r0064 r0064@mail.ru

Многоразрядные шелл-коды

ПИШЕМ RINGO-SHELLCODE ПОД WINDOWS X64

Для тебя наверняка не секрет, что ядро 64-битной Windows подверглось значительным изменениям. Это, в первую очередь, касается ряда системных структур и функций. А значит, на 64-битной винде привычные способы написания шелл-кодов становятся совершенно бесполезными, посему приходится брать дизассемблер и адаптироваться к новым условиям. Копаться в этих самых структурах, сравнивать и анализировать. Чем мы сегодня с тобой и займемся!

Что примечательно — никто до меня эту тему не расписывал. Да и я сам за все время существования x64 видел только пример шелл-кода **ring3** (inj3ct0r.com/exploits/9740). Под нулевое кольцо мне пока ничего увидеть не довелось. Будем это дело исправлять — ведь, в конце концов, область применения позиционно-независимого кода чрезвычайно широка: от вполне легальных пакеров/протекторов до руткитов и эксплойтов.

ЧТО НУЖНО ДЛЯ РАБОТЫ?

Для компиляции драйвера (мы ведь пишем шелл-код ring0) тебе потребуется Macro Assembler x64 (ml64). Его можно утянуть из WDK. Сам WDK доступен для скачивания на сайте Мелкософта по ссылке: microsoft.com/whdc/DevTools/WDK/WDKpkg.mspx.

После компиляции драйвер можно будет загрузить с помощью набора консольных утилит из примеров для FASM (install_drv.exe, start_drv.exe и пр.). И только потому, что мне было лень написать свою утилиту (сваяю в ближайшее время). Скачать их можно по ссылке: flatassembler.net/examples/win64_drivers.zip. Помимо всего прочего тебе понадобятся Microsoft Debugging Tools (64-битные версии) + livekd Руссиновича (нужен для просмотра смещений в ядерных структурах). Первое и второе можно скачать на сайте Microsoft: microsoft.com/whdc/devtools/debugging/install64bit.mspx и technet.microsoft.com/ru-ru/sysinternals/bb897415.aspx. Чтобы лицезреть отладочный вывод своего новоиспеченного драйвера, тебе потребуется DbgView. Он себя (что очень приятно) прекрасно чувствует на 64-битной винде, как, впрочем, и livekd (technet.microsoft.com/en-us/sysinternals/bb896647.aspx).

Дизассемблировать ядерные модули будем с помощью IDA x64.

Ну и для успешного восприятия того, что здесь написано, ты должен быть знаком с форматом PE и хотя бы в общих чертах понимать, что представляет собой 64-битный ассемблер. К вышесказанному надо еще добавить желание разобраться в тонкостях 64-битного пикодинга :).

ПЛАН ДЕЙСТВИЙ

Обрисую задачи, которые нам необходимо будет решить при написании базонезависимого кода. Чтобы в голове все уложилось, всегда лучше следовать определенному плану.

1. Получение адреса начала ядра (aka ntoskrnl);
2. Разбор таблицы экспорта ntoskrnl;
3. Получение адресов нужных функций;
4. Profit!

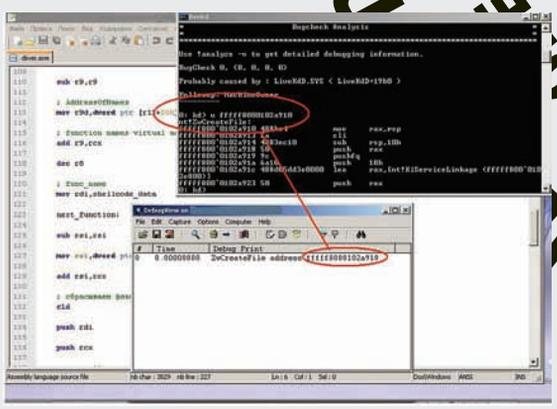
Итак, начнем по порядку...

ПОИСК БАЗЫ ЯДРА

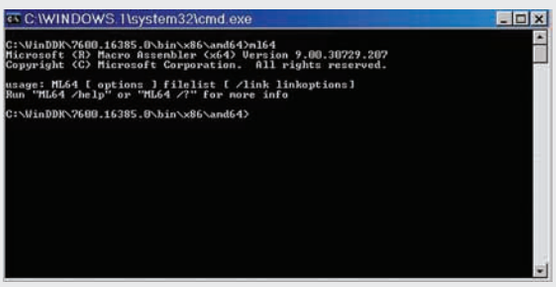
Первым делом нам необходимо получить базу ядра. Я рассмотрю три способа получения:

- через поля структуры Processor Control Region (сокращенно PCR);
- юзая инструкцию sidt;
- через msr (они же машинно-зависимые регистры).

