NDIS_IF_BLOCK_NAME.

```
.text:0000000180001260 aNdisNdis_if_bl db 'ndis!NDIS_
IF_BLOCK',0
```

Выполнение команды:

```
dt ndis!NDIS_IF_BLOCK
```

Сразу дает ответ, почему плагин рушится: «Symbol ndis!NDIS_IF_BLOCK not found». Тогда как dt ndis!_NDIS_IF_BLOCK нормально выводит структуру. Очевидно, надо изменить строки и ссылки на них, чтобы все работало нормально. Для нашей задачи воспользуемся каким-нибудь адекватным hex-редактором. Мне нравится 010 Editor. Исправим строки. В основном выравнивание позволяет нам безболезненно добавлять один символ в строку, не сдвигая остальные, но для пары строк это не так. Поэтому нужно найти ссылки на эти строки и исправить их. На каждую строку в данном случае по одной ссылке, находящейся в секции данных, на которую ссылаются инструкции, поэтому достаточно исправить ее. Например, на строку «ndis!LIST_ENTRY» ссылка выглядит так:

```
.data:0000000180018470 LIST_ENTRY_NAME dq offset
aNdis_list_entr ; DATA XREF: pktpools+5Dr
```

Смотрим, что в хексе это соответствует последовательности «68 14 00 80 01 00 00 00». Теперь нужно найти ее в хекс-редакторе и исправить первые байты.
После этого небольшого патча ndiskd выводит всю нужную информацию.

Отлавливаем загрузку драйвера

А как остановиться на DriverEntry отлаживаемого драйвера? Можно, конечно, поставить int 3 в начале функции, но не всегда хочется уродовать код брейками. Так может быть есть альтернативное решение? Оказывается, что есть. Как я уже говорил, WinDbg – мощный отладчик с огромным количеством возможностей, поэтому грех не воспользоваться ими снова. Но для начала немного теории. Непосредственный переход на точку входа драйвера осуществляет неэкспортируемая функция IopLoadDriver. Да, ты правильно понял, – неэкспортируемая – это значит, надо иметь отладочные символы (и вообще, без них отладка ядра может превратиться в настоящий ад). В моей Vista используется такой код (его можно найти или однократным трейсом или идой, дизассемблируя ядерную IopLoadDriver):

```
PAGE:00000001403AC40A loc_1403AC40A:
PAGE:00000001403AC40A
      ; IopLoadDriver+98Bj
PAGE:00000001403AC40A mov     rdx, rsi
PAGE:00000001403AC40D mov     rcx, rbx
PAGE:00000001403AC410 call    qword ptr [rbx+58h]
      ; DRIVER_OBJECT.DriverInit
```

Собственно, убедиться, что по смещению 0x58 в DRIVER_OBJECT находится именно DriverInit, можно в kd командой dt _DRIVER_OBJECT.
Что соответствует последовательности байт:

```
48 8B D6 48 8B CB FF 53  58
```

Чтобы найти искомый адрес, нужно ввести в командной строке WinDbg:

```
s nt!IopLoadDriver L2000 48 8B D6 48 8B CB FF 53  58
```

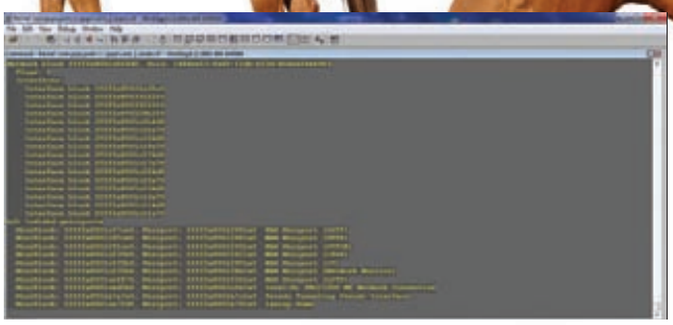
Где s – это команда поиска последовательности, nt!IopLoadDriver – адрес начала поиска, L2000 – количество байт, ограничивающее поиск, «48 8B D6 48 8B CB FF 53 58» – байты, которые мы ищем. Отладчик ответит что-то вроде:

```
fffff800`01c0940a  48 8b d6 48 8b cb ff 53-58 4c 8b
15 5e 20 dd ff  H..H...SXL..^ ..
```