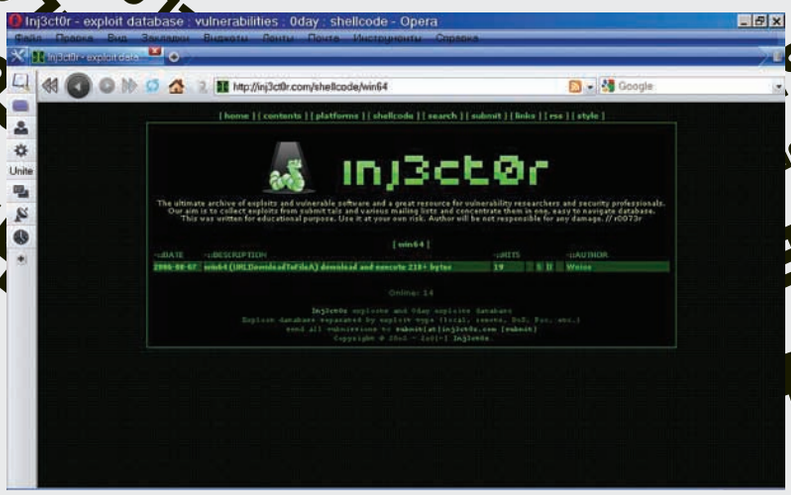
Конечно же, способы получения адреса начала ядра не исчерпываются



Вывод нашего драйвера в DbgView



Родной интерфейс ml64



Единственный x64 шелл-код, обнаруженный мной и мирно лежащий на inj3ct0r.com

только этими тремя. Более того, здесь все ограничивается только фантазией кодера; в мои цели входит лишь расставить путеводные флажки для твоих дальнейших исследований. Что ж, приступим!

СПОСОБ №1

Суть как этого, так и других способов в том, что мы должны отыскать какой-то адрес, принадлежащий ntoskrnl. Такой адрес можно получить из, скажем, обработчика прерывания int 3 из idt. Выбор допустимого вектора определяется тем, принадлежит ли он обычно ядру. Я говорю «обычно», так как могут стоять различные хуки. Хотя на x64 idt проверяется Patch Guard, его защиту можно без труда обойти. В вопросе поиска подходящего обработчика нам поможет livekd и его команда !idt, которая дампит таблицу прерываний.

```
0: kd> !idt
Dumping IDT:
00: fffff8000102c400 nt!KiDivideErrorFault
01: fffff8000102c4c0 nt!KiDebugTrapOrFault
02: fffff8000102c600 nt!KiNmiInterrupt
                Stack = 0xFFFFF8000011D000
03: fffff8000102c940 nt!KiBreakpointTrap
04: fffff8000102ca00 nt!KiOverflowTrap
05: fffff8000102cac0 nt!KiBoundFault
06: fffff8000102cb80 nt!KiInvalidOpcodeFault
07: fffff8000102cd40 nt!KiNpxNotAvailableFault
08: fffff8000102ce00 nt!KiDoubleFaultAbort
                Stack = 0xFFFFF8000011B000
09: fffff8000102cec0 nt!KiNpxSegmentOverrunAbort
0a: fffff8000102cf80 nt!KiInvalidTssFault
0b: fffff8000102d040 nt!KiSegmentNotPresentFault
0c: fffff8000102d140 nt!KiStackFault
0d: fffff8000102d240 nt!KiGeneralProtectionFault
0e: fffff8000102d340 nt!KiPageFault
10: fffff8000102d680 nt!KiFloatingErrorFault
11: fffff8000102d7c0 nt!KiAlignmentFault
12: fffff8000102d880 nt!KiMcheckAbort
                Stack = 0xFFFFF8000011F000
13: fffff8000102dbc0 nt!KiXmmException
1f: fffff800010279e0 nt!KiApcInterrupt
2c: fffff8000102dd40 nt!KiRaiseAssertion
2d: fffff8000102de00 nt!KiDebugServiceTrap
2f: fffff80001067c70 nt!KiDpcInterrupt
```