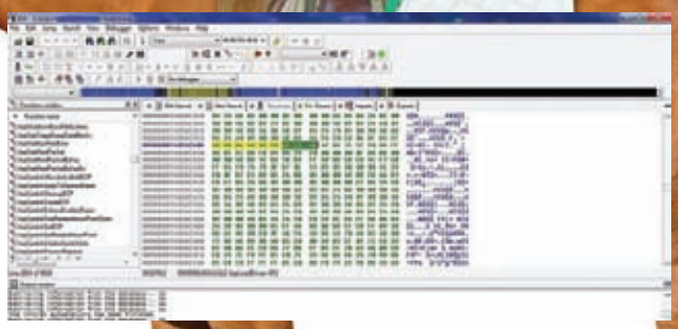
Вот и все. Теперь можно ставить точку останова на полученный адрес. В нашем случае – fffff800`01c0940a. Отныне мы будем получать управление как раз на переходе на точку входа. Этот метод довольно неплох, а что использовать – int 3 или его – решать тебе.

Скриптинг

Написание скриптов для WinDbg – это отдельная, очень большая тема. В рамках данной статьи я сделаю только введение. Для при-



Успешный патч расширения ndiskd – команды работают отлично



Нужные байты удобно копировать прямо из ida

мера напишем простой скрипт для расстановки бряков на всех Major-функциях в DRIVER_OBJECT'e, в качестве аргумента принимает указатель на driver object выбранного драйвера.

```
.block
{
.catch
{

  r $t0 = $arg1
  .printf "Driver object at 0x%I64X\n",@$t0
  r? $t1 = (nt!_DRIVER_OBJECT*)@$t0
  r $t0 = @@c++(@$t1->Type)
  .if(@$t0==4)
  {

  r $t0 = @@c++(@$t1->MajorFunction)
  .for(r $t1=0;@$t1<1c;r $t1=@$t1+1)
  {
     r $t2 = @$t0+@$t1*8
       r? $t3 = *(void**)@$t2
       .printf " Function at 0x%I64X\n",@$t3
    .if(@$t3!=0)
    {
      bp @$t3
      }
  }
  }.else
  {
    .printf "Not a driver object!\n"
  }
}
}
```

$arg1 – это первый аргумент (и так до $argN), t0-t19 – внутренние псевдо-регистры. Заметь, синтаксис очень похож на C++ (циклы for, while...). Сначала в скрипте проверяется поле DRIVER_OBJECT.Type, если оно равно четырем, то это действительно DRIVER_OBJECT (не забывай перед использованием этого скрипта подгружать символы!).

Потом получаем указатель на таблицу DRIVER_OBJECT.MajorFunction. Затем мотаем цикл по всем Major-функциям (всего их 27 - 0x1b). И как только мы находим подходящий указатель (не нулевой), ставим на него бряк (bp @$t3). Отладочные сообщения выводим функцией .printf. Скрипт написан с учетом размера 64-битного указателя (8 байт), что видно из строки «r $t2 = @$t0+@$t1*8». Чтобы скрипт заработал на 32-битной системе, нужно просто изменить размер указателя на 4. Обрати внимание – перед внутренним регистром во время операции присваивания ставится r, когда тип числовой, и r?, когда нечисловой.

Для тестирования скрипта нужно сначала получить указатель на объект-драйвер какого-нибудь драйвера с помощью команды !drvobj или иным способом. В примере ниже показано получение адреса driver object tdx.sys.