Итого 20+ возможных вариантов, что не так уж мало. Но, чтобы определить адрес обработчика, нам потребуется адрес начала idt. Один из способов ее получения — чтение поля KPCR.IdtBase. О виде структуры PCR спросим, как обычно, у livekd.

```
0: kd> dt _KPCR
nt!_KPCR
+0x000 NtTib            : _NT_TIB
+0x000 GdtBase          : Ptr64 _KGDTENTRY64
+0x008 TssBase          : Ptr64 _KTSS64
+0x010 PerfGlobalGroupMask : Ptr64 Void
+0x018 Self             : Ptr64 _KPCR
+0x020 CurrentPrcb      : Ptr64 _KPRCB
+0x028 LockArray        : Ptr64 _KSPIN_LOCK_QUEUE
+0x030 Used_Self        : Ptr64 Void
+0x038 IdtBase          : Ptr64 _KIDTENTRY64
+0x040 Unused           : [2] Uint8B
+0x050 Irql             : UChar
+0x051 SecondLevelCacheAssociativity : UChar
+0x052 ObsoleteNumber   : UChar
+0x053 Fill0            : UChar
+0x054 Unused0          : [3] Uint4B
+0x060 MajorVersion     : Uint2B
+0x062 MinorVersion     : Uint2B
+0x064 StallScaleFactor : Uint4B
+0x068 Unused1          : [3] Ptr64 Void
+0x080 KernelReserved   : [15] Uint4B
+0x0bc SecondLevelCacheSize : Uint4B
+0x0c0 HalReserved      : [16] Uint4B
+0x100 Unused2          : Uint4B
+0x108 KdVersionBlock   : Ptr64 Void
+0x110 Unused3          : Ptr64 Void
+0x118 PcrAlign1        : [24] Uint4B
+0x180 Prcb             : _KPRCB
```

Как ты можешь заметить, указатели расширились до 64 бит, ну и имена полей (сравнивая с 32-разрядной Виндой) тоже поменялись. Но это не самое главное, так как для нас в данный момент важно смещение поля IdtBase относительно начала _PCR.

▸ dvd

Исходники скомпилированного драйвера ищи на нашем DVD.

Хорошо, допустим, структуру x64 PCR мы мало-мальски знаем (честь и хвала livekd). А откуда мы достанем указатель на KPCR? Глянем в код функции из hal.dll HalInitializeProcessor.

```
.text:000000008001F240  public HalInitializeProcessor
.text:000000008001F240  HalInitializeProcessor proc near
; DATA XREF: .pdata:000000008004C804 o
.text:000000008001F240
.text:000000008001F240 var_28 = byte ptr -28h
.text:000000008001F240 var_20 = byte ptr -20h
.text:000000008001F240 var_18 = qword ptr -18h
.text:000000008001F240 arg_0 = byte ptr  8
.text:000000008001F240
.text:000000008001F240  push    rbx
.text:000000008001F242  sub     rsp, 40h
.text:000000008001F246  mov     r8, gs:18h
.text:000000008001F24F  mov     r10d, ecx
.text:000000008001F252  mov     r9d, 1
.text:000000008001F258  mov     rax, [r8+20h]
.text:000000008001F25C  mov     ecx, ecx
.text:000000008001F25E  mov     [rax+4], r10b
.text:000000008001F262  shl     r9, cl
.text:000000008001F265  lea     rax, HalpProcessorPCR
.text:000000008001F26C  or    cs:HalpActiveProcessors, r9
.text:000000008001F273  cmp   cs:HalpStaticIntAffinity, 0
.text:000000008001F27A  mov     dword ptr [r8+64h], 64h
.text:000000008001F282  mov     [rax+r10*8], r8
```

Как видим, указатель на PCR для текущего процессора мы можем утянуть из gs:[18h]. В 32-битной Винде юзали сегментный регистр fs, а теперь gs :).

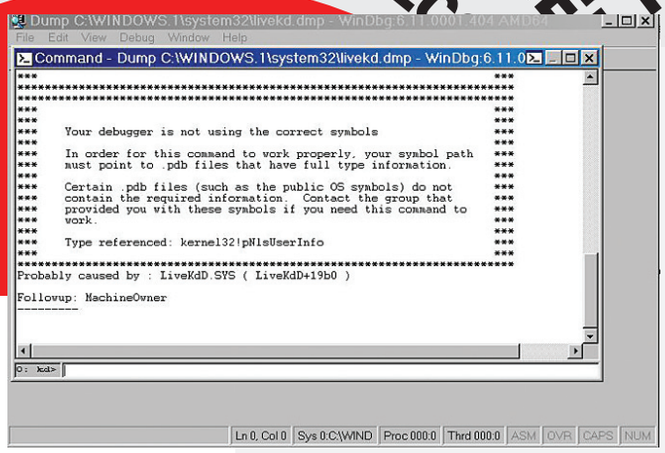
Так, а что стало с дескрипторами шлюзов в idt? Нам ведь надо достать адрес обработчика int 3 (ну или другого, по вкусу), а у нас только дескрипторы шлюзов, по которым «размазано» смещение обработчика. Они, конечно же, в x64 тоже поменялись. Как именно? Изменения можно (и нужно) смотреть в оригинальных мануалах производителя (AMD, Intel), но у нас сейчас под рукой только livekd. И с его помощью тоже можно узнать нужную инфу! Вводим dt _KIDTENTRY64, чтобы вытянуть из отладочных символов структуру дескриптора шлюза.

```
0: kd> dt _KIDTENTRY64
nt!_KIDTENTRY64
   +0x000 OffsetLow        : Uint2B
   +0x002 Selector         : Uint2B
   +0x004 IstIndex         : Pos 0, 3 Bits
   +0x004 Reserved0        : Pos 3, 5 Bits
   +0x004 Type             : Pos 8, 5 Bits
   +0x004 Dpl              : Pos 13, 2 Bits
   +0x004 Present          : Pos 15, 1 Bit
   +0x006 OffsetMiddle     : Uint2B
   +0x008 OffsetHigh       : Uint4B
   +0x00c Reserved1        : Uint4B
   +0x000 Alignment        : Uint8B
0: kd>
```

Из этого дампа мы должны понять, где внутри дескриптора располагаются кусочки адреса обработчика (он же Offset). Адрес, как ты уже понял, тоже 64-битный.

Вот, собственно, и все, что нам нужно. Ниже привожу код с комментариями, который и получает базу ядра этим способом.

```
; _KPRC
mov rcx, gs:[18h]
; +0x038 IdtBase          : Ptr64 _KIDTENTRY64
mov rcx,qword ptr[rcx+38h]
```



Если консоль livekd тебе не по душе — запускай livekd с параметром -w

```
; получение адреса обработчика int 3
; Размер дескриптора шлюза теперь 16 байт. Нужен третий обработчик от 0
; INT_X — константа = 3
add rcx,16*INT_X
; собираем вместе поля OffsetLow, OffsetMiddle и OffsetHigh структуры KIDTENTRY64
mov r11,qword ptr [rcx]
and r11,0FFFFh
mov rcx, qword ptr [rcx+4]
; теперь в rcx адрес обработчика int 3 (Offset)
mov cx,r11w
and cx,0F000h  ; обнуляем младшие 12 бит адреса
search_loo:    ; цикл поиска базы ntoskrnl
cmp word ptr [rcx],'ZM'
jnz nxt
sub rax,rax
; eax -> PE offset
mov eax,dword ptr [rcx+3Ch]
; проверка сигнатуры PE
cmp word ptr [rcx+rax],'EP'
jz founded
nxt:
; продолжаем поиски...
sub rcx,1000h ; так быстрее всего
jmp search_loo
founded:
...
```

СПОСОБ №2

Этот метод также основывается на idt. Он несколько лаконичней, чем предыдущий, но также имеет ряд особенностей по сравнению с аналогичным способом для 32-разрядной версии Windows. Основная причина различий — изменение формата регистра idtr. Поле лимита не поменялось (как было 2 байта, так и осталось), а вот поле базы стало равным 8 байтам.

Структура idtr на x64 следующая:

```
typedef struct _IDTR
{
 USHORT usLimit;
 ULONGLONG uBase;
}IDTR;
```

Получив базу idt, действуем инструкцией sidt так же, как и в предыдущем примере. Обнуляем младшие 12 бит адреса и, вычитая по 1000h, проверяем сигнатуры MZ и PE.

СПОСОБ № 3

Еще один красивый способ заключается в получении адреса KiSystemCall64 (так называется обработчик в 64-битном ядре) из msr регистра lstar (его адрес 0xC0000082). Этот регистр (как, вероятно, тебе известно) используется командой syscall.

```
...
sub rcx,rcx
mov ecx,0C0000082h ; адрес msr в ecx
rdmsr ; читаем машинно-зависимый регистр lstar
...
```

После этого в паре регистров edx:eax будет содержаться адрес KiSystemCall64. Кстати, вовсе необязательно читать именно регистр lstar. Вообще, инструкция syscall в long mode (режим процессора, в котором работает 64-разрядная винда) юзает 2 msr регистра: cstar и lstart для compatibility и 64-bit mode соответственно. Так к чему это я веду? В msr cstar (0xC0000083) тоже лежит адрес, принадлежащий диапазону адресов ntoskrnl! — это адрес процедуры KiSystemCall32. Далее по уже известной тебе схеме получаем адрес начала ядра.