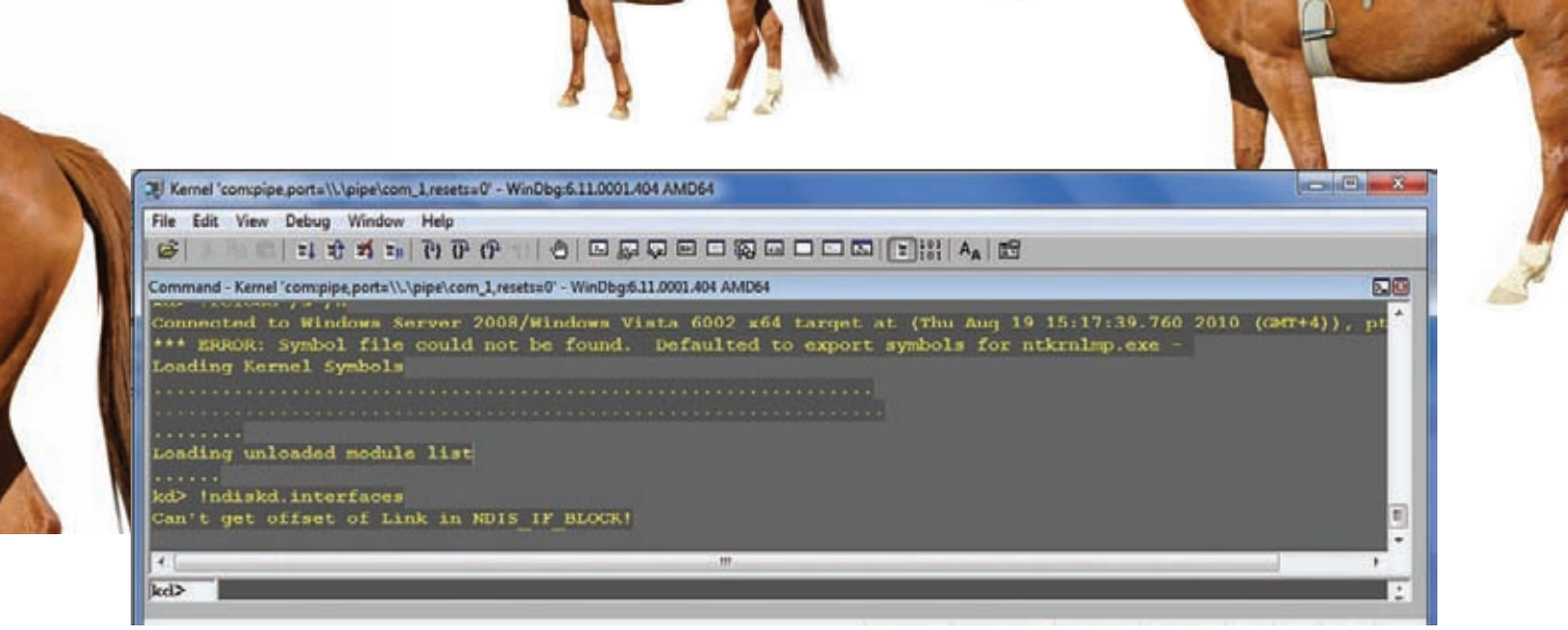
```
kd> !drvobj tdx
Driver object (fffffa8001ea1330) is for:
 \Driver\tdx
Driver Extension List: (id , addr)

Device Object list:
fffffa8001ec72f0  fffffa8001ec52f0  fffffa8001ec32f0
fffffa8001ec12f0
fffffa8001ebf2f0  fffffa8001ebd2f0  fffffa8001eb5300
```

Скрипты бывают многострочные и однострочные. Многострочные запускаются из командной строки $$><файл_скрипта, а однострочные – $$<текст_скрипта (или $<).

Если мы хотим передать скрипту аргументы, то надо писать $$>a<файл_скрипта (как раз наш случай).

Теперь модифицируем скрипт выше, чтобы он ставил брейк на конкретный обработчик device object'a, а не на все сразу.

```
$$ $arg1 - device object
$$ $arg2 - function number
.block
{
.catch
{
  r $t0 = $arg1
  .printf "Driver object at 0x%I64X\n",@$t0
  r? $t1 = (nt!_DRIVER_OBJECT*)@$t0
  r $t0 = @@c++(@$t1->Type)
  .if(@$t0==4)
  {
     r $t0 = @@c++(@$t1->MajorFunction)
       r $t1 = $arg2
       $$checking second argument
       .if(@$t1<1c)
       {
      r $t2 = @$t0+@$t1*8
        r? $t3 = *(void**)@$t2
        .printf " Function at 0x%I64X\n",@$t3
     .if(@$t3!=0)
     {
      bp @$t3
        u @$t3
       }
       .else
       {
         .printf "Invalid function address\n"
       }
       }
       .else
```