РАЗБОР ЭКСПОРТА В WINDOWS X64

Теперь приступаем к получению адресов нужных функций. Так как мы имеем дело с форматом PE32+, надо учитывать его специфику. Обрисую основные моменты, которые поменялись. Если ты имеешь опыт написания шелл-кодов для win32, то наверняка знаешь, что получить указатель на директорию экспорта можно, прибавив к адресу начала IMAGE_NT_HEADERS 78h. В win64 мы прибавляем 88h. Увеличение значения величины смещения обусловлено увеличением некоторых полей в PE32+ по сравнению с PE32.
Взглянем на структуру _IMAGE_NT_HEADERS64:

```
typedef struct _IMAGE_NT_HEADERS64 {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

То есть изменения коснулись полей опционального заголовка, из-за этого пришлось поменять смещения. Кстати IMAGE_EXPORT_DIRECTORY (как и IMAGE_DOS_HEADER) не поменялась, что играет нам на руку.
В заключение привожу код разбора таблицы экспорта для PE32+:

```
...
    shellcode_data:
    db "ZwCreateFile",0
    ;...
    ; ищем адрес нужной функции
    ; export directory
    lea r11,[rcx+rax+88h]
    sub r12,r12
    ; export directory rva
    mov r12d,dword ptr [r11]
    ; Виртуальный адрес IMAGE_EXPORT_DIRECTORY
    add r12,rcx
    sub r8,r8
    ;number of functions
    mov r8d,dword ptr [r12+18h]
    sub r9,r9
    ; AddressOfNames
    mov r9d,dword ptr [r12+20h]
    ; function names virtual address
    add r9,rcx
    dec r8
    ; func_name
    mov rdi,shellcode_data
    next_function:
    sub rsi,rsi
    mov esi,dword ptr [r9+r8*4]
    add rsi,rcx
    ; сбросим флаг направления, чтобы при команде
cmpsb происходил инкремент регистров
    cld
    push rdi
   ; сохраняем rcx — команда cmpsb его модифицирует
    push rcx
   ; устанавливаем счетчик символов для команды cmpsb
    mov rcx,12
   ; начинаем сравнивать строки посимвольно
    repe cmpsb
    jz founded_f
    pop rcx ; восстанавливаем значения модифицируемых
cmpsb регистров
    pop rdi
   ; уменьшаем счетчик функций
    dec r8
    jnz next_function
    jmp not_found
    founded_f:
    pop rcx
    pop rdi
    sub rbx,rbx
    ; AddressOfNameOrdinals RVA
    mov ebx,dword ptr [r12+24h]

    ;NameOrdinals VA
    add rbx,rcx
    ; index in address table into r8
    mov r8w, word ptr [rbx+r8*2] ; кладем значение
индекса в младшее слово регистра r8
    and r8d,0FFFFh
    sub rbx,rbx
    mov ebx,dword ptr [r12+1Ch]
    add rbx,rcx
    sub r12,r12
    mov r12d, [rbx+r8*4] ; кладем в младшее двойное
слово регистра r12 RVA нужной нам функции
; в регистре rcx получаем адрес функции ZwCreateFile
    add rcx,r12
    not_found:
...
```

Как видишь, этот код имеет два существенных недостатка. Во-первых, я не хеширую имя функции, а использую посимвольное сравнение. Во-вторых, отсутствует оптимизация.
Теперь компилируем драйвер с помощью ml64, а отладочный вывод смотрим в DbgView.
Правильность полученного адреса можно проверить в livekd. Для этого введем команду:

```
u полученный_адрес
```

ЗАКЛЮЧЕНИЕ

Вот мы и разобрались с ядерным шелл-кодингом под 64-битную Винду. На первый взгляд ничего сложного. Вообще, надо сказать, что 64-разрядные процессоры очень привлекательны для написания пи-кода. Все дело в таких нововведениях, как rip-relative addressing и большее количество регистров. Правда, некоторые вещи я намеренно упустил из виду — не приводить же совсем законченное решение :). С этим тебе, читатель, еще предстоит разобраться, и да поможет тебе дизассемблер!