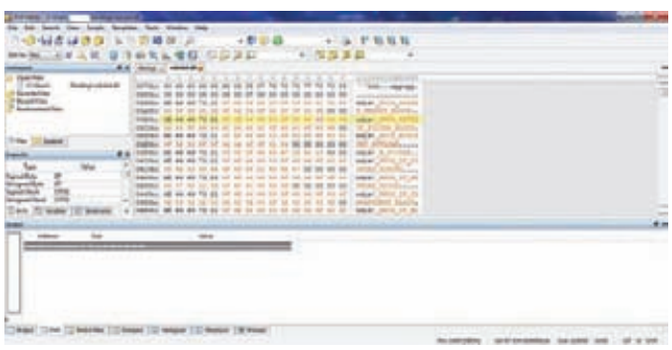
По непонятной причине плагин отказывается получать информацию из структуры

```
        {
          .printf "Invalid function number: must be
0-1B\n"
        }
  }.else
  {
    .printf "Not a driver object!\n"
  }
}
}
```

Комментарии начинаются с символов $$. Проверяем теперь два аргумента. Первый без изменений – поле DeviceObject.Type=4, а второй – это принадлежность к диапазону 0-1B, то есть допустимые обработчики от IRP_MJ_CREATE до IRP_MJ_PNP. Обрати внимание на строку u @$t3 – здесь осуществляется переход по адресу интересующего обработчика. Результат работы скрипта выглядит следующим образом (для разных вариантов входных аргументов):

```
kd> $$>a<c:\do2.wds fffffa8001ea1330 2
Driver object at 0xFFFFFA8001EA1330
 Function at 0xFFFFFA600A60D830
tdx!TdxTdiDispatchClose:
fffffa60`0a60d830     push rbx
fffffa60`0a60d832     sub rsp,20h
fffffa60`0a60d836     cmp rcx,qword ptr
      [tdx!TdxDeviceObject (fffffa60`0a61e650)]
fffffa60`0a60d83d     mov rax,qword ptr [rdx+0B8h]
fffffa60`0a60d844     mov rbx,rdx
fffffa60`0a60d847     je
      tdx!TdxTdiDispatchClose+0x71
(fffffa60`0a60d8a1)
fffffa60`0a60d849     mov rcx,qword ptr [rax+30h]
fffffa60`0a60d84d     cmp qword ptr [rcx+20h],2

kd> $$>a<c:\do2.wds fffffa8001ea1330 24
Driver object at 0xFFFFFA8001EA1330
Invalid function number: must be 0-1B

kd> $$>a<c:\do2.wds fffffa8001ea1350 7
Driver object at 0xFFFFFA8001EA1350
Not a driver object!
```

Вообще, это довольно простые задачи. Иногда требуются более сложные узкоспециализированные скрипты, рассмотрение которых выходит за рамки настоящей статьи. Кстати, сейчас появилось расширение, которое позволяет использовать питон в WinDbg – pykd (что очень хорошо, ведь не у всех есть время и желание осваивать встроенный скриптовый язык), правда он в стадии тестирования. Плагин требует предустановленный Python.



Исправляем строки в 010 Editor

Заключение

Конечно, рассмотренный здесь материал – это капля в море возможностей WinDbg. Я не коснулся, например, такой обширной темы, как написание плагинов. Чтобы освоить этот гигантский функционал, нужно время, терпение и опыт. Свои вопросы, как всегда, шли мне на e-mail.

Полезные ссылки

- **Сайт WinDbg плагина pykd, там же и примеры использования [en/ru]:**
  pykd.codeplex.com/wikipage?referringTitle=Home
- **Страница загрузки Debugging Tools for Windows, в состав которых входит WinDbg [en/ru]:**
  microsoft.com/whdc/devtools/debugging/default.mspx
- **Очень объемный документ, посвященный возможностям Windbg [en]:**
  windbg.info/download/doc/pdf/WinDbg_A_to_Z_color.pdf
- **Очень краткое введение в скриптовый язык обсуждаемого отладчика [en]:**
  dumpanalysis.org/WCDA/WCDA-Sample-Chapter.pdf